© 2001, 2002 Advanced Micro Devices, Inc. All rights reserved.

The contents of this document are provided in connection with Advanced Micro Devices, Inc. ("AMD") products. AMD makes no representations or warranties with respect to the accuracy or completeness of the contents of this publication and reserves the right to make changes to specifications and product descriptions at any time without notice. No license, whether express, implied, arising by estoppel or otherwise, to any intellectual property rights is granted by this publication. Except as set forth in AMD's Standard Terms and Conditions of Sale, AMD assumes no liability whatsoever, and disclaims any express or implied warranty, relating to its products including, but not limited to, the implied warranty of merchantability, fitness for a particular purpose, or infringement of any intellectual property right.

AMD's products are not designed, intended, authorized or warranted for use as components in systems intended for surgical implant into the body, or in other applications intended to support or sustain life, or in any other application in which the failure of AMD's product could create a situation where personal injury, death, or severe property or environmental damage may occur. AMD reserves the right to discontinue or make changes to its products at any time without notice.

Trademarks
AMD, the AMD Arrow logo, AMD Athlon, and combinations thereof, 3DNow!, AMD-751, and Super7 are trademarks, and AMD-K6 and AMD-K6-2 are registered trademarks of Advanced Micro Devices, Inc.

Microsoft, Windows, and Windows NT are registered trademarks of Microsoft Corporation.

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

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

List of Figures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xiii
List of Tables . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xv
Revision History . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . xvii

Chapter 1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
About This Document . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
AMD Athlon™ Processor Family . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
AMD Athlon Processor Microarchitecture Summary . . . . . . . . . . . . . . 4

Chapter 2 Top Optimizations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
Optimization Star . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
Group I Optimizations—Essential Optimizations . . . . . . . . . . . . . . . . . 8
  Memory-Size and Alignment Issues . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
  Use the 3DNow!™ Prefetching Instructions . . . . . . . . . . . . . . . . . . . . . . . 9
  Select DirectPath Over VectorPath Instructions . . . . . . . . . . . . . . . . . . 10
Group II Optimizations—Secondary Optimizations . . . . . . . . . . . . . . . . 10
  Load-Execute Instruction Usage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
  Take Advantage of Write Combining . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
  Optimizing Main Memory Performance for Large Arrays . . . . . . . . 12
  Use 3DNow! Instructions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
  Recognize 3DNow! Professional Instructions . . . . . . . . . . . . . . . . . . . 14
  Avoid Branches Dependent on Random Data . . . . . . . . . . . . . . . . . . . 14
  Avoid Placing Code and Data in the Same 64-Byte Cache Line . . . . 15
Chapter 3 C Source-Level Optimizations

Ensure Floating-Point Variables and Expressions are of Type Float ........................................ 17
Use 32-Bit Data Types for Integer Code ........................................ 17
Consider the Sign of Integer Operands ........................................ 18
Use Array-Style Instead of Pointer-Style Code ............................. 20
Completely Unroll Small Loops ........................................ 22
Avoid Unnecessary Store-to-Load Dependencies ......................... 23
Always Match the Size of Stores and Loads .............................. 24
Consider Expression Order in Compound Branch Conditions ....... 27
Switch Statement Usage ........................................ 28
Use Prototypes for All Functions ........................................ 29
Use Const Type Qualifier ........................................ 29
Generic Loop Hoisting ........................................ 30
Declare Local Functions as Static ........................................ 32
Dynamic Memory Allocation Consideration .............................. 33
Introduce Explicit Parallelism into Code ................................ 33
Explicitly Extract Common Subexpressions .............................. 35
C Language Structure Component Considerations ..................... 36
Sort Local Variables According to Base Type Size ..................... 37
Accelerating Floating-Point Divides and Square Roots ................. 38
Fast Floating-Point-to-Integer Conversion ................................ 40
Speeding Up Branches Based on Comparisons Between Floats ...... 42
Avoid Unnecessary Integer Division ..................................... 44
Copy Frequently Dereferenced Pointer Arguments to Local Variables ........................................ 44
Use Block Prefetch Optimizations ..................................... 46
Chapter 4  Instruction Decoding Optimizations .............................................. 49

Overview ................................................. 49

Select DirectPath Over VectorPath Instructions ................................. 50

Load-Execute Instruction Usage ......................................................... 50

  Use Load-Execute Integer Instructions ........................................ 50

  Use Load-Execute Floating-Point Instructions with Floating-Point
  Operands ............................................................................. 51

  Avoid Load-Execute Floating-Point Instructions with Integer
  Operands ............................................................................. 51

Use Read-Modify-Write Instructions Where Appropriate ............... 52

Align Branch Targets in Program Hot Spots ..................................... 54

Use 32-Bit LEA Rather than 16-Bit LEA Instruction .......................... 54

Use Short Instruction Encodings ....................................................... 54

Avoid Partial-Register Reads and Writes .......................................... 55

Use LEAVE Instruction for Function Epilogue Code ....................... 56

Replace Certain SHLD Instructions with Alternative Code ................. 57

Use 8-Bit Sign-Extended Immediates ............................................... 57

Use 8-Bit Sign-Extended Displacements ......................................... 58

Code Padding Using Neutral Code Fillers ....................................... 58

  Recommendations for AMD-K6® Family and AMD Athlon Processor
  Blended Code ................................................................... 59
Chapter 5  Cache and Memory Optimizations ............................................ 63

Memory Size and Alignment Issues .............................................. 63
  Avoid Memory-Size Mismatches ............................................. 63
  Align Data Where Possible .................................................. 65

Optimizing Main Memory Performance for Large Arrays ............. 66
  Memory Copy Optimization .................................................. 67
  Array Addition ................................................................. 74
  Summary ................................................................. 78

Use the PREFETCH 3DNow!™ Instruction ...................................... 79
  Determining Prefetch Distance .......................................... 83

Take Advantage of Write Combining ........................................... 85

Avoid Placing Code and Data in the Same 64-Byte Cache Line....... 85

Multiprocessor Considerations ................................................. 86
  Store-to-Load Forwarding Restrictions ................................ 86
    Store-to-Load Forwarding Pitfalls—True Dependencies ......... 87
    Summary of Store-to-Load Forwarding Pitfalls to Avoid ....... 90

Stack Alignment Considerations ............................................... 90
  Align TBYTE Variables on Quadword Aligned Addresses .......... 91
  C Language Structure Component Considerations .................. 91
  Sort Variables According to Base Type Size ....................... 92
### Chapter 6 Branch Optimizations

**Avoid Branches Dependent on Random Data**  
AMD Athlon Processor Specific Code  
Blended AMD-K6 and AMD Athlon Processor Code

**Always Pair CALL and RETURN**

**Recursive Functions**

**Replace Branches with Computation in 3DNow! Code**  
Muxing Constructs  
Sample Code Translated into 3DNow! Code

**Avoid the Loop Instruction**

**Avoid Far Control Transfer Instructions**

### Chapter 7 Scheduling Optimizations

**Schedule Instructions According to their Latency**

**Unrolling Loops**  
Complete Loop Unrolling  
Partial Loop Unrolling

**Use Function Inlining**  
Overview  
Always Inline Functions if Called from One Site  
Always Inline Functions with Fewer than 25 Machine Instructions

**Avoid Address Generation Interlocks**

**Use MOVZX and MOVSX**

**Minimize Pointer Arithmetic in Loops**

**Push Memory Data Carefully**
Chapter 8 \hspace{3cm} \textbf{Integer Optimizations} \hspace{3cm} 115

Replace Divides with Multiplies \hspace{1cm} 115
Multiplication by Reciprocal (Division) Utility \hspace{1cm} 116
Unsigned Division by Multiplication of Constant \hspace{1cm} 116
Signed Division by Multiplication of Constant \hspace{1cm} 118

Consider Alternative Code When Multiplying by a Constant \hspace{1cm} 120

Use MMX™ Instructions for Integer-Only Work \hspace{1cm} 123

Repeated String Instruction Usage \hspace{1cm} 123
Latency of Repeated String Instructions \hspace{1cm} 123
Guidelines for Repeated String Instructions \hspace{1cm} 124

Use XOR Instruction to Clear Integer Registers \hspace{1cm} 125

Efficient 64-Bit Integer Arithmetic \hspace{1cm} 125

Efficient Implementation of Population Count Function \hspace{1cm} 136

Efficient Binary-to-ASCII Decimal Conversion \hspace{1cm} 139

Derivation of Multiplier Used for Integer Division by Constants \hspace{1cm} 144

Derivation of Algorithm, Multiplier, and Shift Factor for Unsigned Integer Division \hspace{1cm} 144

Derivation of Algorithm, Multiplier, and Shift Factor for Signed Integer Division \hspace{1cm} 148
Chapter 9  **Floating-Point Optimizations** ........................................ 151

Ensure All FPU Data is Aligned ........................................ 151
Use Multiples Rather than Divides .................................. 151
Use FFREEP Macro to Pop One Register from the FPU Stack ... 152
Floating-Point Compare Instructions .................................... 153
Use the FXCH Instruction Rather than FST/FLD Pairs .......... 153
Avoid Using Extended-Precision Data ................................. 154
Minimize Floating-Point-to-Integer Conversions .................. 154
Check Argument Range of Trigonometric Instructions Efficiently 157
Take Advantage of the FSINCOS Instruction ....................... 159

Chapter 10  **3DNow!™ and MMX™ Optimizations** ......................... 161

Use 3DNow! Instructions ............................................... 161
Use FEMMS Instruction ............................................... 162
Use 3DNow! Instructions for Fast Division ......................... 162
  Optimized 14-Bit Precision Divide ................................. 162
  Optimized Full 24-Bit Precision Divide ......................... 163
  Pipelined Pair of 24-Bit Precision Divides ................... 163
  Newton-Raphson Reciprocal ........................................ 164
Use 3DNow! Instructions for Fast Square Root and Reciprocal Square Root ............................................... 165
  Optimized 15-Bit Precision Square Root ....................... 165
  Optimized 24-Bit Precision Square Root ....................... 165
  Newton-Raphson Reciprocal Square Root ....................... 166
Use MMX PMADDWD Instruction to Perform Two 32-Bit Multiplies in Parallel ............................................... 167
Use PMULHUW to Compute Upper Half of Unsigned Products . 167
3DNow! and MMX Intra-Operand Swapping .......................... 169
Fast Conversion of Signed Words to Floating-Point .............. 170
<table>
<thead>
<tr>
<th>Table of Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width of Memory Access Differs Between PUNPCKL* and PUNPCKH* .......................... 171</td>
</tr>
<tr>
<td>Use MMX PXOR to Negate 3DNow! Data .......................................................... 172</td>
</tr>
<tr>
<td>Use MMX PCMP Instead of 3DNow! PFCMP ................................................................ 173</td>
</tr>
<tr>
<td>Use MMX Instructions for Block Copies and Block Fills ........................................ 174</td>
</tr>
<tr>
<td>Efficient 64-Bit Population Count Using MMX Instructions ..................................... 184</td>
</tr>
<tr>
<td>Use MMX PXOR to Clear All Bits in an MMX Register .............................................. 185</td>
</tr>
<tr>
<td>Use MMX PCMPEQD to Set All Bits in an MMX Register ............................................ 186</td>
</tr>
<tr>
<td>Use MMX PAND to Find Floating-Point Absolute Value in 3DNow! Code ....................... 186</td>
</tr>
<tr>
<td>Integer Absolute Value Computation Using MMX Instructions ................................... 186</td>
</tr>
<tr>
<td>Optimized Matrix Multiplication ............................................................................ 187</td>
</tr>
<tr>
<td>Efficient 3D-Clipping Code Computation Using 3DNow! Instructions ......................... 190</td>
</tr>
<tr>
<td>Efficiently Determining Similarity Between RGBA Pixels ......................................... 192</td>
</tr>
<tr>
<td>Use 3DNow! PAVGUSB for MPEG-2 Motion Compensation ............................................ 195</td>
</tr>
<tr>
<td>Efficient Implementation of floor() Using 3DNow! Instructions ............................... 197</td>
</tr>
<tr>
<td>Stream of Packed Unsigned Bytes ............................................................................ 198</td>
</tr>
<tr>
<td>Complex Number Arithmetic ..................................................................................... 199</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 11</th>
<th>General x86 Optimization Guidelines ......................................................... 201</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Forms</td>
<td>................................................................................. 201</td>
</tr>
<tr>
<td>Dependencies</td>
<td>.............................................................................. 202</td>
</tr>
<tr>
<td>Register Operands</td>
<td>............................................................................. 202</td>
</tr>
<tr>
<td>Stack Allocation</td>
<td>............................................................................... 202</td>
</tr>
</tbody>
</table>
## Appendix A  
**AMD Athlon™ Processor Microarchitecture**  

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>203</td>
</tr>
<tr>
<td>AMD Athlon Processor Microarchitecture</td>
<td>204</td>
</tr>
<tr>
<td>Superscalar Processor</td>
<td>204</td>
</tr>
<tr>
<td>Instruction Cache</td>
<td>205</td>
</tr>
<tr>
<td>Predecode</td>
<td>206</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>206</td>
</tr>
<tr>
<td>Early Decoding</td>
<td>207</td>
</tr>
<tr>
<td>Instruction Control Unit</td>
<td>208</td>
</tr>
<tr>
<td>Data Cache</td>
<td>208</td>
</tr>
<tr>
<td>Integer Scheduler</td>
<td>209</td>
</tr>
<tr>
<td>Integer Execution Unit</td>
<td>209</td>
</tr>
<tr>
<td>Floating-Point Scheduler</td>
<td>210</td>
</tr>
<tr>
<td>Floating-Point Execution Unit</td>
<td>211</td>
</tr>
<tr>
<td>Load-Store Unit (LSU)</td>
<td>212</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>213</td>
</tr>
<tr>
<td>Write Combining</td>
<td>213</td>
</tr>
<tr>
<td>AMD Athlon System Bus</td>
<td>214</td>
</tr>
</tbody>
</table>

## Appendix B  
**Pipeline and Execution Unit Resources Overview**  

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch and Decode Pipeline Stages</td>
<td>215</td>
</tr>
<tr>
<td>Integer Pipeline Stages</td>
<td>218</td>
</tr>
<tr>
<td>Floating-Point Pipeline Stages</td>
<td>220</td>
</tr>
<tr>
<td>Execution Unit Resources</td>
<td>222</td>
</tr>
<tr>
<td>Terminology</td>
<td>222</td>
</tr>
<tr>
<td>Integer Pipeline Operations</td>
<td>223</td>
</tr>
<tr>
<td>Floating-Point Pipeline Operations</td>
<td>224</td>
</tr>
<tr>
<td>Load/Store Pipeline Operations</td>
<td>225</td>
</tr>
<tr>
<td>Code Sample Analysis</td>
<td>226</td>
</tr>
<tr>
<td>Appendix C</td>
<td>Implementation of Write Combining</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>Introduction</td>
<td>229</td>
</tr>
<tr>
<td>Write-Combining Definitions and Abbreviations</td>
<td>230</td>
</tr>
<tr>
<td>What is Write Combining?</td>
<td>230</td>
</tr>
<tr>
<td>Programming Details</td>
<td>230</td>
</tr>
<tr>
<td>Write-Combining Operations</td>
<td>231</td>
</tr>
<tr>
<td>Sending Write-Buffer Data to the System</td>
<td>233</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Appendix D</th>
<th>Performance-Monitoring Counters</th>
<th>235</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>235</td>
<td></td>
</tr>
<tr>
<td>Performance Counter Usage</td>
<td>236</td>
<td></td>
</tr>
<tr>
<td>PerfEvtSel[3:0] MSRs (MSR Addresses C001_0000h–C001_0003h)</td>
<td>236</td>
<td></td>
</tr>
<tr>
<td>PerfCtr[3:0] MSRs (MSR Addresses C001_0004h–C001_0007h)</td>
<td>240</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Appendix E</th>
<th>Programming the MTRR and PAT</th>
<th>243</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>243</td>
<td></td>
</tr>
<tr>
<td>Memory Type Range Register (MTRR) Mechanism</td>
<td>243</td>
<td></td>
</tr>
<tr>
<td>Page Attribute Table (PAT)</td>
<td>249</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Appendix F</th>
<th>Instruction Dispatch and Execution Resources/Timing</th>
<th>259</th>
</tr>
</thead>
</table>

| Index | 305 |

Table of Contents
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AMD Athlon™ Processor Block Diagram</td>
<td>205</td>
</tr>
<tr>
<td>2</td>
<td>Integer Execution Pipeline</td>
<td>209</td>
</tr>
<tr>
<td>3</td>
<td>Floating-Point Unit Block Diagram</td>
<td>211</td>
</tr>
<tr>
<td>4</td>
<td>Load/Store Unit</td>
<td>212</td>
</tr>
<tr>
<td>5</td>
<td>Fetch/Scan/Align/Decode Pipeline Hardware</td>
<td>216</td>
</tr>
<tr>
<td>6</td>
<td>Fetch/Scan/Align/Decode Pipeline Stages</td>
<td>216</td>
</tr>
<tr>
<td>7</td>
<td>Integer Execution Pipeline</td>
<td>218</td>
</tr>
<tr>
<td>8</td>
<td>Integer Pipeline Stages</td>
<td>218</td>
</tr>
<tr>
<td>9</td>
<td>Floating-Point Unit Block Diagram</td>
<td>220</td>
</tr>
<tr>
<td>10</td>
<td>Floating-Point Pipeline Stages</td>
<td>220</td>
</tr>
<tr>
<td>11</td>
<td>PerfEvtSel[3:0] Registers</td>
<td>237</td>
</tr>
<tr>
<td>12</td>
<td>MTRR Mapping of Physical Memory</td>
<td>245</td>
</tr>
<tr>
<td>13</td>
<td>MTRR Capability Register Format</td>
<td>246</td>
</tr>
<tr>
<td>14</td>
<td>MTRR Default Type Register Format</td>
<td>247</td>
</tr>
<tr>
<td>15</td>
<td>Page Attribute Table (MSR 277h)</td>
<td>249</td>
</tr>
<tr>
<td>16</td>
<td>MTRRphysBasen Register Format</td>
<td>255</td>
</tr>
<tr>
<td>17</td>
<td>MTRRphysMaskn Register Format</td>
<td>256</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>Latency of Repeated String Instructions</td>
<td>123</td>
</tr>
<tr>
<td>Table 2</td>
<td>Integer Pipeline Operation Types</td>
<td>223</td>
</tr>
<tr>
<td>Table 3</td>
<td>Integer Decode Types</td>
<td>223</td>
</tr>
<tr>
<td>Table 4</td>
<td>Floating-Point Pipeline Operation Types</td>
<td>224</td>
</tr>
<tr>
<td>Table 5</td>
<td>Floating-Point Decode Types</td>
<td>224</td>
</tr>
<tr>
<td>Table 6</td>
<td>Load/Store Unit Stages</td>
<td>225</td>
</tr>
<tr>
<td>Table 7</td>
<td>Sample 1—Integer Register Operations</td>
<td>227</td>
</tr>
<tr>
<td>Table 8</td>
<td>Sample 2—Integer Register and Memory Load Operations</td>
<td>228</td>
</tr>
<tr>
<td>Table 9</td>
<td>Write Combining Completion Events</td>
<td>232</td>
</tr>
<tr>
<td>Table 10</td>
<td>AMD Athlon™ System Bus Command Generation Rules</td>
<td>233</td>
</tr>
<tr>
<td>Table 11</td>
<td>Performance-Monitoring Counters</td>
<td>238</td>
</tr>
<tr>
<td>Table 12</td>
<td>Memory Type Encodings</td>
<td>246</td>
</tr>
<tr>
<td>Table 13</td>
<td>Standard MTRR Types and Properties</td>
<td>248</td>
</tr>
<tr>
<td>Table 14</td>
<td>PATi 3-Bit Encodings</td>
<td>250</td>
</tr>
<tr>
<td>Table 15</td>
<td>Effective Memory Type Based on PAT and MTRRs</td>
<td>251</td>
</tr>
<tr>
<td>Table 16</td>
<td>Final Output Memory Types</td>
<td>252</td>
</tr>
<tr>
<td>Table 17</td>
<td>MTRR Fixed Range Register Format</td>
<td>254</td>
</tr>
<tr>
<td>Table 18</td>
<td>MTRR-Related Model-Specific Register (MSR) Map</td>
<td>257</td>
</tr>
<tr>
<td>Table 19</td>
<td>Integer Instructions</td>
<td>261</td>
</tr>
<tr>
<td>Table 20</td>
<td>MMX™ Instructions</td>
<td>287</td>
</tr>
<tr>
<td>Table 21</td>
<td>MMX™ Extensions</td>
<td>291</td>
</tr>
<tr>
<td>Table 22</td>
<td>Floating-Point Instructions</td>
<td>292</td>
</tr>
<tr>
<td>Table 23</td>
<td>3DNow!™ Instructions</td>
<td>298</td>
</tr>
<tr>
<td>Table 24</td>
<td>3DNow!™ Extensions</td>
<td>300</td>
</tr>
<tr>
<td>Table 25</td>
<td>Instructions Introduced with 3DNow!™ Professional</td>
<td>301</td>
</tr>
</tbody>
</table>
# Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Rev</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feb. 2002</td>
<td>K</td>
<td>Corrected the code sequences labeled “by 13” and “by 21” in “Consider Alternative Code When Multiplying by a Constant” on page 120.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removed the outdated references to Appendix G.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removed the blank pages at the end of Chapter 10, “3DNow!™ and MMX™ Optimizations.”</td>
</tr>
<tr>
<td>July 2001</td>
<td>J</td>
<td>Replaced memcpy example for arrays in “AMD Athlon™ Processor-Specific Code” on page 178.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Revised “PerfCtr[3:0] MSRs (MSR Addresses C001_0004h–C001_0007h)” on page 240.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added Table 25, “Instructions Introduced with 3DNow!™ Professional,” on page 301.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updated the wording regarding the L2 cache in “L2 Cache” on page 213.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added block copy/prefetch to “Optimizing Main Memory Performance for Large Arrays” on page 66.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Removed Appendix G.</td>
</tr>
<tr>
<td>Sept. 2000</td>
<td>I</td>
<td>Corrected Example 1 under “Muxing Constructs” on page 98.</td>
</tr>
<tr>
<td>June 2000</td>
<td>H</td>
<td>Added Appendix D, “Performance-Monitoring Counters.”</td>
</tr>
<tr>
<td>April 2000</td>
<td>G</td>
<td>Added more details to the optimizations in Chapter 2, “Top Optimizations.”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Further clarified the information in “Use Array-Style Instead of Pointer-Style Code” on page 20.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added the optimization, “Always Match the Size of Stores and Loads” on page 24.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added the optimization, “Fast Floating-Point-to-Integer Conversion” on page 40.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added the optimization, “Speeding Up Branches Based on Comparisons Between Floats” on page 42.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added the optimization, “Use Read-Modify-Write Instructions Where Appropriate” on page 52.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Further clarified the information in “Align Branch Targets in Program Hot Spots” on page 54.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added the optimization, “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added the optimization, “Use LEAVE Instruction for Function Epilogue Code” on page 56.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added more examples to “Memory Size and Alignment Issues” on page 63.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Further clarified the information in “Use the PREFETCH 3DNow!™ Instruction” on page 79.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Further clarified the information in “Store-to-Load Forwarding Restrictions” on page 86.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Changed epilogue code in Example 1 of “Stack Alignment Considerations” on page 90.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added Example 8 to “Avoid Branches Dependent on Random Data” on page 93.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fixed comments in examples 1 and 2 of “Unsigned Division by Multiplication of Constant” on page 116.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Revised code in “Algorithm: Divisors 1 &lt;= d &lt; 2^{31}, Odd d” page 116 and “Algorithm: Divisors 2 &lt;= d &lt; 2^{31}” on page 118.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added more examples to “Efficient 64-Bit Integer Arithmetic” on page 125.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fixed typo in the integer example and added an MMX™ version in “Efficient Implementation of Population Count Function” on page 136.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added the optimization, “Efficient Binary-to-ASCII Decimal Conversion” on page 139.</td>
</tr>
<tr>
<td>Date</td>
<td>Rev</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-----</td>
<td>-------------</td>
</tr>
<tr>
<td>Nov. 1999</td>
<td>E</td>
<td>Added “About This Document” on page 1. Further clarified the information in “Consider the Sign of Integer Operands” on page 18. Added the optimization, “Use Array-Style Instead of Pointer-Style Code” on page 20. Added the optimization, “Accelerating Floating-Point Divides and Square Roots” on page 38. Clarified the examples in “Copy Frequently Dereferenced Pointer Arguments to Local Variables” on page 44. Further clarified the information in “Select DirectPath Over VectorPath Instructions” on page 50. Further clarified the information in “Align Branch Targets in Program Hot Spots” on page 54. Further clarified the use of the REP instruction as filler in “Code Padding Using Neutral Code Fillers” on page 58. Further clarified the information in “Use the PREFETCH 3DNow!™ Instruction” on page 79. Modified examples 1 and 2 of “Unsigned Division by Multiplication of Constant” on page 116. Added the optimization, “Efficient Implementation of Population Count Function” on page 136. Further clarified the information in “Use FFREEP Macro to Pop One Register from the FPU Stack” on page 152. Further clarified the information in “Minimize Floating-Point-to-Integer Conversions” on page 154.</td>
</tr>
<tr>
<td>Date</td>
<td>Rev</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-----</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added the optimization, “Take Advantage of the FSINCOS Instruction” on page 159.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Further clarified the information in “Use 3DNow!™ Instructions for Fast Division” on page 162.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Further clarified the information in “Use FEMMS Instruction” on page 162.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Further clarified the information in “Use 3DNow!™ Instructions for Fast Square Root and Reciprocal Square Root” on page 165.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clarified “3DNow!™ and MMX™ Intra-Operand Swapping” on page 169.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Corrected PCMPGT information in “Use MMX™ PCMP Instead of 3DNow!™ PFCMP” on page 173.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added the optimization, “Use MMX™ Instructions for Block Copies and Block Fills” on page 174.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Modified the rule for “Use MMX™ PXOR to Clear All Bits in an MMX Register” on page 185.</td>
</tr>
<tr>
<td>Oct. 1999 D</td>
<td></td>
<td>Modified the rule in “Use MMX™ PCMPEQD to Set All Bits in an MMX Register” on page 186.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added the optimization, “Efficient 3D-Clipping Code Computation Using 3DNow!™ Instructions” on page 190.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added the optimization, “Complex Number Arithmetic” on page 199.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added Appendix E, “Programming the MTRR and PAT.”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rearranged the appendixes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added index.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

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

About This Document

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

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

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

<table>
<thead>
<tr>
<th>Section</th>
<th>Topic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1</td>
<td>Introduction</td>
<td>Outlines the material covered in this document. Summarizes the AMD Athlon™ microarchitecture.</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>Top Optimizations</td>
<td>Provides convenient descriptions of the most important optimizations a programmer should take into consideration.</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>C Source-Level Optimizations</td>
<td>Describes optimizations that C/C++ programmers can implement.</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>Instruction Decoding Optimizations</td>
<td>Describes methods that will make the most efficient use of the three sophisticated instruction decoders in the AMD Athlon processor.</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>Cache and Memory Optimizations</td>
<td>Describes optimizations that make efficient use of the large L1 and L2 caches and high-bandwidth buses of the AMD Athlon processor.</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>Branch Optimizations</td>
<td>Describes optimizations that improve branch prediction and minimize branch penalties.</td>
</tr>
<tr>
<td>Chapter 7</td>
<td>Scheduling Optimizations</td>
<td>Describes optimizations that improve code scheduling for efficient execution resource utilization.</td>
</tr>
<tr>
<td>Chapter 8</td>
<td>Integer Optimizations</td>
<td>Describes optimizations that improve integer arithmetic and make efficient use of the integer execution units in the AMD Athlon processor.</td>
</tr>
<tr>
<td>Chapter 9</td>
<td>Floating-Point Optimizations</td>
<td>Describes optimizations that make maximum use of the superscalar and pipelined floating-point unit (FPU) of the AMD Athlon processor.</td>
</tr>
</tbody>
</table>
The AMD Athlon processor family uses state-of-the-art decoupled decode/execution design techniques to deliver next-generation performance with x86 binary software compatibility. This next-generation processor family advances x86 code execution by using flexible instruction predecoding, wide and balanced decoders, aggressive out-of-order execution, parallel integer execution pipelines, parallel floating-point execution pipelines, deep pipelined execution for higher delivered operating frequency, dedicated cache memory, and a new high-performance double-rate 64-bit local bus.

As an x86 binary-compatible processor, the AMD Athlon processor implements the industry-standard x86 instruction set...
by decoding and executing the x86 instructions using a proprietary microarchitecture. This microarchitecture allows the delivery of maximum performance when running x86-based PC software.

**AMD Athlon™ Processor Microarchitecture Summary**

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

- High-speed double-rate local-bus interface
- Large, split 128-Kbyte level-one (L1) cache
- External level-two (L2) cache on Models 1 and 2
- On-die L2 cache on Models 3, 4, and 6
- Dedicated level-two (L2) cache
- Instruction predecode and branch detection during cache-line fills
- Decoupled decode/execution core
- Three-way x86 instruction decoding
- Dynamic scheduling and speculative execution
- Three-way integer execution
- Three-way address generation
- Three-way floating-point execution
- 3DNow!™ technology and MMX™ single-instruction multiple-data (SIMD) instruction extensions
- Super data forwarding
- Deep out-of-order integer and floating-point execution
- Register renaming
- Dynamic branch prediction

The AMD Athlon processor communicates through a next-generation high-speed local bus that is beyond the current Socket 7 or Super7™ bus standard. The local bus can transfer data at twice the rate of the bus operating frequency by using both the rising and falling edges of the clock (see
“AMD Athlon™ System Bus” on page 214 for more information).

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

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

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

The 3DNow! execution units allow for high-performance floating-point vector operations, which can replace x87 instructions and enhance the performance of 3D graphics and other floating-point-intensive applications. Because the 3DNow! architecture uses the same registers as the MMX instructions, switching between MMX and 3DNow! has no penalty.

The AMD Athlon processor designers took another innovative step by carefully integrating the traditional x87 floating-point, MMX, and 3DNow! execution units into one operational engine. With the introduction of the AMD Athlon processor, the switching overhead between x87, MMX, and 3DNow! technology is virtually eliminated. The AMD Athlon processor combined with 3DNow! technology brings a better multimedia experience to mainstream PC users while maintaining backward compatibility with all existing x86 software.
Although the AMD Athlon processor can extract code parallelism on-the-fly from off-the-shelf, commercially available x86 software, specific code optimization for the AMD Athlon processor can result in even higher delivered performance. This document describes the proprietary microarchitecture in the AMD Athlon processor and makes recommendations for optimizing execution of x86 software on the processor.

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

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

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

Group I—Essential Optimizations

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

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

Group II—Secondary Optimizations

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

- Load-Execute Instruction Usage—Use Load-Execute instructions—Avoid load-execute floating-point instructions with integer operands
- Take Advantage of Write Combining
- Optimization of Array Operations With Block Prefetching
- Use 3DNow! Instructions
- Recognize 3DNow! Professional Instructions
- Avoid Branches Dependent on Random Data
- Avoid Placing Code and Data in the Same 64-Byte Cache Line
Optimization Star

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

Group I Optimizations—Essential Optimizations

Memory-Size and Alignment Issues

Avoid Memory-Size Mismatches

Avoid memory-size mismatches when different instructions operate on the same data. When an instruction stores and another instruction reloads the same data, keep their operands aligned and keep the loads/stores of each operand the same size. The following code examples result in a store-to-load-forwarding (STLF) stall:

Example 1 (Avoid):

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

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

Example 2 (Avoid):

```
FST   QWORD PTR [FOO]
MOV   EAX, DWORD PTR [FOO]
MOV   EDX, DWORD PTR [FOO+4]
```
Align Data Where Possible

Avoid misaligned data references. All data whose size is a power of two is considered aligned if it is naturally aligned. For example:

- Word accesses are aligned if they access an address divisible by two.
- Doubleword accesses are aligned if they access an address divisible by four.
- Quadword accesses are aligned if they access an address divisible by eight.
- TBYTE accesses are aligned if they access an address divisible by eight.

A misaligned store or load operation suffers a minimum one-cycle penalty in the AMD Athlon processor load/store pipeline. In addition, using misaligned loads and stores increases the likelihood of encountering a store-to-load forwarding pitfall. For a more detailed discussion of store-to-load forwarding issues, see “Store-to-Load Forwarding Restrictions” on page 86.

Use the 3DNow!™ Prefetching Instructions

For code that can take advantage of prefetching, use the 3DNow! PREFETCH and PREFETCHW instructions to increase the effective bandwidth of the AMD Athlon processor, thereby significantly improving performance. All the prefetch instructions are essentially integer instructions and can be used anywhere, in any type of code (for example, integer, x87, 3DNow!, MMX). Use the following formula to determine prefetch distance:

\[
\text{Prefetch Distance} = 200 \times \left( \frac{\text{DS}}{\text{C}} \right)
\]

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

See “Use the PREFETCH 3DNow!™ Instruction” on page 79 for more details.
Select DirectPath Over VectorPath Instructions

Use DirectPath instructions rather than VectorPath instructions. DirectPath instructions are optimized for decode and execute efficiently by minimizing the number of operations per x86 instruction, which includes ‘register←register op memory’ as well as ‘register←register op register’ forms of instructions. Up to three DirectPath instructions can be decoded per cycle. VectorPath instructions block the decoding of DirectPath instructions.

The AMD Athlon processor implements the majority of instructions used by a compiler as DirectPath instructions. Nevertheless, assembly writers must still take into consideration the usage of DirectPath versus VectorPath instructions.

See Appendix F, “Instruction Dispatch and Execution Resources/Timing,” for tables of DirectPath and VectorPath instructions.

Group II Optimizations—Secondary Optimizations

Load-Execute Instruction Usage

Use Load-Execute Instructions

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

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

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

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

**Example 1 (Avoid):**

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

**Example 1 (Preferred):**

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

Avoid Load-Execute Floating-Point Instructions with Integer Operands

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

Use separate FILD and arithmetic instructions for floating-point computations involving integer-memory operands. This optimization has the potential to increase decode bandwidth and OP density in the FPU scheduler. The floating-point load-execute instructions with integer operands are VectorPath and generate two OPs in a cycle, while the discrete equivalent enables a third DirectPath instruction to be decoded in the same cycle. In some situations, this optimization can also reduce execution time if the FILD can be scheduled several instructions ahead of the arithmetic instruction in order to cover the FILD latency.
Example 2 (Avoid):

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

Example 2 (Preferred):

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

Take Advantage of Write Combining

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


Optimizing Main Memory Performance for Large Arrays

Reading Large Arrays and Streams

To process a large array (200 Kbytes or more), or other large sequential data sets that are not already in cache, use block prefetch to achieve maximum performance. The block prefetch technique involves processing the data in blocks. The data for each block is preloaded into the cache by reading just one address per cache line, causing each cache line to be filled with the data from main memory.

Filling the cache lines in this manner, with a single read operation per line, allows the memory system to burst the data at the highest achievable read bandwidth.

Once the input data is in cache, the processing can then proceed at the maximum instruction execution rate, because no memory read accesses will slow down the processor.
Writing Large Arrays to Memory

If data needs to be written back to memory during processing, a similar technique can be used to accelerate the write phase. The processing loop writes data to a temporary in-cache buffer, to avoid memory-access cycles and to allow the processor to execute at the maximum instruction rate. Once a complete data block has been processed, the results are copied from the in-cache buffer to main memory, using a loop that employs the very fast streaming store instruction, MOVNTQ.

See “Optimizing Main Memory Performance for Large Arrays” on page 66 for detailed optimization examples, where the block-prefetch method is used for simply copying memory, and also for adding two floating-point arrays through the use of the x87 floating-point unit.

Also see the complete optimized memcpy routine in “Use MMX™ Instructions for Block Copies and Block Fills” on page 174. This example employs Block Prefetch for large size memory blocks.

Use 3DNow!™ Instructions

When single precision is required, perform floating-point computations using the 3DNow! instructions instead of x87 instructions. The SIMD nature of 3DNow! instructions achieves twice the number of FLOPs that are achieved through x87 instructions. 3DNow! instructions also provide for a flat register file instead of the stack-based approach of x87 instructions.

See Table 23 on page 298 for a list of 3DNow! instructions. For information about instruction usage, see the 3DNow!™ Technology Manual, order no. 21928.
Recognize 3DNow! Professional Instructions

AMD Athlon™ processors that include 3DNow! Professional instructions indicate the presence of Streaming SIMD Extensions (SSE) through the standard CPUID feature bit 25. See Table 25 on page 301 for a list of the additional SSE instructions introduced with 3DNow! Professional technology.

Where SSE optimizations already exist, or are planned for future development, feature-detection code using CPUID should be checked to ensure correct CPU vendor independent recognition of SSE on AMD processors. For a full description of CPU feature detection on AMD processors, please refer to the AMD Processor Recognition Application Note, order no. 20734.

Avoid Branches Dependent on Random Data

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

If possible, design branch-free alternative code sequences, which result in shorter average execution time. This technique is especially important if the branch body is small. See “Avoid Branches Dependent on Random Data” on page 93 for more details.
Avoid Placing Code and Data in the Same 64-Byte Cache Line

Sharing code and data in the same 64-byte cache line may cause the L1 caches to thrash (unnecessary castout of code/data) in order to maintain coherency between the separate instruction and data caches. The AMD Athlon processor has a cache-line size of 64 bytes, which is twice the size of previous processors. Avoid placing code and data together within this larger cache line, especially if the data becomes modified.

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 would result in lower performance.

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

In general, avoid the following:
- Self-modifying code
- Storing data in code segments
C Source-Level Optimizations

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

Ensure Floating-Point Variables and Expressions are of Type Float

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

Use 32-Bit Data Types for Integer Code

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

In many cases, the data stored in integer variables 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.

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

Integer-to-floating-point conversion using integers larger than 16 bits is faster with signed types, as the x86 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:

**Example 1 (Avoid):**

```assembly
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 previous 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:

**Example 1 (Preferred):**

```assembly
double x;         ===> FILD DWORD PTR [i]
int i; FSTP QWORD PTR [x]
```

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

**Example 2 (Avoid):**

```c
int i; ===> MOV EAX, i
CDQ
i = i / 4;
AND EDX, 3
ADD EAX, EDX
SAR EAX, 2
MOV i, EAX
```

**Example 2 (Preferred):**

```c
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-float conversion
Use Array-Style Instead of Pointer-Style Code

The use of pointers in C makes work difficult for the optimizers in C compilers. Without detailed and aggressive pointer analysis, the compiler has to assume that writes through a pointer can write to any place in memory. This includes storage allocated to other variables, creating the issue of aliasing, i.e., the same block of memory is accessible in more than one way.

To help the C compiler optimizer in its analysis, avoid the use of pointers where possible. One example where this is trivially possible is in the access of data organized as arrays. C allows the use of either the array operator [] or pointers to access the array. Using array-style code makes the task of the optimizer easier by reducing possible aliasing.

For example, x[0] and x[2] cannot possibly refer to the same memory location, while *p and *q could. It is highly recommended to use the array style, as significant performance advantages can be achieved with most compilers.

**Example 1 (Avoid):**

```c
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 */
    }
}
```

```c
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++;
        *res++ = dp;  /* write transformed x */
    }
}
```
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 */

}

Example 1 (Preferred):

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];
    }
}
Reality Check

Note that source code transformations interact with a compiler’s code generator and that it is difficult to control the generated machine code from the source level. It is even possible that source code transformations for improving performance and compiler optimizations “fight” each other. Depending on the compiler and the specific source code, it is therefore possible that pointer style code will be compiled into machine code that is faster than that generated from equivalent array style code. It is advisable to check the performance after any source code transformation to see whether performance really has improved.

Completely Unroll Small Loops

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

Example 1 (Avoid):

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

Example 1 (Preferred):

```c
// 3D-transform: multiply vector V by 4x4 transform matrix M
```
Avoid Unnecessary Store-to-Load Dependencies

A store-to-load dependency exists when data is stored to memory, only to be read back shortly thereafter. See “Store-to-Load Forwarding Restrictions” on page 86 for more details. The AMD Athlon processor contains hardware to accelerate such store-to-load dependencies, allowing the load to obtain the store data before it has been written to memory. However, it is still faster to avoid such dependencies altogether and keep the data in an internal register.

Avoiding store-to-load dependencies is especially important if they are part of a long dependency chains, as 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. It is therefore recommended that the programmer remove the dependency manually, e.g., by introducing a temporary variable that can be kept in a register. This can result in a significant performance increase. The following is an example of this.

**Example 1 (Avoid):**

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

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

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

**Example 1 (Preferred):**

```c
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;
}

Always Match the Size of Stores and Loads

The AMD Athlon processor contains a load/store buffer (LS) to speed up the forwarding of store data to dependent loads. However, this store-to-load forwarding (STLF) inside the LS occurs in general only when the addresses and sizes of the store and the dependent load match, and when both memory accesses are aligned (see section “Store-to-Load Forwarding Restrictions” on page 86 for details).

It is impossible to control load and store activity at the source level 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 sized 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 differently sized 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. Example 1 (Avoid) 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 STLF in LS takes place. Example 1 (Preferred) shows how to properly implement fixed_int() in order to allow store-to-load-forwarding in LS.

Example 1 (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 following two variants:

;variant 1
MOV   EDX, DWORD PTR [z]
MOV   EAX, DWORD PTR [y]   ;+-
ADD   EAX, EDX   ; |
MOV   DWORD PTR [y], EAX   ; |+ misaligned/address mismatch, no forwarding in LS
MOV   EAX, DWORD PTR [y+2];<+ size and address mismatch,
AND   EAX, OFFFFh
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,   
MOV   DWORD PTR [q], EAX

Example 1 (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 as follows:

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.
SHR EAX, 16
MOV DWORD PTR [q], EAX
Consider Expression Order in Compound Branch Conditions

Branch conditions in C programs are often compound conditions consisting of multiple boolean expressions joined by the boolean operators && and ||. C guarantees a short-circuit evaluation of these operators. This means that in the case of ||, the first operand to evaluate to TRUE terminates the evaluation, i.e., following operands are not evaluated at all. Similarly for &&, 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 || and &&. This is especially the case when the evaluation of one of the operands causes a side effect. However, in most cases the exchange of operands is possible.

When used to control conditional branches, expressions involving || 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 mispredict for each of the branches generated
- Additional latency incurred due to a branch mispredict
- 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 mispredict probabilities on the nature of the incoming data in data dependent branches)

It is therefore 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. Feed a “typical” data stream to the program while doing the experiments.
Switch Statement Usage

Optimize Switch Statements

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

Example 1 (Avoid):

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

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

Example 1 (Preferred):

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

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

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

Use Const Type Qualifier

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

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

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

**Example 1:**

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

Transform the above loop into:

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

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

**Generalization for Multiple Constant Control Code**

To generalize this further for multiple constant control code, some more work may have to be done to create the proper outer loop. Enumeration of the constant cases reduces this to a simple switch statement.
Example 2:

```c
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 above loop by using the switch statement into:

```c
#define combine( c1, c2 ) (((c1) << 1) + (c2))
switch( combine( CONSTANT0!=0, CONSTANT1!=0 ) ) {
    case combine( 0, 0 ):
        for( i ... ) {
            DoWork0( i );
            DoWork2( i );
        }
        break;
    case combine( 1, 0 ):
        for( i ... ) {
            DoWork1( i );
            DoWork2( i );
        }
        break;
    case combine( 0, 1 ):
        for( i ... ) {
            DoWork0( i );
            DoWork3( i );
        }
        break;
    case combine( 1, 1 ):
        for( i ... ) {
            DoWork1( i );
            DoWork3( i );
        }
        break;
    default:
        break;
}
```
The trick here is that there is some up-front work involved in generating all the combinations for the switch constant and the total amount of code has doubled. However, it is also clear that the inner loops are “if()-free”. In ideal cases where the “DoWork*()” functions are inlined, the successive functions will have greater overlap leading to greater parallelism than would be possible in the presence of intervening if() statements.

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

However, the number of inner loops can also substantially increase. If the number of inner loops is prohibitively high, then only the most common cases need to be dealt with directly, and the remaining cases can fall back to the old code in a “default:” clause for the switch() statement.

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

**Declare Local Functions as Static**

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

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

**Example 1:**
```c
double* p;
double* np;

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

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

Introduce Explicit Parallelism into Code

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

Note that reordered code that is algebraically identical to the original code does not necessarily deliver identical computational results due to the lack of associativity of floating point operations. There are well-known numerical considerations in applying these optimizations (consult a book on numerical analysis). In some cases, these optimizations may lead to unexpected results. Fortunately, in the vast majority of cases, the final result differs only in the least significant bits.
**Example 1 (Avoid):**

double a[100], sum;
int i;

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

**Example 1 (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 was 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 maximal sustained utilization.
Explicitly Extract Common Subexpressions

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

Example 1 (Avoid):
```c
double a, b, c, d, e, f;

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

Example 1 (Preferred):
```c
double a, b, c, d, e, f, t;

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

Example 2 (Avoid):
```c
double a, b, c, e, f;

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

Example 2 (Preferred):
```c
double a, b, c, e, f, t;

t = 1 / c;
e = a * t;
f = b * t;
```
C Language Structure Component Considerations

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

**Sort by Base Type Size**

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

**Pad by Multiple of Largest Base Type Size**

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

The following example demonstrates the reordering of structure member declarations:

**Example 1, Original ordering (Avoid):**

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

**Example 1, New ordering with padding (Preferred):**

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

See “C Language Structure Component Considerations” on page 91 for a different perspective.
Sort Local Variables According to Base Type Size

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

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

The following example demonstrates the reordering of local variable declarations:

Example 1, Original ordering (Avoid):

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

Example 1, Improved ordering (Preferred):

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

See “Sort Variables According to Base Type Size” on page 92 for more information from a different perspective.
Divides and square roots have a much longer latency than other floating-point operations, even though the AMD Athlon processor provides significant acceleration of these two operations. In some codes, these operations occur so often as to seriously impact performance. In these cases, it is recommended to port the code to 3DNow! inline assembly or to use a compiler that can generate 3DNow! code. If code has hot spots that use single-precision arithmetic only (i.e., 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 what precision results get rounded to. It affects the basic arithmetic operations, including divides and square roots. AMD Athlon and AMD-K6® family processors implement divide and square root in such fashion as to only compute the number of bits necessary for the currently selected precision. This means that setting precision control to single precision (versus 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.

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.
Example 1:

/* 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, 0xffffffff);
Fast Floating-Point-to-Integer Conversion

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, the 3DNow! PF2ID instruction, which performs truncating conversion, can be utilized by the compiler to accomplish rapid floating-point to integer conversion.

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 floating-point source operand and store out integer result.
4. Restore original x87 rounding mode.

This algorithm is typically implemented through a C runtime library function called `ftol()`. While the AMD Athlon processor has 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 the “Fast” example below may be helpful. Note that this code uses the current rounding mode instead of truncation when performing the conversion. Therefore the result may differ by one 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.

**Note:** This 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 depending load specifically supported by the store-to-load-forwarding hardware of the AMD Athlon processor.

**Example 1 (Slow):**

double x;
int i;

i = x;

**Example 1 (Fast):**

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

double x;
int i;

DOUBLE2INT(i,x);
```
Branches based on floating-point comparisons are often slow. The AMD Athlon processor supports 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.

With many compilers, the only way they implement branches based on floating-point comparisons is to use the FCOM or FCOMP instructions to compare the floating-point operands, followed by “FSTSW AX” in order to transfer the x87 condition code flags into EAX. This allows a branch based on the contents of that register. Although the AMD Athlon processor has acceleration hardware to speed up the FSTSW instruction, this process is still fairly slow.

One alternative for branches based on comparisons between 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 to 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 IEEE-754. 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:
#define FLOAT2INTCAST(f) (*((int *)(&f)))
#define FLOAT2UINTCAST(f) (*((unsigned int *)(&f)))

// comparisons against zero
if (f < 0.0f)  ==>  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)

// comparisons against positive constant
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) ==>  if (FLOAT2INTCAST(f) >= 0x40400000)

// comparisons among two floats
if (f1 < f2)   ==>  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;
    if (FLOAT2UINTCAST(f) <= 0x80000000U)
Avoid Unnecessary Integer Division

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

Example 1 (Avoid):
```
int i,j,k,m;

m = i / j / k;
```

Example 1 (Preferred):
```
int i,j,k,l;

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

Copy Frequently Dereferenced Pointer Arguments to Local Variables

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

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

Otherwise, copy the data pointed to by the pointer arguments to local variables at the start of the function and if necessary copy them back at the end of the function.
Example 1 (Avoid):

```c
// 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;
}
```

Example 1 (Preferred):

```c
// 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;
}
Use Block Prefetch Optimizations

Block prefetching can be applied to C code without using any assembly level instructions.

This example adds all the values in two arrays of double precision floating point values, to produce a single double precision floating point total. The optimization technique can be applied to any code that processes large arrays from system memory.

This is an ordinary C++ loop that does the job. Bandwidth is approximated for code execution on an AMD Athlon™ 4 DDR:

**Example: Standard C code**

```
(bandwidth: ~750 MB/sec )
```

```
for (int i = 0; i < MEM_SIZE; i += 8) { // 8 bytes per double
    double summo += *a_ptr++ + *b_ptr++; // reads from memory
}
```

Using a block prefetch can significantly improve memory read bandwidth. The same function optimized using block prefetch to read the arrays into cache at maximum bandwidth follows. The block prefetch is implemented in C++ source code, as procedure BLOCK_PREFETCH_4K. It reads 4 Kbytes of data per block. This version gets about 1125 Mbytes/sec on AMD Athlon™ 4 processor DDR, a 50% performance gain over the Standard C Code Example.
Example: C code Using Block Prefetching

(bandwidth: ~1125 Mbytes/sec)

static const int CACHEBLOCK = 0x1000;  // prefetch chunk size (4K bytes)
int p_fetch;  // this "anchor" variable helps us fool the C optimizer

static const void inline BLOCK_PREFETCH_4K (void* addr) {
    int* a = (int*) addr;  // cast as INT pointer for speed

    + a[64] + a[80] + a[96] + a[112]  // 64th address,
    + a[128] + a[144] + a[160] + a[176]  // to hit each

    a += 256;  // point to second 1K stretch of addresses

    p_fetch += a[0] + a[16] + a[32] + a[48]
    + a[64] + a[80] + a[96] + a[112]
    + a[128] + a[144] + a[160] + a[176]
    + a[192] + a[208] + a[224] + a[240];

    a += 256;  // point to third 1K stretch of addresses

    p_fetch += a[0] + a[16] + a[32] + a[48]
    + a[64] + a[80] + a[96] + a[112]
    + a[128] + a[144] + a[160] + a[176]
    + a[192] + a[208] + a[224] + a[240];

    a += 256;  // point to fourth 1K stretch of addresses

    p_fetch += a[0] + a[16] + a[32] + a[48]
    + a[64] + a[80] + a[96] + a[112]
    + a[128] + a[144] + a[160] + a[176]
    + a[192] + a[208] + a[224] + a[240];
}

for (int m = 0; m < MEM_SIZE; m += CACHEBLOCK) {  // process in blocks
    BLOCK_PREFETCH_4K(a_ptr);  // get next 4K bytes of "a" into cache
    BLOCK_PREFETCH_4K(b_ptr);  // get next 4K bytes of "b" into cache

    for (int i = 0; i < CACHEBLOCK; i += 8) {
        double summo += *a_ptr++ + *b_ptr++;  // reads from cache!
    }
}
Caution: Since the prefetch code does not really do anything, from the compiler point of view, there is a danger that it might be optimized out from the code that is generated. So the block prefetch function BLOCK_PREFETCH_4K uses a trick to prevent that from happening. The memory values are read as INTs, added together (which is very fast for INTs), and then assigned to the global variable p_fetch. This assignment should “fool” the optimizer into leaving the prefetch code intact. However, be aware that in general, the compiler might remove block prefetch code.

For a more thorough discussion of block prefetch, see “Optimizing Main Memory Performance for Large Arrays” on page 66, and the optimized memory-copy code in the section “Use MMX™ Instructions for Block Copies and Block Fills” on page 174.
Instruction Decoding Optimizations

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

Overview

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

All instructions (x86, x87, 3DNow!™, and MMX™ instructions) are classified into two types of decodes—DirectPath and VectorPath (see “DirectPath Decoder” and “VectorPath Decoder” under “Early Decoding” on page 207 for more information). DirectPath instructions are common instructions that are decoded directly in hardware. VectorPath instructions are more complex instructions that require the use of a sequence of multiple operations issued from an on-chip ROM.

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

Select DirectPath Over VectorPath Instructions

Use DirectPath instructions rather than VectorPath instructions. DirectPath instructions are optimized for decode and execute efficiently by minimizing the number of operations per x86 instruction, which includes ‘register←register op memory’ as well as ‘register←register op register’ forms of instructions. Up to three DirectPath instructions can be decoded per cycle. VectorPath instructions block the decoding of DirectPath instructions.

The AMD Athlon processor implements the majority of instructions used by a compiler as DirectPath instructions. However, assembly writers must still take into consideration the usage of DirectPath versus VectorPath instructions.

See Appendix F, “Instruction Dispatch and Execution Resources/Timing,” for tables of DirectPath and VectorPath instructions.

Load-Execute Instruction Usage

Use Load-Execute Integer Instructions

Most load-execute integer instructions are DirectPath decodable and can be decoded at the rate of three per cycle.

Splitting a load-execute integer instruction into two separate instructions—a load instruction and a “reg, reg” instruction—reduces decoding bandwidth and increases register pressure, which results in lower performance. Use the split-instruction form to avoid scheduler stalls for longer executing instructions and to explicitly schedule the load and execute operations.
Use Load-Execute Floating-Point Instructions with Floating-Point Operands

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

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

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

**Example 1 (Avoid):**

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

**Example 1 (Preferred):**

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

Avoid Load-Execute Floating-Point Instructions with Integer Operands

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

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

Example 2 (Avoid):

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

Example 2 (Preferred):

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

Use Read-Modify-Write Instructions Where Appropriate

The AMD Athlon processor handles read-modify-write (RMW) instructions such as “ADD [mem], reg32” very efficiently. The vast majority of RMW instructions are DirectPath instructions. Use of RMW instructions can provide a performance benefit over the use of an equivalent combination of load, load-execute and store instructions. In comparison to the load/load-execute/store combination, the equivalent RMW instruction promotes code density (better I-cache utilization), preserves decode bandwidth, and saves execution resources as it occupies only one reservation station and requires only one address computation. It may also reduce register pressure, as demonstrated in Example 2 on page 53.

Use of RMW instructions is indicated if an operation is performed on data that is in memory, and the result of that operation is not reused soon. Due to the limited number of integer registers in an x86 processor, it is often the case that data needs to be kept in memory instead of in registers. Additionally, it can be the case that the data, once operated upon, is not reused soon. An example would be an accumulator inside a loop of unknown trip count, where the accumulator result is not reused inside the loop. Note that for loops with a known trip count, the accumulator manipulation can frequently be hoisted out of the loop.
Example 1 (C code):
/* C code */

int accu, increment;

while (condition) {
    ...
    /* accu is not read and increment is not written here */
    ...
    accu += increment;
}

Example 1 (Avoid):
MOV    EAX, [increment]
ADD    EAX, [accu]
MOV    [accu], EAX

Example 1 (Preferred):
MOV    EAX, [increment]
ADD    [accu], EAX

Example 2 (C code):
/* C code */

int iterationcount;

iteration_count = 0;
while (condition) {
    ...
    /* iteration count is not read here */
    ...
    iteration_count++;
}

Example 2 (Avoid):
MOV    EAX, [iteration_count]
INC    EAX
MOV    [iteration_count], EAX

Example 2 (Preferred):
INC    [iteration_count]
Align Branch Targets in Program Hot Spots

In program hot spots (as determined by either profiling or loop nesting analysis), place branch targets at or near the beginning of 16-byte aligned code windows. This guideline improves performance inside hotspots by maximizing the number of instruction fills into the instruction-byte queue and preserves I-cache space in branch-intensive code outside such hotspots.

Use 32-Bit LEA Rather than 16-Bit LEA Instruction

The 32-bit Load Effective Address (LEA) instruction is implemented as a DirectPath operation with an execute 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.

Use Short Instruction Encodings

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

Example 1 (Avoid):

```
81 C0 78 56 34 12   ADD EAX, 12345678h ;uses 2-byte opcode
; form (with ModR/M)
81 C3 FB FF FF FF   ADD EBX, -5 ;uses 32-bit
; immediate
0F 84 05 00 00 00   JZ $label1 ;uses 2-byte opcode,
; 32-bit immediate
```
Avoid Partial-Register Reads and Writes

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

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

Example 1 (Preferred):

```
05 78 56 34 12 ADD EAX, 12345678h ;uses single byte opcode form
83 C3 FB ADD EBX, -5 ;uses 8-bit sign extended immediate
74 05 JZ $label1 ;uses 1-byte opcode, 8-bit immediate
```

Example 1 (Avoid):

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

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

Example 2 (Avoid):

```
MOV BX, 12h ;inst 1
MOV BL, DL ;inst 2, false dependency on completion of inst 1
MOV BH, CL ;inst 3, false dependency on completion of inst 2
MOV AL, BL ;inst 4, depends on completion of inst 2
```
Use LEAVE Instruction for Function Epilogue Code

A classical approach for referencing function arguments and local variables inside a function is the use of a so-called frame pointer. In x86 code, the EBP register is customarily used as a frame pointer. In function prologue code, the frame pointer is set up as follows:

```
PUSH EBP        ;save old frame pointer
MOV EBP, ESP    ;new frame pointer
SUB ESP, nnnnnnn ;allocate local variables
```

Function arguments on the stack can now be accessed at positive offsets relative to EBP, and local variables are accessible at negative offsets relative to EBP. In the function epilogue code, the following work is performed:

```
MOV ESP, EBP    ;deallocate local variables
POP EBP         ;restore old frame pointer
```

The functionality of these two instructions is identical to that of the LEAVE instruction. The LEAVE instruction is a single-byte instruction and thus saves two bytes of code space over the MOV/POP epilogue sequence. Replacing the MOV/POP sequence with LEAVE also preserves decode bandwidth.

Therefore, use the LEAVE instruction in function epilogue code for both specific AMD Athlon processor optimized and blended code (code that performs well on both AMD-K6® and AMD Athlon processors).

For functions that do not allocate local variables, the prologue and epilogue code can be simplified to the following:

```
PUSH EBP        ;save old frame pointer
MOV EBP, ESP    ;new frame pointer
[...]
POP EBP         ;restore old frame pointer
```

This is optimal in cases where the use of a frame pointer is desired. For highest performance code, do not use a frame pointer at all. Function arguments and local variables should be accessed directly through ESP, thus freeing up EBP for use as a general purpose register and reducing register pressure.
Replace Certain SHLD Instructions with Alternative Code

Certain instances of the SHLD instruction can be replaced by alternative code sequences using ADD and ADC or SHR and LEA. The alternative code has lower latency and requires less execution resources. ADD, ADC, SHR and LEA (32-bit version) are DirectPath instructions, while SHLD is a VectorPath instruction. Use of the replacement code optimizes decode bandwidth as it potentially enables the decoding of a third DirectPath instruction. The replacement code may increase register pressure since it destroys the contents of REG2, whereas REG2 is preserved by SHLD. In situations where register pressure is high, use of the replacement sequences may therefore not be indicated.

**Example 1 (Avoid):**

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

**Example 1 (Preferred):**

```
ADD   REG2, REG2
ADC   REG1, REG1
```

**Example 2 (Avoid):**

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

**Example 2 (Preferred):**

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

**Example 3 (Avoid):**

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

**Example 3 (Preferred):**

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

Use 8-Bit Sign-Extended Immediates

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

Chapter 4  Instruction Decoding Optimizations  57
Use 8-Bit Sign-Extended Displacements

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

Code Padding Using Neutral Code Fillers

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

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

Not all of these instructions are equally suitable for purposes of code padding. For example, SHLD/SHRD are microcoded, which reduces decode bandwidth and takes up execution resources.
Recommendations for AMD-K6® Family and AMD Athlon™ Processor

Blended Code

The instructions and instruction sequences presented below are recommended for code padding on both AMD-K6 family processors and the AMD Athlon processor.

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

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

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

```
NOP2_EAX TEXTEQU <DB 08Bh,0C0h> ; MOV EAX, EAX
NOP2_EBX TEXTEQU <DB 08Bh,0D8h> ; MOV EBX, EBX
NOP2xECX TEXTEQU <DB 08Bh,0C9h> ; MOV ECX, ECX
NOP2_EDX TEXTEQU <DB 08Bh,0D2h> ; MOV EDX, EDX
NOP2_ESI TEXTEQU <DB 08Bh,0F6h> ; MOV ESI, ESI
NOP2_EDI TEXTEQU <DB 08Bh,0FFh> ; MOV EDI, EDI
NOP2_ESP TEXTEQU <DB 08Bh,0E4h> ; MOV ESP, ESP
NOP2_EBP TEXTEQU <DB 08Bh,0EDh> ; MOV EBP, EBP

NOP3_EAX TEXTEQU <DB 08Dh,004h,020h> ; LEA EAX, [EAX]
NOP3_EBX TEXTEQU <DB 08Dh,01Ch,023h> ; LEA EBX, [EBX]
NOP3xECX TEXTEQU <DB 08Dh,00Ch,021h> ; LEA ECX, [ECX]
NOP3_EDX TEXTEQU <DB 08Dh,014h,022h> ; LEA EDX, [EDX]
NOP3_ESI TEXTEQU <DB 08Dh,024h,024h> ; LEA ESI, [ESI]
NOP3_EDI TEXTEQU <DB 08Dh,034h,026h> ; LEA EDI, [EDI]
NOP3_ESP TEXTEQU <DB 08Dh,03Ch,027h> ; LEA ESP, [ESP]
NOP3_EBP TEXTEQU <DB 08Dh,06Dh,000h> ; LEA EBP, [EBP]

NOP4_EAX TEXTEQU <DB 08Dh,044h,020h,000h> ; LEA EAX, [EAX+00]
```
NOP4_EBX TEXTEQU <DB 08Dh,05Ch,023h,000h> ; LEA EBX, [EBX+00]
NOP4(ECX TEXTEQU <DB 08Dh,04Ch,021h,000h> ; LEA ECX, [ECX+00]
NOP4_EDX TEXTEQU <DB 08Dh,054h,022h,000h> ; LEA EDX, [EDX+00]
NOP4_ESI TEXTEQU <DB 08Dh,064h,024h,000h> ; LEA ESI, [ESI+00]
NOP4_EDI TEXTEQU <DB 08Dh,074h,026h,000h> ; LEA EDI, [EDI+00]
NOP4_ESP TEXTEQU <DB 08Dh,07Ch,027h,000h> ; LEA ESP, [ESP+00]

; LEA EAX, [EAX+00]; NOP
NOP5_EAX TEXTEQU <DB 08Dh,044h,020h,000h,090h>

; LEA EBX, [EBX+00]; NOP
NOP5_EBX TEXTEQU <DB 08Dh,05Ch,023h,000h,090h>

; LEA ECX, [ECX+00]; NOP
NOP5_ECX TEXTEQU <DB 08Dh,04Ch,021h,000h,090h>

; LEA EDX, [EDX+00]; NOP
NOP5_EDX TEXTEQU <DB 08Dh,054h,022h,000h,090h>

; LEA ESI, [ESI+00]; NOP
NOP5_ESI TEXTEQU <DB 08Dh,064h,024h,000h,090h>

; LEA EDI, [EDI+00]; NOP
NOP5_EDI TEXTEQU <DB 08Dh,074h,026h,000h,090h>

; LEA ESP, [ESP+00]; NOP
NOP5_ESP TEXTEQU <DB 08Dh,07Ch,027h,000h,090h>

; LEA EAX, [EAX+00000000]
NOP6_EAX TEXTEQU <DB 08Dh,080h,0.0,0.0,0>

; LEA EBX, [EBX+00000000]
NOP6_EBX TEXTEQU <DB 08Dh,09Bh,0.0,0.0,0>

; LEA ECX, [ECX+00000000]
NOP6_ECX TEXTEQU <DB 08Dh,089h,0.0,0.0,0>

; LEA EDX, [EDX+00000000]
NOP6_EDX TEXTEQU <DB 08Dh,092h,0.0,0.0,0>

; LEA ESI, [ESI+00000000]
NOP6_ESI TEXTEQU <DB 08Dh,0B6h,0.0,0.0,0>

; LEA EDI, [EDI+00000000]
NOP6_EDI TEXTEQU <DB 08Dh,0BFh,0.0,0.0,0>

; LEA EBP, [EBP+00000000]
NOP6_EBP TEXTEQU <DB 08Dh,0ADh,0.0,0.0,0>

; LEA EAX, [EAX*1+00000000]
NOP7_EAX TEXTEQU <DB 08Dh,004h,005h,0.0,0.0>
;LEA EBX, [EBX*1+00000000]
LEA ECX, [ECX*1+00000000]
LEA EDX, [EDX*1+00000000]
LEA ESI, [ESI*1+00000000]
LEA EDI, [EDI*1+00000000]
LEA EBP, [EBP*1+00000000]
LEA EAX, [EAX*1+00000000]
JMP

NOP7_EBX TEXTEQU <DB 08Dh,01Ch,01Dh,0,0,0,0>
NOP7_ECX TEXTEQU <DB 08Dh,00Ch,00Dh,0,0,0,0>
NOP7_EDX TEXTEQU <DB 08Dh,014h,015h,0,0,0,0>
NOP7_ESI TEXTEQU <DB 08Dh,034h,035h,0,0,0,0>
NOP7 EDI TEXTEQU <DB 08Dh,03Ch,03Dh,0,0,0,0>
NOP7_EBP TEXTEQU <DB 08Dh,02Ch,02Dh,0,0,0,0>
NOP8_EAX TEXTEQU <DB 08Dh,004h,005h,0,0,0,0,90h>
NOP8_EBX TEXTEQU <DB 08Dh,01Ch,01Dh,0,0,0,0,90h>
NOP8 ECX TEXTEQU <DB 08Dh,00Ch,00Dh,0,0,0,0,90h>
NOP8_EDX TEXTEQU <DB 08Dh,014h,015h,0,0,0,0,90h>
NOP8_ESI TEXTEQU <DB 08Dh,034h,035h,0,0,0,0,90h>
NOP8 EDI TEXTEQU <DB 08Dh,03Ch,03Dh,0,0,0,0,90h>
NOP8 EBP TEXTEQU <DB 08Dh,02Ch,02Dh,0,0,0,0,90h>
NOP9 TEXTEQU <DB 0EBh,007h,90h,90h,90h,90h,90h,90h>
This chapter describes code optimization techniques that take advantage of the large L1 caches and high-bandwidth buses of the AMD Athlon™ processor. Guidelines are listed in order of importance.

Memory Size and Alignment Issues

Avoid Memory-Size Mismatches

Avoid memory-size mismatches when different instructions operate on the same data. When an instruction stores and another instruction reloads the same data, keep their operands aligned and keep the loads/stores of each operand the same size. The following code examples result in a store-to-load-forwarding (STLF) stall:

Example (avoid):

```assembly
MOV    DWORD PTR [FOO], EAX
MOV    DWORD PTR [FOO+4], EDX
FLD    QWORD PTR [FOO]
```
Example (avoid):

```
MOV       [FOO], EAX
MOV       [FOO+4], EDX
...
MOVQ      MM0, [FOO]
```

Example (preferred):

```
MOV       [FOO], EAX
MOV       [FOO+4], EDX
...
MOVD      MM0, [FOO]
PUNPCKLDQ MM0, [FOO+4]
```

Example (preferred if stores are close to the load):

```
MOVD      MM0, EAX
MOV       [FOO+4], EDX
PUNPCKLDQ MM0, [FOO+4]
```

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

Example (avoid):

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

Example (avoid):

```
MOVQ      [foo], MM0
...
MOV       EAX, [foo]
MOV       EDX, [foo+4]
```

Example (preferred):

```
MOVD      [foo], MM0
PSWAPD    MM0, MM0
MOVD      [foo+4], MM0
PSWAPD    MM0, MM0
...
MOV       EAX, [foo]
MOV       EDX, [foo+4]
```
Example (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]
```

Example (preferred if the stores and loads are close together, option 1):

```
MOVD EAX, MM0
PSWAPD MM0, MM0
MOVD EDX, MM0
PSWAPD MM0, MM0
```

Example (preferred if the stores and loads are close together, option 2):

```
MOVD EAX, MM0
PUNPCKHDQ MM0, MM0
MOVD EDX, MM0
```

Align Data Where Possible

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

- Word accesses are aligned if they access an address divisible by two.
- Doubleword accesses are aligned if they access an address divisible by four.
- Quadword accesses are aligned if they access an address divisible by eight.
- TBYTE accesses are aligned if they access an address divisible by eight.

A misaligned store or load operation suffers a minimum one-cycle penalty in the AMD Athlon processor load/store pipeline. In addition, using misaligned loads and stores increases the likelihood of encountering a store-to-load forwarding pitfall. For a more detailed discussion of store-to-load forwarding issues, see “Store-to-Load Forwarding Restrictions” on page 86.
Optimizing Main Memory Performance for Large Arrays

This section outlines a process for taking advantage of main memory bandwidth by using block prefetch and three-phase processing.

Block prefetch is a technique for reading blocks of data from main memory at very high data rates. Three-phase processing is a programming style that divides data into blocks, which are processed in sequence. Specifically, three-phase processing employs block prefetch to read the input data for each block, operates on each block entirely within the cache, and writes the results to memory with high efficiency.

The prefetch techniques are applicable to applications that access large, localized data objects in system memory, in a sequential or near-sequential manner. The best advantage is realized with data transfers of more than 64 Kbytes. The basis of the techniques is to take best advantage of the processor's cache memory.

The code examples in this section explore the most basic and useful memory function: copying data from one area of memory to another. This foundation is used to explore the main optimization ideas, then these ideas are applied to optimizing a bandwidth-limited function that uses the FPU to process linear data arrays.

The performance metrics were measured for code samples running on a 1.0 GHz AMD Athlon™ 4 processor with DDR2100 memory. The data sizes are chosen to be several megabytes, i.e. much larger than the cache. Exact performance numbers are different on other platforms, but the basic techniques are widely applicable.
Memory Copy Optimization

Memory Copy: Step 1  The simplest way to copy memory is to use the REP MOVSB instruction as used in the Baseline example.

Example Code: Baseline

(bandwidth: ~570 Mbytes/sec)

```assembly
mov esi, [src]     // source array
mov edi, [dst]     // destination array
mov ecx, [len]     // number of QWORDS (8 bytes)
shl ecx, 3        // convert to byte count
rep movsb
```

Memory Copy: Step 2  Starting from this baseline, several optimizations can be implemented to improve performance. The next example increases data size from a byte copy to a doubleword copy using REP MOVSD instruction.

Example Code: Doubleword Copy

(bandwidth: ~700 Mbytes/sec    improvement: 23%)

```assembly
mov esi, [src]     // source array
mov edi, [dst]     // destination array
mov ecx, [len]     // number of QWORDS (8 bytes)
shl ecx, 1        // convert to DWORD count
rep movsd
```
Memory Copy: Step 3

The bandwidth was significantly improved using the doubleword copy. The REP MOVS instructions are often not as efficient as an explicit loop which uses simple "RISC" instructions. The simple explicit instructions can be executed in parallel and sometimes even out-of-order, within the CPU. The explicit loop example uses a loop to perform the copy by using MOV instructions.

Example Code: Explicit Loop

(bandwidth: ~720 Mbytes/sec improvement: 3%)

```
mov esi, [src]      // source array
mov edi, [dst]      // destination array
mov ecx, [len]      // number of QWORDS (8 bytes)
shl ecx, 1          // convert to DWORD count

copyloop:
    mov eax, dword ptr [esi]
    mov dword ptr [edi], eax
    add esi, 4
    add edi, 4
    dec ecx
    jnz copyloop
```

Memory Copy: Step 4

The explicit loop is a bit faster than REP MOVSD. And now that we have an explicit loop, further optimization can be implemented by unrolling the loop. This reduces the overhead of incrementing the pointers and counter, and reduces branching. The unrolled loop example uses the [Register + Offset] form of addressing, which runs just as fast as the simple [Register] address, and uses an unroll factor of four.

Example Code: Unrolled Loop - Unroll Factor Four

(bandwidth: ~700 Mbytes/sec improvement: −3%)  

```
mov esi, [src]     // source array
mov edi, [dst]     // destination array
mov ecx, [len]     // number of QWORDS (8 bytes)
shr ecx, 1         // convert to 16-byte size count
    // (assumes len / 16 is an integer)

copyloop:
    mov eax, dword ptr [esi]
    mov dword ptr [edi], eax
    mov ebx, dword ptr [esi+4]
    mov dword ptr [edi+4], ebx
    mov eax, dword ptr [esi+8]
```
Memory Copy: Step 5

The performance drops when the loop is unrolled, but a new optimization can now be implemented: grouping read operations together and write operations together. In general, it is a good idea to read data in blocks, and write in blocks, rather than alternating frequently.

Example Code: Read and Write Grouping

(bandwidth: ~750 Mbytes/sec     improvement: 7%)

```assembly
mov esi, [src]     // source array
mov edi, [dst]     // destination array
mov ecx, [len]     // number of QWORDS (8 bytes)
shr ecx, 1         // convert to 16-byte size count

copyloop:
  mov eax, dword ptr [esi]
  mov ebx, dword ptr [esi+4]
  mov dword ptr [edi], eax
  mov dword ptr [edi+4], ebx
  mov eax, dword ptr [esi+8]
  mov ebx, dword ptr [esi+12]
  mov dword ptr [edi+8], eax
  mov dword ptr [edi+12], ebx
  add esi, 16
  add edi, 16
  dec ecx
  jnz copyloop
```
Memory Copy: Step 6

The next optimization uses MMX™ extensions, available on all modern x86 processors. The MMX registers permit 64 bytes of sequential reading, followed by 64 bytes of sequential writing. The MMX register example loop also introduces an optimization on the loop counter, which starts negative and counts up to zero. This allows the counter to serve double duty as a pointer, and eliminates the need for a CMP instruction.

Example Code: Grouping Using MMX Registers

(bandwidth: ~800 Mbytes/sec    improvement: 7%)

```assembly
mov esi, [src]    // source array
mov edi, [dst]    // destination array
mov ecx, [len]    // number of QWORDS (8 bytes)
lea esi, [esi+ecx*8]   // end of source
lea edi, [edi+ecx*8]   // end of destination
neg ecx            // use a negative offset
emms

copyloop:
movq mm0, qword ptr [esi+ecx*8]
movq mm1, qword ptr [esi+ecx*8+8]
movq mm2, qword ptr [esi+ecx*8+16]
movq mm3, qword ptr [esi+ecx*8+24]
movq mm4, qword ptr [esi+ecx*8+32]
movq mm5, qword ptr [esi+ecx*8+40]
movq mm6, qword ptr [esi+ecx*8+48]
movq mm7, qword ptr [esi+ecx*8+56]

movq qword ptr [edi+ecx*8], mm0
movq qword ptr [edi+ecx*8+8], mm1
movq qword ptr [edi+ecx*8+16], mm2
movq qword ptr [edi+ecx*8+24], mm3
movq qword ptr [edi+ecx*8+32], mm4
movq qword ptr [edi+ecx*8+40], mm5
movq qword ptr [edi+ecx*8+48], mm6
movq qword ptr [edi+ecx*8+56], mm7

add ecx, 8
jnz copyloop

emms
```

Example Code: Grouping Using MMX Registers

(bandwidth: ~800 Mbytes/sec    improvement: 7%)

```assembly
mov esi, [src]    // source array
mov edi, [dst]    // destination array
mov ecx, [len]    // number of QWORDS (8 bytes)
lea esi, [esi+ecx*8]   // end of source
lea edi, [edi+ecx*8]   // end of destination
neg ecx            // use a negative offset
emms

copyloop:
movq mm0, qword ptr [esi+ecx*8]
movq mm1, qword ptr [esi+ecx*8+8]
movq mm2, qword ptr [esi+ecx*8+16]
movq mm3, qword ptr [esi+ecx*8+24]
movq mm4, qword ptr [esi+ecx*8+32]
movq mm5, qword ptr [esi+ecx*8+40]
movq mm6, qword ptr [esi+ecx*8+48]
movq mm7, qword ptr [esi+ecx*8+56]

movq qword ptr [edi+ecx*8], mm0
movq qword ptr [edi+ecx*8+8], mm1
movq qword ptr [edi+ecx*8+16], mm2
movq qword ptr [edi+ecx*8+24], mm3
movq qword ptr [edi+ecx*8+32], mm4
movq qword ptr [edi+ecx*8+40], mm5
movq qword ptr [edi+ecx*8+48], mm6
movq qword ptr [edi+ecx*8+56], mm7

add ecx, 8
jnz copyloop

emms
```
MOVNTQ can be used now that MMX™ registers are being used. This is a streaming store instruction, for writing data to memory. This instruction bypasses the on-chip cache, and goes directly into a write combining buffer, effectively increasing the total write bandwidth. The MOVNTQ instruction executes much faster than an ordinary MOV to memory. An SFENCE is required to flush the write buffer.

Example Code: MOVNTQ and SFENCE Instructions

(bandwidth: ~1120 Mbytes/sec      improvement: 32%)

```assembly
mov esi, [src]          // source array
mov edi, [dst]          // destination array
mov ecx, [len]          // number of QWORDS (8 bytes)
lea esi, [esi+ecx*8]
lea edi, [edi+ecx*8]

neg ecx
emms

copyloop:
  movq mm0, qword ptr [esi+ecx*8]
  movq mm1, qword ptr [esi+ecx*8+8]
  movq mm2, qword ptr [esi+ecx*8+16]
  movq mm3, qword ptr [esi+ecx*8+24]
  movq mm4, qword ptr [esi+ecx*8+32]
  movq mm5, qword ptr [esi+ecx*8+40]
  movq mm6, qword ptr [esi+ecx*8+48]
  movq mm7, qword ptr [esi+ecx*8+56]

  movntq qword ptr [edi+ecx*8], mm0
  movntq qword ptr [edi+ecx*8+8], mm1
  movntq qword ptr [edi+ecx*8+16], mm2
  movntq qword ptr [edi+ecx*8+24], mm3
  movntq qword ptr [edi+ecx*8+32], mm4
  movntq qword ptr [edi+ecx*8+40], mm5
  movntq qword ptr [edi+ecx*8+48], mm6
  movntq qword ptr [edi+ecx*8+56], mm7

  add ecx, 8
  jnz copyloop

  sfence
  emms
```
Memory Copy: Step 8

The MOVNTQ instruction in the previous example improves the speed of writing the data. The Prefetch Instruction example uses a prefetch instruction to improve the performance on reading the data. Prefetching cannot increase the total read bandwidth, but it can get the processor started on loading the data to the cache before the data is needed.

Example Code: Prefetch Instruction (prefetchnta)

(bandwidth: ~1250 Mbytes/sec      improvement: 12%)

```assembly
    mov    esi, [src]    // source array
    mov    edi, [dst]    // destination array
    mov    ecx, [len]    // number of QWORDS (8 bytes)
    lea    esi, [esi+ecx*8]
    lea    edi, [edi+ecx*8]
    neg    ecx
    emms

    copyloop:          
        prefetchnta  [esi+ecx*8 + 512]
        movq     mm0, qword ptr [esi+ecx*8]
        movq     mm1, qword ptr [esi+ecx*8+8]
        movq     mm2, qword ptr [esi+ecx*8+16]
        movq     mm3, qword ptr [esi+ecx*8+24]
        movq     mm4, qword ptr [esi+ecx*8+32]
        movq     mm5, qword ptr [esi+ecx*8+40]
        movq     mm6, qword ptr [esi+ecx*8+48]
        movq     mm7, qword ptr [esi+ecx*8+56]
        movntq  qword ptr [edi+ecx*8],    mm0
        movntq  qword ptr [edi+ecx*8+8],  mm1
        movntq  qword ptr [edi+ecx*8+16], mm2
        movntq  qword ptr [edi+ecx*8+24], mm3
        movntq  qword ptr [edi+ecx*8+32], mm4
        movntq  qword ptr [edi+ecx*8+40], mm5
        movntq  qword ptr [edi+ecx*8+48], mm6
        movntq  qword ptr [edi+ecx*8+56], mm7
        add    ecx, 8
        jnz    copyloop
        sfence
        emms
```
Memory Copy: Step 9 (final)

In the final optimization to the memory copy code, the technique called block prefetch is applied. Much as read grouping gave a boost to performance, block prefetch is an extreme extension of this idea. The strategy is to read a large stream of sequential data from main memory into the cache, without any interruptions.

In block prefetch, the MOV instruction is used, rather than the software prefetch instruction. Unlike a prefetch instruction, the MOV instruction cannot be ignored by the CPU. The result is that a series of MOVs will force the memory system to read sequential, back-to-back address blocks, which maximizes memory bandwidth.

And because the processor always loads an entire cache line (e.g. 64 bytes) whenever it accesses main memory, the block prefetch MOV instructions only need to read ONE address per cache line. Reading just one address per cache line is a subtle trick, which is essential in achieving maximum read performance.

Example: Block Prefetching

(bandwidth: ~1630 Mbytes/sec   improvement: 30%)

#define CACHEBLOCK 400h   // QWORDS in a block (8K bytes)

mov esi, [src]     // source array
mov edi, [dst]     // destination array
mov ecx, [len]   // total number of QWORDS (8 bytes)
// (assumes len / CACHEBLOCK = integer)
lea esi, [esi+ecx*8]
lea edi, [edi+ecx*8]

neg ecx
emms

mainloop:

mov eax, CACHEBLOCK / 16  // note: prefetch loop is unrolled 2X
Array Addition

The following Two Array Addition example applies the block prefetch technique and other concepts from the memory copy optimization example, and optimizes a memory-intensive loop that processes large arrays.

Baseline Code

This loop adds two arrays of floating-point numbers together, using the x87 FPU, and writes the results to a third array. This example also shows how to handle the issue of combining MMX code (required for using the MOVNTQ instruction) with FPU code (needed for adding the numbers).

The Two Array Baseline example is a slightly optimized, first pass, baseline version of the code.
Example: Two Array Add Baseline

(bandwidth: ~840 MB/sec baseline performance)

```assembly
mov esi, [src1]      // source array one
mov ebx, [src2]      // source array two
mov edi, [dst]       // destination array

mov ecx, [len]      // number of Floats (8 bytes)
// (assumes len / 8 = integer)
lea esi, [esi+ecx*8]
lea ebx, [ebx+ecx*8]
lea edi, [edi+ecx*8]

neg ecx

addloop:

fld qword ptr [esi+ecx*8+56]
fadd qword ptr [ebx+ecx*8+56]
fld qword ptr [esi+ecx*8+48]
fadd qword ptr [ebx+ecx*8+48]
fld qword ptr [esi+ecx*8+40]
fadd qword ptr [ebx+ecx*8+40]
fld qword ptr [esi+ecx*8+32]
fadd qword ptr [ebx+ecx*8+32]
fld qword ptr [esi+ecx*8+24]
fadd qword ptr [ebx+ecx*8+24]
fld qword ptr [esi+ecx*8+16]
fadd qword ptr [ebx+ecx*8+16]
fld qword ptr [esi+ecx*8+8]
fadd qword ptr [ebx+ecx*8+8]
fld qword ptr [esi+ecx*8+0]
fadd qword ptr [ebx+ecx*8+0]

fstp qword ptr [edi+ecx*8+0]
fst p qword ptr [edi+ecx*8+8]
fst p qword ptr [edi+ecx*8+16]
fst p qword ptr [edi+ecx*8+24]
fst p qword ptr [edi+ecx*8+32]
fst p qword ptr [edi+ecx*8+40]
fst p qword ptr [edi+ecx*8+48]
fst p qword ptr [edi+ecx*8+56]
add ecx, 8
jnz addloop
```
Optimized Code

After all the relevant optimization techniques have been applied, the code appears as in the Two Array Add with Optimizations Example. The data is still processed in blocks, as in the memory copy example. But in this case, the code must process the data using the FPU, not simply copy it. Because of this need for FPU mode operation, the processing is divided into three distinct phases: block prefetch, processing, and memory write. The block prefetch phase reads the input data into the cache at maximum bandwidth. The processing phase operates on the in-cache input data and writes the results to an in-cache temporary buffer. The memory write phase uses MOVNTQ to quickly transfer the temporary buffer to the destination array in main memory.

These three phases are the components of three phase processing. This general technique provides a significant performance boost, as seen in this optimized code.

Example: Two Array Add with Optimizations

(bandwidth: ~1370 MB/sec improvement: 63% )

#define CACHEBLOCK 400h // QWORDS in a block, (8K bytes)
int* storedest
char buffer[CACHEBLOCK * 8] // in-cache temporary storage

mov esi. [src1] // source array one
mov ebx. [src2] // source array two
mov edi. [dst] // destination array

mov ecx. [len] // number of Floats (8 bytes)
// (assumes len /CACHEBLOCK = integer)
lea esi. [esi+ecx*8]
lea ebx. [ebx+ecx*8]
lea edi. [edi+ecx*8]

mov [storedest], edi // save the real dest for later
mov edi. [buffer] // temporary in-cache buffer...
lea edi. [edi+ecx*8] // stays in cache from heavy use
neg ecx

mainloop:
mov eax, CACHEBLOCK / 16

prefetchloop1: // block prefetch array #1
mov edx. [esi+ecx*8]
mov edx. [esi+ecx*8+64] // (this loop is unrolled 2X)
add ecx, 16
dec eax
jnz prefetchloop1
sub ecx, CACHEBLOCK
mov eax, CACHEBLOCK / 16

prefetchloop2: // block prefetch array #2
mov edx, [ebx+ecx*8]
mov edx, [ebx+ecx*8+64] // (this loop is unrolled 2X)
add ecx, 16
dec eax
jnz prefetchloop2
sub ecx, CACHEBLOCK
mov eax, CACHEBLOCK / 8

processloop: // this loop read/writes all in cache!
fld qword ptr [esi+ecx*8+56]
fadd qword ptr [ebx+ecx*8+56]
...
fld qword ptr [esi+ecx*8+0]
fadd qword ptr [ebx+ecx*8+0]
fstp qword ptr [edi+ecx*8+0]
...
fstp qword ptr [edi+ecx*8+56]
add ecx, 8
dec eax
jnz processloop

emms
sub ecx, CACHEBLOCK
mov edx, [storedest]
mov eax, CACHEBLOCK / 8

writeloop: // write buffer to main mem
movq mm0, qword ptr [edi+ecx*8]
...
movq mm7, qword ptr [edi+ecx*8+56]
movntq qword ptr [edx+ecx*8], mm0
...
movntq qword ptr [edx+ecx*8+56], mm7
add ecx, 8
dec eax
jnz writeloop
or ecx, ecx
jge exit
sub edi, CACHEBLOCK * 8 // reset edi back to start of
// buffer
sfence
eums

jmp mainloop
exit:

Summary

Block prefetch and three phase processing are general techniques for improving the performance of memory-intensive applications. The key points are:

- To get the maximum memory read bandwidth, read data into the cache in large blocks, using block prefetch. A block prefetch loop should:
  - Be unrolled by at least 2X
  - Use the MOV instruction (not the Prefetch instruction)
  - Read only one address per cache line
  - Read data into an ALU scratch register, e.g. EAX
  - Make sure all data is aligned
  - Read only one address stream per loop
  - Use separate loops to prefetch several streams

- To get maximum memory write bandwidth, write data from the cache to main memory in large blocks, using streaming store instructions. The write loop should:
  - Use the MMX registers to pass the data
  - Read from cache
  - Use MOVNTQ (streaming store) for writing to memory
  - Make sure the data is aligned
  - Write every address, in ascending order, without gaps
  - End with an SFENCE to flush the write buffer

- Whenever possible, code that actually “does the real work” should read data from cache, and write output to an in-cache buffer. To enable this cache access, follow the first two guidelines above.

Align branch targets to 16-byte boundaries, in these critical sections of code. This optimization is described in “Align Branch Targets in Program Hot Spots” on page 54.
Use the PREFETCH 3DNow!™ Instruction

For code that can take advantage of prefetching, and situations where small data sizes or other constraints limit the applicability of block prefetch optimizations, use the 3DNow! PREFETCH and PREFETCHW instructions to increase the effective bandwidth to the AMD Athlon processor.

The PREFETCH and PREFETCHW instructions take advantage of the high bus bandwidth of the AMD Athlon processor to hide long latencies when fetching data from system memory.

The prefetch instructions are essentially integer instructions and can be used anywhere, in any type of code (integer, x87, 3DNow!, MMX, etc.).

Prefetching versus Preloading

In code that uses the block prefetch technique as described in “Optimizing Main Memory Performance for Large Arrays” on page 66, a standard load instruction is the best way to prefetch data. But in other situations, load instructions may be able to mimic the functionality of prefetch instructions, but they do not offer the same performance advantage. Prefetch instructions only update the cache line in the L1/L2 cache and do not update an architectural register. This uses one less register compared to a load instruction. Prefetch instructions also do not cause normal instruction retirement to stall.

Another benefit of prefetching versus preloading is that the prefetching instructions can retire even if the load data has not arrived yet. A regular load used for preloading will stall the machine if it gets to the bottom of the fixed-issue reorder buffer (part of the Instruction Control Unit) and the load data has not arrived yet. The load is "blocking" whereas the prefetch is "non-blocking."

Unit-Stride Access

Large data sets typically require unit-stride access to ensure that all data pulled in by PREFETCH or PREFETCHW is actually used. If necessary, reorganize algorithms or data structures to allow unit-stride access. See “Definitions” on page 84 for a definition of unit-stride access.
Hardware Prefetch

Some AMD Athlon processors implement a hardware prefetch mechanism. This feature was implemented beginning with AMD Athlon processor Model 6. The 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 processors are 64 bytes, but cache line size is implementation dependent.

In some cases, using the PREFETCH or PREFETCHW instruction on processors with hardware prefetch may incur a reduction in performance. In these cases, the PREFETCH instruction may need to be removed. The engineer needs to weigh the measured gains obtained on non-hardware prefetch enabled processors by using the PREFETCH instruction, versus any loss in performance on processors with the hardware prefetcher.

PREFETCH/W versus PREFETCHNTA/T0/T1/T2

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

PREFETCHW Usage

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

Programmers can initiate multiple outstanding prefetches on the AMD Athlon processor. While the AMD-K6-2 and AMD-K6-III processors can have only one outstanding prefetch, the AMD Athlon processor can have up to six outstanding prefetches. When all six buffers are filled by various memory read requests, the processor will simply ignore any new prefetch requests until a buffer frees up. Multiple prefetch requests are essentially handled in-order. Prefetch data in the order that it is needed.

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

Example 1: Multiple Prefetches Code

```assembly
.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) ;used 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+128] ;two cachelines ahead
PREFETCH [EBX+128] ;two cachelines ahead
PREFETCH [ECX+128] ;two cachelines 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+8] ;b[i+1]
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]
```
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.

- Since the array "array_a" is written rather than read, use PREFETCHW instead of PREFETCH to avoid overhead for switching cache lines to the correct MESI state. The PREFETCH lookahead is optimized such that each loop iteration is working on three cache lines while six active PREFETCHes bring in the next six 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 six PREFETCHes in flight at any given time. As processors achieve speeds of 1 GHz and faster, the second constraint starts to limit how far ahead a programmer can PREFETCH.

Formula

Given the latency of a typical AMD Athlon processor system and expected processor speeds, use the following formula to determine the prefetch distance in bytes for a single array:

\[\text{Prefetch Distance} = 200 \times \frac{\text{DS}}{\text{C}} \text{ bytes}\]

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

Programmers should isolate the loop and have the loop work on a data set that fits in L1 and determine the L1 loop time.

\[\text{L1\_loop\_time} = \frac{\text{execution time in cycles}}{\# \text{ loop iterations}}\]

Where multiple arrays are being prefetched, the prefetch distance usually needs to be increased over what the above formula suggests, as prefetches for one array are delayed by prefetches to a different array.
Definitions

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

```c
char j, k[MAX];
for (i=0; i<MAX; i++) {
  ...
  j += k[i];
  ...
}
```
```
double x, y[MAX];
for (i=0; i<MAX; i++) {
  ...
  x += y[i];
  ...
}
```

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 PREFETCH properly. For example, assume 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 (e.g., 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"):

```c
PREFETCH [EAX+n*STRUCTURE_SIZE]
...
ADD EAX, STRUCTURE_SIZE
```

Programmers may need to 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 refers to the number of bytes accessed in the array per loop iteration. For example:

```assembly
FLDZ $add_loop:
FADD QWORD PTR [EBX*8+base_address]
DEC EBX
JNZ $add_loop
```

The data stride of the above loop is 8 bytes. In general, for optimal use of prefetch, the data stride per iteration is the length of a cache line (64 bytes in the AMD Athlon processor).
If the "loop stride" is smaller, unroll the loop. Note that this can be unfeasible if the original loop stride is very small, e.g., 2 bytes.

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

Take Advantage of Write Combining

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


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

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

For example, programmers should consider that a memory indirect JMP instruction may have the data for the jump table residing in the same 64-byte cache line as the JMP instruction, which would result in lower performance.
Although unlikely, do not place critical code at the border between 32-byte aligned code segments and data segments. The code at the start or end of your data segment should be executed as infrequently as possible or simply padded with garbage.

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

**Multiprocessor Considerations**

Sharing data between processors which reside in the same cache line can reduce performance. Whenever possible, restructure the data organization so this does not occur. Cache lines on AMD Athlon™ processors are presently 64 bytes but a scheme which avoids this problem regardless of cache line size makes for more performance portable code.

**Store-to-Load Forwarding Restrictions**

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

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

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

Due to the significant depth of the AMD Athlon processor LS buffer, any load dependent on a store that cannot bypass data
through LS can 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 that are acceptable and those to avoid.

**Store-to-Load Forwarding Pitfalls—True Dependencies**

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

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

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

**Narrow-to-Wide Store-Buffer Data Forwarding Restriction**

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

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

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

**Example 1 (avoid):**

```
MOV EAX, 10h
MOV WORD PTR [EAX], BX ;word store
...
MOV ECX, DWORD PTR [EAX] ;doubleword load ;cannot forward upper byte from store buffer
```
Example 2 (avoid):
MOV EAX, 10h
MOV BYTE PTR [EAX + 3], BL ; byte store
...
MOV ECX, DWORD PTR [EAX] ; doubleword load
; cannot forward upper byte
; from store buffer

Wide-to-Narrow Store-Buffer Data Forwarding Restriction

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

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

Example 3 (avoid):
MOV EAX, 10h
ADD DWORD PTR [EAX], EBX ; doubleword store
MOV CX, WORD PTR [EAX + 2] ; word load—cannot forward high
; word from store buffer

Example 4 (avoid):
MOVD [foo], MM1 ; store upper and lower half
...
ADD EAX, [foo] ; fine
ADD EDX, [foo+4] ; not good!

Example 5 (preferred):
MOVD [foo], MM1 ; store lower half
PUNPCKHDQ MM1, MM1 ; get upper half into lower half
MOVD [foo+4], MM1 ; store lower half
...
ADD EAX, [foo] ; fine
ADD EDX, [foo+4] ; fine

Misaligned Store-Buffer Data Forwarding Restriction

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

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

A common case of misaligned store-data forwarding involves the passing of misaligned quadword floating-point data on the
doubleword-aligned integer stack. Avoid the type of code shown in the following example.

**Example 6 (avoid):**

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

**High-Byte Store-Buffer Data Forwarding Restriction**

If the following condition is present, there is a high-byte store-data buffer forwarding restriction – the store data is from a high-byte register (AH, BH, CH, DH).

Avoid the type of code shown in the following example.

**Example 6 (avoid):**

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

**One Supported Store-to-Load Forwarding Case**

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

**Example 7 (allowed):**

```
MOVQ [AlignedQword], mm0
...
MOV EAX, [AlignedQword]
```
Summary of Store-to-Load Forwarding Pitfalls to Avoid

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

- Maintain consistent use of operand size across all loads and stores. Preferably, use doubleword or quadword operand sizes.
- Avoid misaligned data references.
- Avoid narrow-to-wide and wide-to-narrow forwarding cases.
- When using word or byte stores, avoid loading data from anywhere in the same doubleword of memory other than the identical start addresses of the stores.

Stack Alignment Considerations

Make sure the stack is suitably aligned for the local variable with the largest base type. Then, using the technique described in “C Language Structure Component Considerations” on page 91, all variables can be properly aligned with no padding.

Extend to 32 Bits Before Pushing onto Stack

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

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

Example 1 (preferred):

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 via EBP, and local variables can be accessed via ESP. In order to free EBP for general use, it needs to be saved and restored between the prologue and the epilogue.

**Align TBYTE Variables on Quadword Aligned Addresses**

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

**C Language Structure Component Considerations**

Structures (‘struct’ in C language) should be made the size of a multiple of the largest base type of any of their components. To meet this requirement, use padding where necessary. This ensures that all elements of an array of structures are properly aligned provided the array itself is properly aligned.

To minimize padding, sort and allocate structure components (language definitions permitting) such that the components with a larger base type are allocated ahead of those with a smaller base type. For example, consider the following code:

**Example 1:**

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

Allocate the structure components (lowest to highest address) as follows:

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

See “C Language Structure Component Considerations” on page 36 for more information from a C source code perspective.
Sort Variables According to Base Type Size

Sort local variables according to their base type size and allocate variables with larger base type size ahead of those with smaller base type size. Assuming the first variable allocated is naturally aligned, all other variables are naturally aligned without any padding. The following example is a declaration of local variables in a C function:

Example 1:

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

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

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

See “Sort Local Variables According to Base Type Size” on page 37 for more information from a C source code perspective.
Branch Optimizations

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

Avoid Branches Dependent on Random Data

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

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

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

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

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

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

**Blended AMD-K6® and AMD Athlon™ Processor Code**

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

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

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

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

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

MOV AL, [X] ;load X value
CMP AL, 10 ;if x is less than 10, set carry flag
SBB AL, 69h ;0..9 -> 96h, Ah..Fh -> A1h...A6h
DAS ;0..9: subtract 66h, Ah..Fh: Sub.
60h
MOV [Y], AL ;save conversion in y
Example 6 — Increment Ring Buffer Offset:

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

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

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

Example 7 — Integer Signum Function:

//C Code
int a, s;

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

;Assembly Code
MOV     EAX, [a]         ;load a
CDQ                      ;t = a < 0 ? 0xffffffff : 0
CMP     EDX, EAX         ;a > 0 ? CF : NC
ADC     EDX, 0           ;a > 0 ? t+1 : t
MOV     [s], EDX         ;signum(x)
Example 8 – 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

Always Pair CALL and RETURN

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

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

Because returns are predicted as described in the prior section, recursive functions should be written carefully. If there are only recursive function calls within the function as shown in Example 1, the return address for each iteration of the recursive function is properly predicted.

Example 1 (Preferred):

```c
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 Example 2, some returns can be mis-predicted. If the number of recursive function calls plus the number of non-recursive 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.

Example 2 (Avoid):

```c
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;
}
```
Since the function `fac` in Example 2 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. Example 3 shows the re-written code.

**Example 3 (Preferred):**
```c
long fac1(long a)
{
    long t=1;
    while (a > 0) {
        myp(a);
        t *= a;
        a--;
    }
    return (t);
}
```

**Replace Branches with Computation in 3DNow!™ Code**

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

**Muxing Constructs**

The most important construct to avoiding branches in 3DNow! and MMX™ code is a 2-way muxing construct that is equivalent to the ternary operator “?::” in C and C++. It is implemented using the PCMP/PFCMP, PAND, PANDN, and POR instructions. To maximize performance, it is important to apply the PAND and PANDN instructions in the proper order.
Example 1 (Avoid):

: r = (x < y) ? a : b

: in:  mm0  a
:      mm1  b
:      mm2  x
:      mm3  y
: out: mm0 r

PCMPGT 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 PCMP, the mask needs to be saved, which requires an additional register. This adds an instruction, lengthens the dependency chain, and increases register pressure. Therefore, write 2-way muxing constructs as follows.

Example 1 (Preferred):

: r = (x < y) ? a : b

: in:  mm0  a
:      mm1  b
:      mm2  x
:      mm3  y
: out: mm0 r

PCMPGT 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 3DNow!™ Code

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

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

Example 2

C code:

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

3DNow! code:

```assembly
;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 3

C code:

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

3DNow! code:

```assembly
;in: MM0 = x
;out: MM0 = z
MOVQ MM5, mabs ;0xffffffff
PAND MM0, MM5 ;z=abs(x)
PFRCP MM2, MM0 ;1/z approx
MOVQ MM1, MM0 ;save z
```
Example 4

C code:

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

3DNow! code:

```asm
; 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
PCMPGT D 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 MM1, MM0 ; z < 0.575 ? r : 0
PANDN MM2, MM1 ; z < 0.575 ? 0 : pi/2 - 2*r
POR MM0, MM0 ; z < 0.575 ? r : pi/2 - 2 * r
```
Example 5

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

3DNow! code:
;in: MM0 = x  
;    MM1 = r  
;out: MM1 = res
MOVQ  MM5,  mabs ; mask to clear sign bit
MOVQ  MM6,  one  ; 1.0
PAND  MM0, MM5 ; z=abs(x)
PCMPGT D MM6, MM0 ; z < 1 ? 0xffffffff : 0
MOVQ  MM4, MM2 ; 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 6

**C code:**

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

**3DNow! code:**

```assembly
;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 ;ys
PCMPGTDD 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
PSRAWD MM6, 31 ;df ? 0xffffffff : 0
PANDMM MM6, MM5 ;xs ? (df ? 0 : pi/2) : (df ? 0 : -pi/2)
PFSUB MM6, MM3 ;pr = pi/2 + (xs ? (df ? 0 : pi/2) :
    ; (df ? 0 : -pi/2))
POR MM0, MM7 ;ar = xs^df ? -r : r
PFADD MM0, MM6 ;res = ar + pr```
Avoid the Loop Instruction

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

**Example 1 (Avoid):**
```
LOOP LABEL
```

**Example 1 (Preferred):**
```
DEC ECX
JNZ LABEL
```

Avoid Far Control Transfer Instructions

Avoid using far control transfer instructions. Far control transfer branches cannot be predicted by the branch target buffer.
This chapter describes how to code instructions for efficient scheduling. Guidelines are listed in order of importance.

**Schedule Instructions According to their Latency**

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

See Appendix F, “Instruction Dispatch and Execution Resources/Timing,” for a list of latency numbers.
Unrolling Loops

Complete Loop Unrolling

Make use of the large AMD Athlon processor 64-Kbyte instruction cache and unroll loops to get more parallelism and reduce loop overhead, even with branch prediction. Complete unrolling reduces register pressure by removing the loop counter. To completely unroll a loop, remove the loop control and replicate the loop body N times. In addition, completely unrolling a loop increases scheduling opportunities.

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

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

Partial Loop Unrolling

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

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

Consider the following piece of C code:

double a[MAX_LENGTH], b[MAX_LENGTH];

    for (i=0; i< MAX_LENGTH; i++) {
        a[i] = a[i] + b[i];
    }
Without loop unrolling, the code looks like the following:

**Example 1 (Without Loop Unrolling):**

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

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

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

**Example 1 (With Partial Loop Unrolling):**

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

$add_loop:
FLD QWORD PTR [EAX]
FADD QWORD PTR [EBX]
FSTP QWORD PTR [EAX]
FLD QWORD PTR [EAX+8]
FADD QWORD PTR [EBX+8]
FSTP QWORD PTR [EAX+8]
ADD EAX, 16
ADD EBX, 16
DEC ECX
JNZ $add_loop
```
Now the loop consists of ten instructions. Based on the decode/retire bandwidth of three MacroOPs per cycle, this loop goes no faster than three iterations in ten cycles, or 6/10 floating-point adds per cycle, or 1.4 times as fast as the original loop.

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

**Example 2 (Rolled Loop):**

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

**Example 2 (Partially Unrolled Loop):**

```c
for (k = lo; k <= (hi - (fac-1)*inc); k += fac*inc) {
    x[k] = ...
    x[k+inc] = ...
    ...
    x[k+(fac-1)*inc] = ...
}
/* handle end cases */
for (k = k; k <= hi; k += inc) {
    x[k] = ...
}
```
Use Function Inlining

Overview

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

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

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

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

Always inline a function if it can be established that this 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. This case occurs quite frequently, as functionality that could be concentrated in a single large function is split across multiple small functions for improved maintainability and readability.

Always Inline Functions with Fewer than 25 Machine Instructions

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

Avoid Address Generation Interlocks

Loads and stores are scheduled by the AMD Athlon processor to access the data cache in program order. Newer loads and stores with their addresses calculated can be blocked by older loads and stores whose addresses are not yet calculated—this is known as an address generation interlock. Therefore, it is advantageous to schedule loads and stores that can calculate their addresses quickly, ahead of loads and stores that require the resolution of a long dependency chain in order to generate their addresses. Consider the following code examples:
Example 1 (Avoid):
ADD EBX, ECX ;inst 1
MOV EAX, DWORD PTR [10h] ;inst 2 (fast address calc.)
MOV ECX, DWORD PTR [EAX+EBX] ;inst 3 (slow address calc.)
MOV EDX, DWORD PTR [24h] ;this load is stalled from accessing data cache due to long latency for generating address for ; inst 3

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

Use MOVZX and MOVSX

Use the MOVZX and MOVSX instructions to zero-extend and sign-extend byte-size and word-size operands to doubleword length. Typical code for zero extension that replaces MOVZX, as shown in Example 1 (Avoid), 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 1 (Avoid):
XOR EAX, EAX
MOV AL, [MEM]

Example 1 (Preferred):
MOVZX EAX, BYTE PTR [MEM]
Minimize Pointer Arithmetic in Loops

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

Example 1 (Avoid):

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

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

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

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

Example 1 (Preferred):

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

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

```assembly
MOV ECX, MAXSIZE-1;initialize loop counter

$add_loop:
MOV EAX, [ECX*4 + a] ;get element a
MOV EDX, [ECX*4 + b] ;get element b
ADD EAX, EDX        ;a[i] + b[i]
MOV [ECX*4 + c], EAX ;write result to c
DEC ECX              ;decrement index
JNS $add_loop        ;until index negative
```
Note that the code in the preferred example traverses the arrays in a downward direction (i.e., from higher addresses to lower addresses), whereas the original code to avoid traverses the arrays in an upward direction. Such a change in the direction of the traversal is possible if each loop iteration is completely independent of all other loop iterations, as is the case here.

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

**Example 2 (Preferred):**

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

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

MOV ECX, (-MAXSIZE) ;initialize index

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

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

Example 1 (Avoid):
MOV EAX, [MEM]
PUSH EAX

Example 2 (Preferred):
PUSH [MEM]
This chapter describes ways to improve integer performance through optimized programming techniques. The guidelines are listed in order of importance.

Replace Divides with Multiplies

Replace integer division by constants with multiplication by the reciprocal. Because the AMD Athlon™ processor has a very fast integer multiply (5–9 cycles signed, 4–8 cycles unsigned) and the integer division delivers only one bit of quotient per cycle (22–47 cycles signed, 17–41 cycles unsigned), the equivalent code is much faster. The user can follow the examples in this chapter that illustrate the use of integer division by constants, or access the executables in the opt_utilities directory in the AMD documentation CD-ROM (order no. 21860) to find alternative code for dividing by a constant.
Multiplication by Reciprocal (Division) Utility

The code for the utilities can be found at “Derivation of Multiplier Used for Integer Division by Constants” on page 144. All utilities were compiled for the Microsoft® Windows 95, Windows 98, and Windows NT® environments. All utilities are provided ‘as is’ and are not supported by AMD.

Signed Division Utility

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

Unsigned Division Utility

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

Unsigned Division by Multiplication of Constant

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

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

```assembly
; 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
```

Determination of a, m, s

How to determine 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 Unsigned Integer Division” on page 144.
Algorithm: Divisors
Where \(2^{31} \leq d < 2^{32}\)

For divisors \(2^{31} \leq d < 2^{32}\), the possible quotient values are either 0 or 1. This makes it easy to establish the quotient by simple comparison of the dividend and divisor. In cases where the dividend needs to be preserved, Example 1 is recommended.

Example 1:

; 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

In cases where the dividend does not need to be preserved, the division can be accomplished without the use of an additional register, thus reducing register pressure. This is shown in Example 2 below:

Example 2:

; 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
Where 2 \( \leq d < 2^{31} \)

These algorithms work if the divisor is positive. If the divisor is negative, use \( \text{abs}(d) \) instead of \( d \), and append a ‘NEG EDX’ 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

Determination for a, m, s
How to determine the algorithm (a), multiplier (m), and shift factor (s) is found in the section “Derivation of Algorithm, Multiplier, and Shift Factor for Signed Integer Division” on page 148.

Signed Division by 2

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

Signed Division by 2^n

;IN: EAX = dividend
;OUT: EAX = quotient
CDQ ;Sign extend into EDX
AND EDX, (2^{n}-1) ;Mask correction (use divisor -1)
ADD EAX, EDX ;Apply correction if necessary
SAR EAX, (n) ;Perform right shift by \( \log_2 \) (divisor)
Signed Division by $-2$

; IN: \ EAX = dividend
; OUT: \ EAX = quotient
CMP EAX, 800000000h ; CY = 1, if dividend >= 0
SBB EAX, -1 ; Increment dividend if it is < 0
SAR EAX, 1 ; Perform right shift
NEG EAX ; Use \( x/(-2) = -(x/2) \)

Signed Division by $-2^n$ for $n \geq 0$

; 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 \( \log_2(-\text{divisor}) \)
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 EDX, 1 ; Compute remainder
XOR EAX, EDX ; Negate remainder if
SUB EAX, EDX ; Dividend was < 0

Remainder of Signed Division $2^n$ or $-2^n$ for $n \geq 0$

; 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)
ADD EAX, EDX ; Apply pre-correction if necessary
Consider Alternative Code When Multiplying by a Constant

A 32-bit integer multiplied by a constant has a latency of five cycles. For certain constant multipliers, instruction sequences can be devised which accomplish the multiplication with lower latency. Since the AMD Athlon processor contains only one integer multiplier, but three integer execution units, the throughput of the replacement code may 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 using an integer multiply with a constant is already high, it might still be better for 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 such that the original source also receives the final result. Other sequences are possible if the result is in a different register. Sequences requiring no temporary register have been favored over ones requiring a temporary register even if the latency is higher. ALU operations have preferred over shifts to keep code size small. Similarly, both ALU operations and shifts have been favored over the LEA instruction.

Replacement sequences for other multipliers are found in the file multiply_by_constants.txt located in the same directory where this document is located in the SDK. The user may also use the program “FINDMUL” to find the appropriate sequence for other multipliers. FINDMUL is located in the opt_utilities directory of the AMD Documentation CD-ROM.
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:  LEA  REG1, [REG1+REG1*2]   ;4 cycles
       LEA  REG1, [REG1+REG1*4]
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
SHLREG1, 4
ADDREG1, REG2

by 20: LEA REG1, [REG1+REG1*4] ;3 cycles
SHLREG1, 2

by 21: LEA REG2, [REG1+REG1*4] ;3 cycles
SHLREG1, 4
ADDREG1, REG2

by 22: LEA REG2, [REG1+REG1*4] ;4 cycles
ADDREG1, REG1
LEA REG1, [REG1+REG2*4]

by 23: LEA REG2, [REG1+REG1*8] ;3 cycles
SHLREG1, 5
SUBREG1, REG2

by 24: LEA REG1, [REG1+REG1*2] ;3 cycles
SHLREG1, 3

by 25: LEA REG1, [REG1+REG1*4] ;4 cycles
LEA REG1, [REG1+REG1*4]

by 26: LEA REG2, [REG1+REG1*2] ;4 cycles
ADDREG1, REG1
LEA REG1, [REG1+REG2*8]

by 27: LEA REG1, [REG1+REG1*2] ;4 cycles
LEA REG1, [REG1+REG1*8]

by 28: LEA REG2, [REG1*4] ;3 cycles
SHLREG1, 5
SUBREG1, REG2

by 29: LEA REG2, [REG1+REG1*2] ;3 cycles
SHLREG1, 5
SUBREG1, REG2

by 30: LEA REG2, [REG1+REG1] ;3 cycles
SHLREG1, 5
SUBREG1, REG2

by 31: MOV REG2, REG1 ;2 cycles
SHLREG1, 5
SUBREG1, REG2

by 32: SHLREG1, 5 ;1 cycle
Use MMX™ Instructions for Integer-Only Work

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

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

Be careful with regards to passing data between MMX and integer registers and of creating mismatched store-to-load forwarding cases. See “Unrolling Loops” on page 106.

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

Repeated String Instruction Usage

Latency of Repeated String Instructions

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

Table 1. Latency of Repeated String Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ECX=0 (cycles)</th>
<th>DF = 0 (cycles)</th>
<th>DF = 1 (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>REP MOVES</td>
<td>11</td>
<td>15 + (4/3*c)</td>
<td>25 + (4/3*c)</td>
</tr>
<tr>
<td>REP STOS</td>
<td>11</td>
<td>14 + (1*c)</td>
<td>24 + (1*c)</td>
</tr>
<tr>
<td>REP LODS</td>
<td>11</td>
<td>15 + (2*c)</td>
<td>15 + (2*c)</td>
</tr>
<tr>
<td>REP SCAS</td>
<td>11</td>
<td>15 + (5/2*c)</td>
<td>15 + (5/2*c)</td>
</tr>
<tr>
<td>REP CMPS</td>
<td>11</td>
<td>16 + (10/3*c)</td>
<td>16 + (10/3*c)</td>
</tr>
</tbody>
</table>

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

**Guidelines for Repeated String Instructions**

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

**Use the Largest Possible Operand Size**

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

**Ensure DF=0 (UP)**

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

While string instructions with DF = 1 (DOWN) are slower, only the overhead part of the cycle equation is larger and not the throughput part. See Table 1 on page 123 for additional latency numbers.

**Align Source and Destination with Operand Size**

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

**Inline REP String with Low Counts**

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

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

Using MOVQ and MOVNTQ for Block Copy/Fill

To fill or copy blocks of data that are larger than 512 bytes, or where the destination is in uncacheable memory, use the MMX instructions MOVQ/MOVNTQ instead of REP STOS and REP MOVS in order to achieve maximum performance. (See the guideline, “Use MMX™ Instructions for Block Copies and Block Fills” on page 174.)

Use XOR Instruction to Clear Integer Registers

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

Example 1 (Acceptable):

```
MOV REG, 0
```

Example 2 (Preferred):

```
XOR REG, REG
```

Efficient 64-Bit Integer Arithmetic

This section contains a collection of code snippets and subroutines showing the efficient implementation of 64-bit arithmetic. Addition, subtraction, negation, and shifts are best handled by inline code. Multiplies, divides, and remainders are less common operations and should usually be implemented as subroutines. If these subroutines are used often, the programmer should consider inlining them. Except for division and remainder, the code presented works for both signed and unsigned integers. The division and remainder code shown works for unsigned integers, but can easily be extended to handle signed integers.
Example 1 (Addition):
; add operand in ECX:EBX to operand EDX:EAX, result in
; EDX:EAX
ADD EAX, EBX
ADC EDX, ECX

Example 2 (Subtraction):
; subtract operand in ECX:EBX from operand EDX:EAX, result in
; EDX:EAX
SUB EAX, EBX
SBB EDX, ECX

Example 3 (Negation):
; negate operand in EDX:EAX
NOT EDX
NEG EAX
SBB EDX, -1 ; fixup: increment hi-word if low-word was 0

Example 4 (Left shift):
; shift operand in EDX:EAX left, shift count in ECX (count
; applied modulo 64)
SHLD EDX, EAX, CL ; first apply shift count
SHL EAX, CL ; mod 32 to EDX:EAX
TEST ECX, 32 ; need to shift by another 32?
JZ $lshift_done ; no, done
MOV EDX, EAX ; left shift EDX:EAX
XOR EAX, EAX ; by 32 bits

$lshift_done:

Example 5 (Right shift):
SHRD EAX, EDX, CL ; first apply shift count
SHR EDX, CL ; mod 32 to EDX:EAX
TEST ECX, 32 ; need to shift by another 32?
JZ $rshift_done ; no, done
MOV EAX, EDX ; left shift EDX:EAX
XOR EDX, EDX ; by 32 bits

$rshift_done:
Example 6 (Multiplication):

\texttt{\_llmul} computes the low-order half of the product of its arguments, two 64-bit integers.

;INPUT: [ESP+8]:[ESP+4] multiplicand
; [ESP+16]:[ESP+12] multiplier

;OUTPUT: EDX:EAX \text{ (multiplicand \times multiplier)} \mod 2^{64}

;DESTROYS: EAX,ECX,EDX,EFlags

\texttt{\_llmul \text{ PROC}}
\texttt{MOV EDX, [ESP+8] ;multiplicand\_hi}
\texttt{MOV ECX, [ESP+16] ;multiplier\_hi}
\texttt{OR EDX, ECX ;one operand \geq 2^{32} ?}
\texttt{MOV EDX, [ESP+12] ;multiplier\_lo}
\texttt{MOV EAX, [ESP+4] ;multiplicand\_lo}
\texttt{JNZ \$twomul ;yes, need two multiplies}
\texttt{MUL EDX ;multiplicand\_lo \times multiplier\_lo}
\texttt{RET ;done, return to caller}

$twomul:
\texttt{IMUL EDX, [ESP+8] ;p3\_lo = multiplicand\_hi \times multiplier\_lo}
\texttt{IMUL ECX, EAX ;p2\_lo = multiplier\_hi \times multiplicand\_lo}
\texttt{ADD ECX, EDX ;p2\_lo + p3\_lo}
\texttt{MUL DWORD PTR [ESP+12] ;p1=multiplicand\_lo \times multiplier\_lo}
\texttt{ADD EDX, ECX ;p1+p2lo+p3\_lo = result in EDX:EAX}
\texttt{RET ;done, return to caller}

\texttt{\_llmul \text{ ENDP}}
Example 7 (Unsigned Division):

_uulldiv divides two unsigned 64-bit integers, and returns the quotient.

;INPUT: [ESP+8]:[ESP+4] dividend
;     [ESP+16]:[ESP+12] divisor
;
;OUTPUT: EDX:EAX quotient of division
;
;DESTROYS: EAX,ECX,EDX,EFlags

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

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

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

_ulldiv ENDP
Example 8 (Signed Division):

: _lldiv divides two signed 64-bit numbers and delivers the quotient.

: INPUT:     [ESP+8]:[ESP+4]  dividend
:            [ESP+16]:[ESP+12] divisor
:
: OUTPUT:    EDX:EAX  quotient of division
:
: DESTROYS:  EAX,ECX,EDX,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
Chapter 8 Integer Optimizations

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
SHR EDX, CL ; less than 2^32 (i.e. fits in EBX)
ROL EDI, 1 ; restore original divisor-hi
DIV EBX ; compute quotient
MOV EBX, [ESP] ; dividend-lo
MOV ECX, EAX ; save quotient
IMUL EDI, EAX ; quotient * divisor hi-word (low only)
MUL DWORD PTR [ESP+4] ; quotient * divisor lo-word
ADD EDX, EDI ; EDX:EAX = quotient * divisor
SUB EBX, EAX ; dividend-lo - (quotient*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 negative
XOR EDX, EDX ; clear hi-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
_1ldiv ENDP
Example 9 (Unsigned Remainder):

\_ullrem divides two unsigned 64-bit integers, and returns the remainder.

; INPUT: [ESP+8]:[ESP+4] dividend
; [ESP+16]:[ESP+12] divisor
; OUTPUT: EDX:EAX remainder of division
; DESTROYS: EAX,ECX,EDX,EFlags

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

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

$r_big_divisor:
PUSH EDI        ; save EDI as per calling convention
MOV  EDI, ECX   ; save divisor_hi
SHR  EDX, 1     ; shift both divisor and dividend right
RCR  EAX, 1     ; by 1 bit
ROR  EDI, 1
RCR  EBX, 1
BSR  ECX, ECX    ; ECX = number of remaining shifts
SHRD EBX, EDI, CL  ; scale down divisor and dividend such that divisor is less than 2^32
SHRD EAX, EDX, CL  ; (i.e. fits in EBX)
Chapter 8  Integer Optimizations

ROL EDI, 1
DIV EBX
MOV EBX, [ESP+12]
MOV ECX, EAX
IMUL EDI, EAX
MUL DWORD PTR [ESP+20]
ADD EDX, EDI
MOV ECX, [ESP+16]
MOV EAX, [ESP+20]
SBB ECX, EDX
SBB EDX, EDX
AND EAX, EDX
AND EDX, [ESP+24]
ADD EAX, EBX
ADC EDX, ECX
POP EDI
POP EBX
RET

_ullrem ENDP
**Example 10 (Signed Remainder):**

_\_llrem divides two signed 64-bit numbers and returns the remainder:

- INPUT:  [ESP+8]:[ESP+4]  dividend
  [ESP+16]:[ESP+12]  divisor

- OUTPUT:  EDX:EAX  remainder of division

- DESTROYS:  EAX,ECX,EDX,EFlags

```assembly
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  ;nope, need two divisions
DIV  EBX          ;EAX = quotient_lo
MOV  EAX, EDX     ;EAX = remainder_lo
MOV  EDX, ECX     ;DX = 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
SHR EDX, CL       ;less than 2^32 (i.e. fits in EBX)
ROL EDI, 1        ;restore original divisor_hi
DIV EBX           ;compute quotient
MOV EBX, [ESP]    ;dividend_lo
MOV ECX, EAX      ;save quotient
IMUL EDI, EAX     ;quotient * divisor hi-word (low only)
MUL DWORD PTR [ESP+4] ;quotient * divisor lo-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
Efficient Implementation of Population Count Function

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 following example code 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.

For an efficient population count function operating on 64-bit integers, see “Efficient 64-Bit Population Count Using MMX™ Instructions” on page 184.

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 algorithms consist of the following steps:

**Step 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 $v \cdot (v >> 1)$. 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:

$$w = v \cdot ((v >> 1) \& 0x55555555)$$

**Step 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 value 0000b, 0001b, 0010b, 0011b, or 0100b.
**Step 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. In a symbolic form:

\[ x = 0aaa0bbb0ccc0ddd0eee0fff0ggg0hhh \]
\[ x >> 4 = 00000aaa0bbb0ccc0ddd0eee0fff0ggg \]
\[ sum = 0aaaWWWWiiiiXXXXjjjjYYYYkkkkZZZZ \]

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

**Step 4**

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

\[
\begin{array}{l}
0p0q0r0s \times 01010101 = \\
\hline
0p0q0r0s \\
0p0q0r0s \\
0p0q0r0s \\
0p0q0r0s \\
000pxxww:vvuutt0s
\end{array}
\]

Here \( p, q, r, \) and \( s \) are the 4-bit sums from the previous step, and \( vv \) is the final result in which we are interested. Thus, the final result:

\[ z = (y \times 0x01010101) >> 24 \]

**Example 1 (Integer Version):**

```c
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)
        ;
        POP EAX
    }
    return x;
}
```
MOV EAX, EDX ; 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);
}

**MMX Version**

The following code sample is an MMX version of popcount() that works on 64 bits at a time. This MMX code can do popcounts about twice as fast as the integer version (for an identical number of bits). Notice that the source was loaded using two instructions instead of a simple MOVQ to avoid a bad STLFL case (size mismatch from two DWORDs feeding into a QWORD).

Example 1 (MMX version):

```c
#include "amd3d.h"

__declspec (naked) unsigned int __stdcall popcount64_1 (unsigned __int64 v)
{
static const __int64 C55 = 0x5555555555555555;
static const __int64 C33 = 0x3333333333333333;
static const __int64 C0F = 0x0F0F0F0F0F0F0F0F;

__asm {
MOVD      MM0, [ESP+4] ; v_low
PUNPCKLDQ MM0, [ESP+8] ; v
MOVO      MM1, MM0 ; v
PSRLD     MM0, 1 ; v >> 1
PAND      MM0, [C55] ; (v >> 1) & 0x55555555
PSUBD     MM1, MM0 ; w = v - ((v >> 1) & 0x55555555)
MOVO      MM0, MM1 ; w
PSRLD     MM1, 2 ; w >> 2
PAND      MM0, [C33] ; w & 0x33333333
PAND      MM1, [C33] ; (w >> 2) & 0x33333333
PADDD     MM0, MM1 ; x = (w & 0x33333333) + ((w >> 2) & 0x33333333)
MOVO      MM1, MM0 ; x
PSRLD     MM0, 4 ; x >> 4
PADDD     MM0, MM1 ; x + (x >> 4)
PAND      MM0, [C0F] ; y = (x + (x >> 4) & 0x0F0F0F0F)
PXOR      MM1, MM1 ; 0
PSADDBW   (MM0, MM1) ; sum across all 8 bytes
MOVD      EAX, MM0 ; result in EAX per calling
```
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 processors. The code is written for the Microsoft Visual C compiler.

Function uint_to_ascii_lz() converts like sprintf (sptr, "%010u", x), i.e., leading zeros are retained, whereas function uint_to_ascii_nlz() converts like sprintf (sptr, "%u", x), i.e., 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 Example 3 in “Avoid Branches Dependent on Random Data” on page 93 before starting the conversion process. For restricted argument range, more efficient conversion routines can be constructed using the same algorithm as is 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 “Replace Divides with Multiplies” on page 115. Each block is then converted into a fixed-point format that consists of one (decimal) integer digit and a binary fraction. This allows 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 mispredicts, 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.

Example 1 (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                ; 10000 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 EAX, [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], C L      ; store 1st digit out to 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 out to 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, 0x0003fff    ; fraction part
    OR ECX, '0'            ; convert 2nd digit to ASCII
    MOV [EDI+10], C L     ; store 2nd digit out to memory
    LEA ECX, [EBX+EBX*4]   ; 5*fraction, new digit ECX<31:13>
    LEA EBX, [EBX+EBX*4]   ; 5*fraction, new fraction EBX<12:0>
SHR    EDX, 14            ; 7th digit
AND    ESI, 0x000003fff    ; fraction part
OR     EDX, '0'           ; convert 7th digit to ASCII
MOV    [EDI+6], DL        ; store 7th digit out to 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 out to 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 out to 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 out to 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 out to 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 out to memory
SHR    EDX, 11            ; 10th digit
OR     EDX, '0'           ; convert 10th digit to ASCII
MOV    [EDI+9], dx        ; store 10th digit and end marker to 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
Example 2 (Binary to ASCII decimal conversion suppressing leading zeros):

```asm
__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           ;init 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 non-zero 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
        MOV    EAX, EDX           ; point 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 out to 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 non-zero digit yet ?
        SBB    EDI, -1            ;yes->increment ptr, no->keep old ptr
        SHR    EAX, 27            ;next digit
        AND    EDX, 07fffffffh    ;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<31:26>
        LEA    EDX, [EDX*4+EDX]   ;5*fraction, new fraction EDX<25:0>
        CMP    EBX, 1             ;any non-zero digit yet ?
        SBB    EDI, -1            ;yes->increment ptr, no->keep old ptr
        SHR    EAX, 26            ;next digit
        AND    EDX, 03fffffffh    ;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<31:25>
LEA    EDX, [EDX*4+EDX]   ;5*fraction, new fraction EDX<24:0>
CMP    EBX, 1             ;any non-zero 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 out to 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 non-zero 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 out to memory
LEA    EAX, [EDX*4+EDX]   ;5*fraction, new digit EAX<31:23>
LEA    EDX, [EDX*4+EDX]   ;5*fraction, new fraction EDX<22:0>
CMP    EBX, 1             ;any non-zero 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<21:0>
CMP    EBX, 1             ;any non-zero 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 out to memory
LEA    EAX, [EDX*4+EDX]   ;5*fraction, new digit EAX<31:21>
LEA    EDX, [EDX*4+EDX]   ;5*fraction, new fraction EDX<20:0>
CMP    EBX, 1             ;any non-zero digit yet ?
SBB    EDI, -1            ;yes->increment ptr, no->keep old ptr
SHR    EAX, 21            ;next digit
AND    EDX, 001ffffffh     ;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<31:20>
CMP    EBX, 1             ;any non-zero 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 out to memory
Derivation of Multiplier Used for Integer Division by Constants

Derivation of Algorithm, Multiplier, and Shift Factor for Unsigned Integer Division

The utility udiv.exe was compiled using the code shown in this section. The executable and source code are located in the opt_utilities directory of the AMD Documentation CDROM and the SDK. The program is provided “as is.”

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

Example Code

```c
/* Program to determine algorithm, multiplier, and shift factor 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 ("
");
printf ("SHR dividend, %d
", n);
printf ("
");
printf ("; quotient replaced dividend\n");
}
go to printed_code;
}

l = log2(t) + 1;
j = (((U64)(0xffffffff)) % ((U64)(t)));
k = ((((U64)(1)) << (32+l)) / ((U64)(0xffffffff-j)));
m_low = ((((U64)(1)) << (32+l)) / (U64)(t));
m_high = ((((U64)(1)) << (32+l)) + k) / (U64)(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, 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");
    printf ("SHR    EDX, %d\n", s);
    printf ("\n");
    printf ("; quotient now in EDX\n");
}

printed_code:

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

return(0);
Derivation of Algorithm, Multiplier, and Shift Factor for Signed Integer Division

The utility sdiv.exe was compiled using the following code. The executable and source code are located in the opt_utilities directory of the AMD Documentation CDROM and the SDK. The program is provided “as is.”

Example Code

/* Program to determine algorithm, multiplier, and shift factor 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, 0800000000h\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
*/

l = log2(d);
j = (((U64)(0x80000000)) % ((U64)(d)));
k = (((U64)(1)) << (32+1)) / (((U64)(0x80000000)-j));
m_low = (((U64)(1)) << (32+1)) / d;
m_high = (((U64)(1)) << (32+1)) + 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);
}
This chapter details the methods used to optimize floating-point code to the pipelined floating-point unit (FPU). Guidelines are listed in order of importance.

Ensure All FPU Data is Aligned

As described in “Memory Size and Alignment Issues” on page 63, align floating-point data naturally. That is, align words on word boundaries, doublewords on doubleword boundaries, and quadwords on quadword boundaries. Misaligned memory accesses reduce the available memory bandwidth.

Use Multiplies Rather than Divides

If accuracy requirements allow, convert floating-point division by a constant to a multiply by the reciprocal. Divisors that are powers of two and their reciprocals are exactly representable, and therefore do not cause an accuracy issue, except for the rare cases in which the reciprocal overflows or underflows. Unless such an overflow or underflow occurs, always convert a division by a power of two to a multiply. Although the AMD Athlon™ processor has high-performance division, multiplies are significantly faster than divides.
Use FFREEP Macro to Pop One Register from the FPU Stack

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

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

On the AMD Athlon processor, a faster alternative is to use the FFREEP instruction below. Note that the FFREEP instruction, although insufficiently documented in the past, is supported by all 32-bit x86 processors. The opcode bytes for FFREEP ST(i) are listed in Table 22 on page 292.

```
FFREEP ST(0) ;removes one register from stack
FFREEP ST(i) works like FFREE ST(i) except that it increments the FPU top-of-stack after doing the FFREE work. In other words, FFREEP ST(i) marks ST(i) as empty, then increments the x87 stack pointer. On the AMD Athlon processor, the FFREEP instruction converts to an internal NOP, which can go down any pipe with no dependencies.

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

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

To free up all remaining occupied FPU stack register and set the x87 stack pointer to zero, use the FEMMS or EMMS instruction instead of a series of FFREEP ST(0) instructions. This promotes code density and preserves decode and execution bandwidth. Note that use of FEMMS/EMMS in this fashion is not recommended for AMD-K6 family processors.
Floating-Point Compare Instructions

For branches that are dependent on floating-point comparisons, use the following instructions:

- FCOMI
- FCOMIP
- FUCOMI
- FUCOMIP

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

Use the FXCH Instruction Rather than FST/FLD Pairs

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

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

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

Minimize Floating-Point-to-Integer Conversions

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

Example 1 (Fast):

```
FLD    QWORD PTR [X] ;load double to be converted
FSTCW  [SAVE_CW] ;save current FPU control word
MOVZX  EAX, WORD PTR[SAVE_CW];retrieve control word
OR     EAX, OCOOh ;rounding control field = truncate
MOV    WORD PTR [NEW_CW], AX ;new FPU control word
FLDCW  [NEW_CW] ;load new FPU control word
FISTP  DWORD PTR [I] ;do double->int conversion
FLDCW  [SAVE_CW] ;restore original control word
```

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

According to the recommendations for inlining (see “Always Inline Functions with Fewer than 25 Machine Instructions” on page 110), the above code should not be put into a separate subroutine (e.g., ftol). It should rather be inlined into the main code.

In some codes, floating-point numbers are converted to an integer and the result is immediately converted back to floating-point. In such cases, use the FRNDINT instruction for maximum performance instead of FISTP in the code above. FRNDINT delivers the integral result directly to a FPU register.
in floating-point form, which is faster than first using FISTP to store the integer result and then converting it back to floating-point with FILD.

If there are multiple, consecutive floating-point-to-integer conversions, the cost of FLDCW operations should be minimized by saving the current FPU control word, forcing the FPU into truncating mode, and performing all of the conversions before restoring the original control word.

The speed of the code in Example 1 is somewhat dependent on the nature of the code surrounding it. For applications in which the speed of floating-point-to-integer conversions is extremely critical for application performance, experiment with either of the following substitutions, which may or may not be faster than the code above.

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

Example 2 (Potentially faster)

```assembly
FLD QWORD PTR [X] ;load double to be converted
FST DWORD PTR [TX] ;store X because sign(X) is needed
FIST DWORD PTR [I] ;store rndint(x) as default result
FISUB DWORD PTR [I] ;compute DIFF = X - rndint(X)
FSTP DWORD PTR [DIFF] ;store DIFF as we need sign(DIFF)
MOV EAX, [TX] ;X
MOV EDX, [DIFF] ;DIFF
TEST EDX, EDX ;DIFF == 0 ?
JZ $DONE ;default result is OK, done
XOR EDX, EAX ;need correction if sign(X) != sign(DIFF)
SAR EAX, 31 ;(X<0) ? 0x00000001 : 0
SAR EDX, 31 ; sign(X)!=sign(DIFF)?0x00000001:0
LEA EAX, [EAX+EAX+1] ;(X<0) ? 0x00000001 : 1
AND EDX, EAX ;correction: -1, 0, 1
SUB [I], EDX ;trunc(X)=rndint(X)-correction
$DONE:
```

The second substitution simulates a truncating floating-point to integer conversion using only integer instructions and therefore works correctly independent of the FPU’s current rounding mode. It does not handle NaNs, infinities, and overflows according to the IEEE-754 standard. Note that the first
instruction of this code may cause an STLF size mismatch resulting in performance degradation if the variable to be converted has been stored recently.

**Example 3 (Potentially faster):**

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

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

**Example 4 (Fastest):**

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

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

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

The transcendental instructions FSIN, FCOS, FPTAN, and FSINCOS are architecturally restricted in their argument range. Only arguments with a magnitude of \( \leq 2^{63} \) can be evaluated. If the argument is out of range, the C2 bit in the FPU status word is set, and the argument is returned as the result. Software needs to guard against such (extremely infrequent) cases.

If an “argument out of range” is detected, a range reduction subroutine is invoked which reduces the argument to less than \( 2^{63} \) before the instruction is attempted again. While an argument > \( 2^{63} \) is unusual, it often indicates a problem elsewhere in the code and the code may completely fail in the absence of a properly guarded trigonometric instruction. For example, in the case of FSIN or FCOS generated from a sin() or cos() function invocation in the high-level language, the downstream code might reasonably expect that the returned result is in the range \([-1,1]\).

A overly simple solution for guarding a trigonometric instruction may check the C2 bit in the FPU status word after each FSIN, FCOS, FPTAN, and FSINCOS instruction, and take appropriate action if it is set (indicating an argument out of range).

**Example 1 (Avoid):**

```
FLD    QWORD PTR [x]   ;argument
FSIN   ;compute sine
FSTSW AX ;store FPU status word to AX
TEST AX, 0400h ;is the C2 bit set?
JZ    $in_range ;no, argument was in range, all OK
CALL $reduce_range ;reduce argument in ST(0) to < 2^63
FSIN   ;compute sine (in-range argument: guaranteed)
$in_range:
```

Such a solution is inefficient since the FSTSW instruction is serializing with respect to all x87\(^\text{TM}\)/3DNow!\(^\text{TM}/\)MMX\(^\text{TM}\) instructions and should thus be avoided (see the section “Floating-Point Compare Instructions” on page 153). Use of
FSTSW in the above fashion slows down the common path through the code.

Instead, it is advisable to check the argument before one of the trigonometric instructions is invoked.

**Example 2 (Preferred):**

```
FLD  QWORD PTR [x]          ;argument
FLD  DWORD PTR [two_to_the_63]  ;2^63
FCOMIP ST,ST(1)               ;argument <= 2^63 ?
JBE  $in_range               ;Yes. It is in range.
CALL $reduce_range ;reduce argument in ST(0) to < 2^63
$in_range:
FSIN                          ;compute sine (in-range argument ; guaranteed)
```

Since out-of-range arguments are extremely uncommon, the conditional branch will be perfectly predicted, and the other instructions used to guard the trigonometric instruction can execute in parallel to it.
Take Advantage of the FSINCOS Instruction

Frequently, a piece of code that needs to compute the sine of an argument also needs to compute the cosine of that same argument. In such cases, use the FSINCOS instruction to compute both trigonometric functions concurrently, which is faster than using separate FSIN and FCOS instructions to accomplish the same task.

Example 1 (Avoid):

```
FLD    QWORD PTR [x]
FLD    DWORD PTR [two_to_the_63]
FCOMIP ST,ST(1)
JBE    $in_range
CALL   $reduce_range
$in_range:
FLD    ST(0)
FCOS
FSTP   QWORD PTR [cosine_x]
FSIN
FSTP   QWORD PTR [sine_x]
```

Example 1 (Preferred):

```
FLD    QWORD PTR [x]
FLD    DWORD PTR [two_to_the_63]
FCOMIP ST,ST(1)
JBE    $in_range
CALL   $reduce_range
$in_range:
FSINCOS
FSTP   QWORD PTR [cosine_x]
FSTP   QWORD PTR [sine_x]
```
Chapter 10 3DNow!™ and MMX™ Optimizations

This chapter describes 3DNow! and MMX code optimization techniques for the AMD Athlon™ processor. Guidelines are listed in order of importance. 3DNow! porting guidelines can be found in the 3DNow!™ Instruction Porting Guide, order no. 22621.

Use 3DNow!™ Instructions

When single precision is required, perform floating-point computations using the 3DNow! instructions instead of x87 instructions. The SIMD nature of 3DNow! achieves twice the number of FLOPs that are achieved through x87 instructions. 3DNow! instructions provide for a flat register file instead of the stack-based approach of x87 instructions.

See the 3DNow!™ Technology Manual, order no. 21928, for information on instruction usage.
Use FEMMS Instruction

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

3DNow! and MMX instructions are designed to be used concurrently with no switching issues. Likewise, enhanced 3DNow! instructions can be used simultaneously with MMX instructions. However, x87 and 3DNow! instructions share the same architectural registers so there is no easy way to use them concurrently without cleaning up the register file in between using FEMMS/EMMS.

Use 3DNow!™ Instructions for Fast Division

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

Optimized 14-Bit Precision Divide

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

Example 1:

\[
\begin{align*}
\text{MOVQ} & \quad \text{MM0. [MEM]} : \quad 0 \mid W \\
\text{PFRCP} & \quad \text{MM0. MM0} : \quad 1/W \mid 1/W \quad \text{(approximate)} \\
\text{PFRCP} & \quad \text{MM2. [MEM]} : \quad Y \mid X \\
\text{PFMUL} & \quad \text{MM2. MM0} : \quad Y/W \mid X/W
\end{align*}
\]
Optimized Full 24-Bit Precision Divide

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

Example 2:
```
MOVD MM0, [W] ; 0 | W
PFRCP MM1, MM0 ; 1/W | 1/W (approximate)
PUNPCKLDQ MM0, MM0 ; W | W (MMX instr.)
PFRCPIT1 MM0, MM1 ; 1/W | 1/W (refine)
MOVQ MM2, [X_Y] ; Y | X
PFRCPIT2 MM0, MM1 ; 1/W | 1/W (final)
PFMUL MM2, MM0 ; Y/W | X/W
```

Pipelined Pair of 24-Bit Precision Divides

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

Example 3:
```
MOVQ MM0, [DIVISORS] ; y | x
PFRCP MM1, MM0 ; 1/x | 1/x (approximate)
MOVQ MM2, MM0 ; y | x
PUNPCKHDQ MM0, MM0 ; y | y
PFRCP MM0, MM0 ; 1/y | 1/y (approximate)
PUNPCKLDQ MM1, MM0 ; 1/y | 1/x (approximate)
MOVQ MM0, [DIVIDENDS] ; z | w
PFRCPIT1 MM2, MM1 ; 1/y | 1/x (intermediate)
PFRCPIT2 MM2, MM1 ; 1/y | 1/x (final)
PFMUL MM0, MM2 ; z/y | w/x
```
Newton-Raphson Reciprocal

Consider the quotient \( q = \frac{a}{b} \). An (on-chip) ROM-based table lookup can be used to quickly produce a 14-to-15-bit precision approximation of \( \frac{1}{b} \) using just one 3-cycle latency PFRCP instruction. A full 24-bit precision reciprocal can then be quickly computed from this approximation using a Newton-Raphson algorithm.

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

\[
Z_{i+1} = Z_i \cdot (2 - b \cdot Z_i)
\]

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

\[
\begin{align*}
X_0 & = \text{PFRCP}(b) \\
X_1 & = \text{PFRCPIT1}(b,X_0) \\
X_2 & = \text{PFRCPIT2}(X_1,X_0) \\
q & = \text{PFMUL}(a,X_2)
\end{align*}
\]

The 24-bit final reciprocal value is \( X_2 \). In the AMD Athlon processor 3DNow! technology implementation the operand \( X_2 \) contains the correct round-to-nearest single precision reciprocal for approximately 99% of all arguments.
Use 3DNow!™ Instructions for Fast Square Root and Reciprocal Square Root

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

Optimized 15-Bit Precision Square Root

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

Example 1:

\[
\begin{align*}
\text{MOV}D &\quad \text{MM0, [MEM]} \quad : \quad 0 \quad | \quad a \\
\text{PFRSQRT} &\quad \text{MM1, MM0} \quad : \quad 1/\sqrt{a} \quad | \quad 1/\sqrt{a} \quad (\text{approximate}) \\
\text{PUNPCKLDQ} &\quad \text{MM0, MM0} \quad : \quad a \quad | \quad a \quad (\text{MMX instr.}) \\
\text{PFMUL} &\quad \text{MM0, MM1} \quad : \quad \sqrt{a} \quad | \quad \sqrt{a}
\end{align*}
\]

Optimized 24-Bit Precision Square Root

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

Example 2:

\[
\begin{align*}
\text{MOV}D &\quad \text{MM0, [MEM]} \quad : \quad 0 \quad | \quad a \\
\text{PFRSQRT} &\quad \text{MM1, MM0} \quad : \quad 1/\sqrt{a} \quad | \quad 1/\sqrt{a} \quad (\text{approx.}) \\
\text{MOVQ} &\quad \text{MM2, MM1} \quad : \quad X_0 = 1/(\sqrt{a}) \quad (\text{approx.}) \\
\text{PFMUL} &\quad \text{MM1, MM1} \quad : \quad X_0 \times X_0 \quad | \quad X_0 \times X_0 \quad (\text{step 1}) \\
\text{PUNPCKLDQ} &\quad \text{MM0, MM0} \quad : \quad a \quad | \quad a \quad (\text{MMX instr}) \\
\text{PFRSQIT} &\quad \text{MM1, MM0} \quad : \quad (\text{intermediate}) \quad (\text{step 2}) \\
\text{PFRCPI} &\quad \text{MM1, MM2} \quad : \quad 1/\sqrt{a} \quad | \quad 1/\sqrt{a} \quad (\text{step 3}) \\
\text{PFMUL} &\quad \text{MM0, MM1} \quad : \quad \sqrt{a} \quad | \quad \sqrt{a}
\end{align*}
\]
Newton-Raphson Reciprocal Square Root

The general Newton-Raphson reciprocal square root recurrence is:

\[ Z_{i+1} = 1/2 \cdot Z_i \cdot (3 - b \cdot Z_i^2) \]

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

\[
X_0 = \text{PFRSQRT}(b) \\
X_1 = \text{PFMUL}(X_0, X_0) \\
X_2 = \text{PFRSQIT1}(b, X_1) \\
X_3 = \text{PFRCPI2}(X_2, X_0) \\
X_4 = \text{PFMUL}(b, X_3)
\]

The 24-bit final reciprocal square root value is \( X_3 \). In the AMD Athlon processor 3DNow! implementation, the estimate contains the correct round-to-nearest value for approximately 87% of all arguments. The remaining arguments differ from the correct round-to-nearest value by one unit-in-the-last-place. The square root (\( X_4 \)) is formed in the last step by multiplying by the input operand \( b \).
Use MMX™ PMADDWD Instruction to Perform Two 32-Bit Multiplies in Parallel

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

Example 1:

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

Use PMULHUW to Compute Upper Half of Unsigned Products

The PMULHUW is an MMX extension that computes the upper 16 bits of four unsigned 16x16→32 products. The previously available MMX PMULHW instruction can be used to compute the upper 16 bits of four signed 16x16→32 products. Note that PMULLW can be used to compute the lower 16 bits of four 16x16→32 bit products regardless of whether the multiplication is signed or unsigned.

Without PMULHUW, it is actually quite difficult to perform unsigned multiplies using MMX instructions. Example 2 shows how this can be accomplished if this is required in blended code that needs to run well on both the AMD Athlon processor and AMD-K6 family processors. A restriction of the replacement code is that all words of the multiplicand must be in range 0...0x7FFF, a condition that is frequently met.
The replacement code uses the following algorithm. Let A be the unsigned multiplicand in range 0...0x7FFF, and let B be the unsigned multiplier in range 0...0xFFFF. The unsigned multiplication A*B can be accomplished as follows when only signed multiplication, denoted by @, is available.

If B is in range 0...0x7FFF, A*B = A @ B. However, if B is in range 0x8000...0xFFFF, then B is interpreted as a signed operand with value B'=B-2^16. Thus A @ B = A*B - 2^16*A, or A*B = A @ B + 2^16*A. Given that PMULLW computes the lower 16 bits of the result, only the upper 16 bits of the result, R = (A*B >> 16), needs to be found. Thus R = PMULHW(A,B) if B in 0...0x7FFF, and R = A + PMULHW(A,B) if B in 0x8000...0xFFFF. This means that the next step is to conditionally add A to the output of PMULHW if bit 15 of B is set, i.e., if B is negative under a signed interpretation.

**Example 1:**

| IN:  | MM0 = A3 A2 | A1 A0 | \(A_i\) are unsigned words |
| MM1 = B3 B2 | B2 B1 | \(B_i\) are unsigned words |
| OUT: | MM0 = A1*B1 | A0*B0 | unsigned DWORD results |
| MM2 = A3*B3 | A2*B2 | unsigned DWORD results |

MOVQ MM2, MM0 ; \(A_i, \text{ } i = \{0..4\}\)
PMULLW MM0, MM1 ; \((A_i*B_i)_{15:0}, \text{ } i = \{0..4\}\)
PMULHUW MM1, MM2 ; \((A_i*B_i)_{31:16}, \text{ } i = \{0..4\}\)
MOVQ MM2, MM0 ; \((A_i*B_i)_{15:0}, \text{ } i = \{0..4\}\)
PUNPCKLWD MM0, MM1 ; \((A1*B1)_{31:0} \mid (A0*B0)_{31:0}\)
PUNPCKHWD MM2, MM1 ; \((A3*B3)_{31:0} \mid (A2*B2)_{31:0}\)

**Example 2:**

| IN:  | MM0 = A3 A2 | A1 A0 | \(A_i\) are unsigned words <= 0x7FFF |
| MM1 = B3 B2 | B2 B1 | \(B_i\) are unsigned words |
| OUT: | MM0 = A1*B1 | A0*B0 | unsigned DWORD results |
| MM2 = A3*B3 | A2*B2 | unsigned DWORD results |

MOVQ MM2, MM0 ; \(A_i, \text{ } i = \{0..4\}\)
PMULLW MM0, MM1 ; \((A_i*B_i)_{15:0}, \text{ } i = \{0..4\}\)
MOVQ MM3, MM1 ; \(B_i, \text{ } i = \{0..4\}\)
PSRAW MM3, 15 ; \(M_i = B_i < 0 \? 0xffff : 0, \text{ } i = \{0..4\}\)
PAND MM3, MM2 ; \(M_i = B_i < 0 \? A_i : 0, \text{ } i = \{0..4\}\)
PMULHW MM1, MM2 ; \((A_i@B_i)_{31:16}, \text{ } i = \{0..4\}\)
PADDW MM1, MM3 ; \((A_i@B_i)_{31:16} = (A_i < 0) \? (A_i@B_i)_{31:16} + A_i : (A_i@B_i)_{31:16}\)
MOVQ MM2, MM0 ; \((A_i@B_i)_{15:0}, \text{ } i = \{0..4\}\)
PUNPCKLWD MM0, MM1 ; \((A1*B1)_{31:0} \mid (A0*B0)_{31:0}\)
PUNPCKHWD MM2, MM1 ; \((A3*B3)_{31:0} \mid (A2*B2)_{31:0}\)
3DNow!™ and MMX™ Intra-Operand Swapping

If the swapping of MMX register halves is necessary, use the PSWAPD instruction, which is a new AMD Athlon 3DNow! DSP extension. Use this instruction only for AMD Athlon processor-specific code. “PSWAPD MMreg1, MMreg2” performs the following operation:

\[
\begin{align*}
\text{temp} &= \text{mmreg2} \\
\text{mmreg1}[63:32] &= \text{temp}[31:0] \\
\text{mmreg1}[31:0] &= \text{temp}[63:32]
\end{align*}
\]

See the *AMD Extensions to the 3DNow!™ and MMX™ Instruction Set Manual*, order no. 22466, for more usage information.

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

**Example 1 (Preferred, faster):**

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

**Example 2 (Preferred, fast):**

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

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

In many applications there is a need to quickly convert data consisting of packed 16-bit signed integers into floating-point numbers. The following two examples show how this can be accomplished efficiently on AMD processors.

The first example shows how to do the conversion on a processor that supports AMD's 3DNow! extensions, such as the AMD Athlon processor. It demonstrates the increased efficiency from using the PI2FW instruction. Use of this instruction should only be for AMD Athlon processor specific code. See the AMD Extensions to the 3DNow!™ and MMX™ Instruction Set Manual, order no. 22466 for more information on this instruction.

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

**Example 1 (AMD Athlon processor specific code using 3DNow! DSP extension):**

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

**Example 2 (AMD-K6 and AMD Athlon processor blended code):**

```
MOVD      MM1, [packed_signed_word]    ; 0 0 | b a
PXOR      MM0, MM0                     ; 0 0 | 0 0
PUNPCKLWD MM0, MM1                     ; b 0 | a 0
PSRAD     MM0, 16          ; sign extend: b | a
PI2FD     MM0, MM0             ;xb=float(b) | xa=float(a)
MOVQ      [packed_float], MM0  ;    store xb | xa
```
Width of Memory Access Differs Between PUNPCKL* and PUNPCKH*

The width of the memory access performed by the load-execute forms of PUNPCKLBW, PUNPCKLWD, and PUNPCKLDQ is 32 bits (a DWORD), while the width of the memory access of the load-execute forms of PUNPCKHBW, PUNPCKHWD, and PUNPCKHDQ is 64 bits (a QWORD).

This means that the alignment requirements for memory operands of PUNPCKL* instructions (DWORD alignment) are less strict than the alignment requirements for memory operands of PUNPCKH* instructions (QWORD alignment). Code can take advantage of this in order to reduce the number of misaligned loads in a program. A second advantage of using PUNPCKL* instead of PUNPCKH* is that it helps avoid size mismatches during load-to-store forwarding. Store data from either a DWORD store or the lower DWORD of a QWORD store can be bypassed inside the load/store buffer to PUNPCKL*, but only store data from a QWORD store can be bypassed to PUNPCKH*.

Example 1 (Avoid):

```
MOV [foo], EAX : a       // DWORD aligned store
MOV [foo+4], EDX : b     // DWORD aligned store
PUNPCKHDQ MM0, [foo-4] ; a | <junk> // STL size mismatch,
                         // potentially misaligned
PUNPCKHDQ MM0, [foo] ; b | a   // STL size mismatch,
                         // potentially misaligned
```

Example 2 (Preferred):

```
MOV [foo], EAX : a       // DWORD aligned store
MOV [foo+4], EDX : b     // DWORD aligned store
MOVD MM0, [foo] : 0 | a   // DWORD aligned load,
                         // STL size match
PUNPCKLDQ MM0, [foo+4] : b | a   // DWORD aligned load,
                         // STL size match
```
Use MMX™ PXOR to Negate 3DNow!™ Data

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

Use the following code to negate 3DNow! data:

```
msgn  DQ 8000000080000000h
PXOR  MM0, [msgn] ;toggle sign bit
```
**Use MMX™ PCMP Instead of 3DNow!™ PFCMP**

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

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

<table>
<thead>
<tr>
<th>Both Numbers Positive</th>
<th>If both arguments are positive, PCMP always works.</th>
</tr>
</thead>
<tbody>
<tr>
<td>One Negative, One Positive</td>
<td>If one number is negative and the other is positive, PCMP still works, except when one number is a positive zero and the other is a negative zero.</td>
</tr>
<tr>
<td>Both Numbers Negative</td>
<td>Be careful when performing integer comparison using PCMPGT on two negative 3DNow! numbers. The result is the inverse of the PFCMPGT floating-point comparison. For example:</td>
</tr>
<tr>
<td></td>
<td>(-2 = 84000000)</td>
</tr>
<tr>
<td></td>
<td>(-4 = 84800000)</td>
</tr>
<tr>
<td></td>
<td>PCMPGT gives 84800000 &gt; 84000000, but (-4 &lt; -2). To address this issue, simply reverse the comparison by swapping the source operands.</td>
</tr>
</tbody>
</table>
Use MMX™ Instructions for Block Copies and Block Fills

For moving or filling small blocks of data of less than 512 bytes between cacheable memory areas, the REP MOVS and REP STOS families of instructions deliver good performance and are straightforward to use. For moving and filling larger blocks of data, or to move/fill blocks of data where the destination is in non-cacheable space, it is recommended to make use of MMX instructions and extended MMX instructions. The following examples demonstrate how to copy any number of DWORDs between a DWORD aligned source and a DWORD aligned destination, and how to fill any number of DWORDs at a DWORD aligned destination.

The following example code is written for the inline assembler of Microsoft Visual C, and uses instruction macros defined in the file AMD3DX.H from the AMD Athlon Processor SDK. It is suitable for moving/filling a DWORD aligned block of data in the following situations:

- Blended code, i.e., code that needs to perform well on both the AMD Athlon processor and AMD-K6 family processors, operating on a data block of more than 512 bytes
- AMD Athlon processor-specific code where the destination is in cacheable memory, the data block is smaller than 8 Kbytes, and immediate data re-use of the data at the destination is expected
- AMD-K6 processor-specific code where the destination is in non-cacheable memory and the data block is larger than 64 bytes.
Example 1:

```c
#include "amd3dx.h"

// block copy: copy a number of DWORDs from DWORD aligned source
// to DWORD aligned destination using cacheable stores.

_asm |
    MOV     ESI, [src_ptr] ; pointer to src, DWORD aligned
    MOV     EDI, [dst_ptr] ; pointer to dst, DWORD aligned
    MOV     ECX, [blk_size] ; number of DWORDs to copy

    PREFETCH (ESI) ; prefetch first src cache line
    CMP     ECX, 1 ; less than one DWORD to copy ?
    JB      $copydone2_cc ; yes, must be no DWORDs to copy, done
    TEST    EDI, 7 ; dst QWORD aligned?
    JZ      $dstqaligned2_cc ; yes

    MOVD    MM0, [ESI] ; read one DWORD from src
    MOVD    [EDI], MM0 ; store one DWORD to dst
    ADD     ESI, 4 ; src++
    ADD     EDI, 4 ; dst++
    DEC     ECX ; number of DWORDs to copy

$dstqaligned2_cc:
    MOV     EBX, ECX ; number of DWORDs to copy
    SHR     ECX, 4 ; number of cache lines to copy
    JZ      $copyqwords2_cc ; no whole cache lines to copy, maybe QWORDS

    prefetchm (ESI, 64) ; prefetch src cache line one ahead
    prefetchmlong (ESI, 128) ; prefetch src cache line two ahead
    ALIGN   16 ; align loop for optimal performance

$cloop2_cc:
    prefetchmlong (ESI, 192) ; prefetch cache line three ahead
    MOVQ    MM0, [ESI] ; load first QWORD in cache line from src
    ADD     EDI, 64 ; src++
    MOVQ    MM1, [EDI+8] ; load second QWORD in cache line from src
    ADD     ESI, 64 ; dst++
    MOVQ    MM2, [ESI+48] ; load third QWORD in cache line from src
    MOVQ    [EDI-64], MM0 ; store first DWORD in cache line to dst
    MOVQ    MM0, [ESI-40] ; load fourth QWORD in cache line from src
    MOVQ    [EDI-56], MM1 ; store second DWORD in cache line to dst
    MOVQ    MM1, [ESI-32] ; load fifth QWORD in cache line from src
    MOVQ    [EDI-48], MM2 ; store third DWORD in cache line to dst
    MOVQ    MM2, [ESI-24] ; load sixth QWORD in cache line from src
    MOVQ    [EDI-40], MM0 ; store fourth DWORD in cache line to dst
    MOVQ    MM0, [ESI-16] ; load seventh QWORD in cache line from src
    MOVQ    [EDI-32], MM1 ; store fifth DWORD in cache line to dst
    MOVQ    MM1, [ESI-8] ; load eight QWORD in cache line from src
    MOVQ    [EDI-24], MM2 ; store sixth DWORD in cache line to dst
    MOVQ    [EDI-16], MM0 ; store seventh DWORD in cache line to dst
    DEC     ECX ; count--
    JNZ     $cloop2_cc ; until no more cache lines to copy
```

$copyqwords2_cc:
    MOV       ECX, EBX         ;number of DWORDs to copy
    AND       EBX, 0xE         ;number of QWORDS left to copy * 2
    JZ        $copydword2_cc   ;no QWORDS left, maybe DWORD left
    ALIGN     16               ;align loop for optimal performance

$qloop2_cc:
    MOVQ     MM0, [ESI]        ;read QWORD from src
    MOVQ     [EDI], MM0        ;store QWORD to dst
    ADD      ESI, 8            ;src++
    ADD      EDI, 8            ;dst++
    SUB      EBX, 2            ;count--
    JNZ      $qloop2_cc        ;until no more QWORDs left to copy

$copydword2_cc:
    TEST     ECX, 1            ;DWORD left to copy ?
    JZ       $copydone2_cc     ;nope, we're done
    MOVD     MM0, [ESI]        ;read last DWORD from src
    MOVD     [EDI], MM0        ;store last DWORD to dst

$copydone2_cc:  
    FEMMS                      ;clear MMX state
}

/* block fill: fill a number of DWORDs at DWORD aligned destination
 with DWORD initializer using cacheable stores 
*/
__asm {
    MOV       EDI, [dst_ptr]   ;pointer to dst, DWORD aligned
    MOV       ECX, [blk_size]  ;number of DWORDs to copy
    MOVD      MM0, [fill_data] ;initialization data
    PUNPCKLDQ MM0, MM0         ;extend fill data to QWORD
    CMP       ECX, 1           ;less than one DWORD to fill ?
    JB        $filldone2_fc    ;yes, must be no DWORDs to fill, done
    TEST      EDI, 7           ;dst QWORD aligned?
    JZ        $dstqaligned2_fc ;yes
    MOVD      [EDI], MM0       ;store one DWORD to dst
    ADD       EDI, 4           ;dst++
    DEC       ECX              ;number of DWORDs to fill

$dstqaligned2_fc:
    MOV       EBX, ECX         ;number of DWORDs to fill
    SHR       ECX, 4           ;number of cache lines to fill
    JZ        $fillqwords2_fc  ;no whole cache lines to fill, maybe QWORDS
    ALIGN     16               ;align loop for optimal performance
$cloop2_fc:
   ADD EDI, 64 ;dst++
   MOVQ [EDI-64], MM0 ;store 1st DWORD in cache line to dst
   MOVQ [EDI-56], MM0 ;store 2nd DWORD in cache line to dst
   MOVQ [EDI-48], MM0 ;store 3rd DWORD in cache line to dst
   MOVQ [EDI-40], MM0 ;store 4th DWORD in cache line to dst
   MOVQ [EDI-32], MM0 ;store 5th DWORD in cache line to dst
   MOVQ [EDI-24], MM0 ;store 6th DWORD in cache line to dst
   MOVQ [EDI-16], MM0 ;store 7th DWORD in cache line to dst
   DEC ECX ;count--
   MOVQ [EDI-8], MM0 ;store 8th DWORD in cache line to dst
   JNZ $cloop2_fc ;until no more cache lines to copy

$fillqwords2_fc:
   MOV ECX, EBX ;number of DWORDs to fill
   AND EBX, 0xE ;number of QWORDS left to fill * 2
   JZ $filldword2_fc ;no QWORDS left, maybe DWORD left

   ALIGN 16 ;align loop for optimal performance

$qloop2_fc:
   MOVQ [EDI], MM0 ;store QWORD to dst
   ADD EDI, 8 ;dst++
   SUB EBX, 2 ;count--
   JNZ $qloop2_fc ;until no more QWORDS left to copy

$filldword2_fc:
   TEST ECX, 1 ;DWORD left to fill?
   JZ $filldone2_fc ;nope, we're done
   MOVQ [EDI], MM0 ;store last DWORD to dst

$filldone2_fc:
   FEMMS ;clear MMX state
}

Chapter 10  3DNow™ and MMX™ Optimizations  177
The following memory copy example is written with Microsoft Visual C++ in-line assembler syntax, and assumes that the Microsoft Processor Pack is installed (available from Microsoft's web site). This is a general purpose memcpy() routine, which can efficiently copy any size block, small or large. Data alignment is strongly recommended for good performance, but this code can handle non-aligned blocks.

Example 2: Optimized memcpy() for Any Data Size or Alignment

```c
#define TINY_BLOCK_COPY 64 // upper limit for movsd type copy
// The smallest copy uses the X86 "movsd" instruction, in an optimized
// form which is an "unrolled loop".

#define IN_CACHE_COPY 64 * 1024 // upper limit for movq/movq copy w/SW prefetch
// Next is a copy that uses the MMX registers to copy 8 bytes at a time.
// also using the "unrolled loop" optimization. This code uses
// the software prefetch instruction to get the data into the cache.

#define UNCACHED_COPY   197 * 1024 // upper limit for movq/movntq w/SW prefetch
// For larger blocks, which will spill beyond the cache, it's faster to
// use the Streaming Store instruction MOVNTQ. This write instruction
// bypasses the cache and writes straight to main memory. This code also
// uses the software prefetch instruction to pre-read the data.
// USE 64 * 1024 FOR THIS VALUE IF YOU'RE ALWAYS FILLING A "CLEAN CACHE"

#define BLOCK_PREFETCH_COPY  infinity // no limit for movq/movntq w/block prefetch
#define CACHEBLOCK 80h // # of 64-byte blocks (cache lines) for block prefetch
// For the largest size blocks, a special technique called Block Prefetch
// can be used to accelerate the read operations. Block Prefetch reads
// one address per cache line, for a series of cache lines, in a short loop.
// This is faster than using software prefetch. The technique is great for
// getting maximum read bandwidth, especially in DDR memory systems.

__asm {
    mov ecx, [n] ; number of bytes to copy
    mov edi, [dest] ; destination
    mov esi, [src] ; source
    mov ebx, ecx ; keep a copy of count

    cld
    cmp ecx, TINY_BLOCK_COPY
    jb $memcpy_ic_3 ; tiny? skip mmx copy

    cmp ecx, 32*1024 ; don't align between 32k-64k because
    jbe $memcpy_do_align ; it appears to be slower
    cmp ecx, 64*1024
    jbe $memcpy_align_done

    // continues on next page
```
$memcpy_do_align:
    mov ecx, 8 ; a trick that's faster than rep movsb...
    sub ecx, edi ; align destination to qword
    and ecx, 111b ; get the low bits
    sub ebx, ecx ; update copy count
    neg ecx ; set up to jump into the array
    add ecx, offset $memcpy_align_done
    jmp ecx ; jump to array of movsb's

align 4
    movsb
    movsb
    movsb
    movsb
    movsb
    movsb
    movsb
    movsb

$memcpy_align_done: ; destination is dword aligned
    mov ecx, ebx ; number of bytes left to copy
    shr ecx, 6 ; get 64-byte block count
    jz $memcpy_ic_2 ; finish the last few bytes
    cmp ecx, IN_CACHE_COPY/64 ; too big 4 cache? use uncached copy
    jae $memcpy_uc_test

    // continues on next page
// This is small block copy that uses the MMX registers to copy 8 bytes
// at a time. It uses the "unrolled loop" optimization, and also uses
// the software prefetch instruction to get the data into the cache.

align 16
$memcpy_ic_1: ; 64-byte block copies, in-cache copy

prefetchnta [esi + (200*64/34+192)] ; start reading ahead
movq mm0, [esi+0] ; read 64 bits
movq mm1, [esi+8]
movq [edi+0], mm0 ; write 64 bits
movq [edi+8], mm1 ; note: the normal movq writes the
movq mm2, [esi+16] ; data to cache; a cache line will be
movq mm3, [esi+24] ; allocated as needed, to store the data
movq [edi+16], mm2
movq [edi+24], mm3
movq mm0, [esi+32]
movq mm1, [esi+40]
movq [edi+32], mm0
movq [edi+40], mm1
movq mm2, [esi+48]
movq mm3, [esi+56]
movq [edi+48], mm2
movq [edi+56], mm3

add esi, 64 ; update source pointer
add edi, 64 ; update destination pointer
dec ecx ; count down
jnz $memcpy_ic_1 ; last 64-byte block?

$memcpy_ic_2:
mov ecx, ebx ; has valid low 6 bits of the byte count
$memcpy_ic_3:
shr ecx, 2 ; dword count
and ecx, 1111b ; only look at the "remainder" bits
neg ecx ; set up to jump into the array
add ecx, offset $memcpy_last_few
jmp ecx ; jump to array of movsd's

$memcpy_uc_test:
cmp ecx, UNCACHED_COPY/64 ; big enough? use block prefetch copy
jae $memcpy_bp_1

$memcpy_64_test:
or ecx, ecx ; tail end of block prefetch will jump here
jz $memcpy_ic_2 ; no more 64-byte blocks left

// continues on next page
For larger blocks, which will spill beyond the cache, it's faster to use the Streaming Store instruction MOVNTQ. This write instruction bypasses the cache and writes straight to main memory. This code also uses the software prefetch instruction to pre-read the data.

align 16

$memcpy_uc_1: ; 64-byte blocks, uncached copy

prefetchnta [esi + (200*64/34+192)] ; start reading ahead

movq   mm0,[esi+0]   ; read 64 bits
add    edi,64        ; update destination pointer
movq   mm1,[esi+8]
add    esi,64        ; update source pointer
movq   mm2,[esi-48]
movntq [edi-64], mm0 ; write 64 bits, bypassing the cache
movq   mm0,[esi-40]  ; note: movntq also prevents the CPU
movntq [edi-66], mm1 ; from READING the destination address
movq   mm1,[esi-32]  ; into the cache, only to be over-written
movntq [edi-48], mm2 ; so that also helps performance
movq   mm2,[esi-24]
movntq [edi-40], mm0
movq   mm0,[esi-16]
movntq [edi-32], mm1
movq   mm1,[esi-8]
movntq [edi-24], mm2
movntq [edi-16], mm0
dec    ecx
movntq [edi-8], mm1
jnz    $memcpy_uc_1  ; last 64-byte block?

jmp    $memcpy_ic_2  ; almost dont
// For the largest size blocks, a special technique called Block Prefetch
// can be used to accelerate the read operations. Block Prefetch reads
// one address per cache line, for a series of cache lines, in a short loop.
// This is faster then using software prefetch. The technique is great for
// getting maximum read bandwidth, especially in DDR memory systems.

#define COPY_LARGE_BLOCKS

memcpy_bp_1: ; large blocks, block prefetch copy

    cmp    ecx, CACHEBLOCK ; big enough to run another prefetch loop?
    jl     $memcpy_64_test ; no, back to regular uncached copy

    mov    eax, CACHEBLOCK / 2 ; block prefetch loop, unrolled 2X
    add    esi, CACHEBLOCK * 64 ; move to the top of the block

align 16

memcpy_bp_2:

    mov    edx, [esi-64] ; grab one address per cache line
    mov    edx, [esi-128] ; grab one address per cache line
    sub    esi, 128 ; go reverse order
    dec    eax ; count down the cache lines
    jnz    $memcpy_bp_2 ; keep grabbing more lines into cache

    mov    eax, CACHEBLOCK ; now that it's in cache, do the copy

align 16

memcpy_bp_3:

    movq   mm0, [esi] ; read 64 bits
    movq   mm1, [esi+ 8]
    movq   mm2, [esi+16]
    movq   mm3, [esi+24]
    movq   mm4, [esi+32]
    movq   mm5, [esi+40]
    movq   mm6, [esi+48]
    movq   mm7, [esi+56]

    add    esi, 64 ; update source pointer
    movntq [edi], mm0 ; write 64 bits, bypassing cache
    movntq [edi+ 8], mm1 ; note: movntq also prevents the CPU
    movntq [edi+16], mm2 ; from READING the destination address
    movntq [edi+24], mm3 ; into the cache, only to be over-written.
    movntq [edi+32], mm4 ; so that also helps performance
    movntq [edi+40], mm5
    movntq [edi+48], mm6
    movntq [edi+56], mm7

    add    edi, 64 ; update dest pointer

    dec    eax ; count down

    jnz    $memcpy_bp_3 ; keep copying
    sub    ecx, CACHEBLOCK ; update the 64-byte block count
    jmp    $memcpy_bp_1 ; keep processing blocks

// continues on next page
// The smallest copy uses the X86 "movsd" instruction, in an optimized
// form which is an "unrolled loop". Then it handles the last few bytes.
align 4
movsd
movsd ; perform last 1-15 dword copies
movsd
movsd
movsd
movsd
movsd
movsd
movsd
movsd
movsd ; perform last 1-7 dword copies
movsd
movsd
movsd
movsd
movsd
movsd
movsd
movsd
movsd

$memcpy_last_few: ; dword aligned from before movsd's
    mov ecx, ebx ; has valid low 2 bits of the byte count
    and ecx, 11b ; the last few cows must come home
    jz $memcpy_final ; no more, let's leave
    rep movsb ; the last 1, 2, or 3 bytes

$memcpy_final:
    emms ; clean up the MMX state
    sfence ; flush the write buffer
    mov eax, [dest] ; ret value = destination pointer

} }


Efficient 64-Bit Population Count Using MMX™ Instructions

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 following example code shows how to efficiently implement a population count function for 64-bit operands. The example is written for the inline assembler of Microsoft Visual C.

Function popcount64() is based on an O(log(n)) algorithm that successively groups the bits into groups of 2, 4, and 8 bits, while maintaining a count of the set bits in each group. This phase of the algorithm is described in detail in steps 1 through 3 of the section “Efficient Implementation of Population Count Function” on page 136.

In the final phase of popcount64(), the intermediate results from all eight 8-bit groups are summed using the PSADBW instruction. PSADBW is an extended MMX instruction that sums the absolute values of byte-wise differences between two MMX registers. In order to sum the eight bytes in an MMX register, the second source operand is set to zero. Thus the absolute difference for each byte equals the value of that byte in the first source operand.

Example:

```c
#include "amd3d.h"

__declspec (naked) unsigned int __stdcall popcount64 (unsigned __int64 v)
{
    static const __int64 C55 = 0x5555555555555555;
    static const __int64 C33 = 0x3333333333333333;
    static const __int64 C0F = 0x0F0F0F0F0F0F0F0F;

    __asm {
        MOVD MM0, [ESP+4] ; v_low
        PUNPCKLDQ MM0, [ESP+8] ; v
        MOVQ MM1, MM0 ; v
        PSRLD MM0, 1 ; v >> 1
        PAND MM0, [C55] ; (v >> 1) & 0x55555555
        PSUBD MM1, MM0 ; w = v - ((v >> 1) & 0x55555555)
        MOVQ MM0, MM1 ; w
        PSRLD MM1, 2 ; w >> 2
        PAND MM0, [C33] ; w & 0x33333333
        PAND MM1, [C33] ; (w >> 2) & 0x33333333
    }
}
```
PADDD MM0, MM1 ; x = (w & 0x33333333) + 
 ; ((w >> 2) & 0x33333333)
MOVQ MM1, MM0 ; x
PSRLD MM0, 4 ; x >> 4
PADDD MM0, MM1 ; x + (x >> 4)
PAND MM0, [C0F] ; y = (x + (x >> 4) & 
 ; 0x0F0F0F0F)
PXOR MM1, MM1 ; 0
PSADBW (MM0, MM1) ; sum across all 8 bytes
MOVD EAX, MM0 ; result in EAX per calling 
 ; convention
FEMMS ; clear MMX state
RET 8 ; pop 8-byte argument off 
 ; stack and return
}

Use MMX™ PXOR to Clear All Bits in an MMX Register

To clear all the bits in an MMX register to zero, use:

PXOR MMreg, MMreg

Note that PXOR MMreg, MMreg is dependent on previous 
writes to MMreg. Therefore, using PXOR in the manner 
described can lengthen dependency chains, which in return 
may lead to reduced performance. An alternative in such cases 
is to use:

zero DD 0

MOVD MMreg, DWORD PTR [zero]

i.e., to load a zero from a statically initialized and properly 
aligned memory location. However, loading the data from 
memory runs the risk of cache misses. Cases where MOVD is 
superior to PXOR are therefore rare and PXOR should be used 
in general.
Use MMX™ PCMPEQD to Set All Bits in an MMX Register

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

\[ \text{PCMPEQD MMreg, MMreg} \]

Note that “PCMPEQD MMreg, MMreg” is dependent on previous writes to MMreg. Therefore, using PCMPEQD in the manner described can lengthen dependency chains, which in turn may lead to reduced performance. An alternative in such cases is to use:

\[ \text{ones DQ 0FFFFFFFFFFFFFFFFFFh} \]
\[ \text{MOVQ MMreg, QWORD PTR [ones]} \]

i.e., to load a quadword of 0xFFFFFFFFFFFFFFFF from a statically initialized and properly aligned memory location. However, loading the data from memory runs the risk of cache misses. Therefore, cases where MOVQ is superior to PCMPEQD are rare and PCMPEQD should be used in general.

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

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

\[ \text{mabs DQ 7FFFFFFF7FFFFFFFh} \]
\[ \text{PAND MM0, [mabs]} ; \text{mask out sign bit} \]

Integer Absolute Value Computation Using MMX™ Instructions

The following examples show how to efficiently compute the absolute value of packed signed WORDs and packed signed DWORDs using MMX instructions. The algorithm works by checking the sign bit of the operand and constructing a mask from it. The mask is then used to conditionally compute first the one’s complement and then the two’s complement of the
operand in case the operand is negative, and leave the operand unchanged if the operand is positive or zero.

Note that the smallest negative number is mapped to itself by this code, but this also happens for calls to the C library function labs() and thus is perfectly acceptable.

Example 1 (packed WORDs):

```
; IN: MM0 = x
; OUT: MM0 = abs(x)
MOVQ MM1, MM0       ;x
PSRAW MM1, 15       ;x < 0 ? 0xffff : 0
PXOR MM0, MM1       ;x < 0 ? ~x : x
PSUBW MM0, MM1      ;x < 0 ? -x : x
```

Example 2 (packed DWORDs):

```
; IN: MM0 = x
; OUT: MM0 = abs(x)
MOVQ MM1, MM0       ;x
PSRAD MM1, 31       ;x < 0 ? 0xffffffff : 0
PXOR MM0, MM1       ;x < 0 ? ~x : x
PSUBD MM0, MM1      ;x < 0 ? -x : x
```

Optimized Matrix Multiplication

The multiplication of a 4x4 matrix with a 4x1 vector is commonly used in 3D graphics for geometry transformation. This routine serves to translate, scale, rotate, and apply perspective to 3D coordinates represented in homogeneous coordinates. The following code sample is a general 3D vertex transformation and 3DNow! optimized routine that completes in 18 cycles if aligned to a 32-byte cache line boundary and 22 cycles if aligned to a 16-byte, but not 32-byte boundary on the AMD-K6-2 and AMD-K6-III processors. The transformation takes 16 cycles on the AMD Athlon processor.
Matrix Multiplication

Code Sample

/* Function XForm performs a fully generalized 3D transform on an array of vertices pointed to by "v" and stores the transformed vertices in the location pointed to by "res". Each vertex consists of four floats. The 4x4 transform matrix is pointed to by "m". The matrix elements are also floats. The argument "numverts" indicates how many vertices have to be transformed. The computation performed for each vertex is:

res->x = v->x*m[0][0] + v->y*m[1][0] + v->z*m[2][0] + v->w*m[3][0]
res->y = v->x*m[0][1] + v->y*m[1][1] + v->z*m[2][1] + v->w*m[3][1]
res->z = v->x*m[0][2] + v->y*m[1][2] + v->z*m[2][2] + v->w*m[3][2]
res->w = v->x*m[0][3] + v->y*m[1][3] + v->z*m[2][3] + v->w*m[3][3]
*/

#define M00  0
#define M01  4
#define M02  8
#define M03 12
#define M10 16
#define M11 20
#define M12 24
#define M13 28
#define M20 32
#define M21 36
#define M22 40
#define M23 44
#define M30 48
#define M31 52
#define M32 56
#define M33 60

void XForm (float *res, const float *v, const float *m, int numverts)
{
    _asm {
        MOV EDX, [V]                ;EDX = source vector ptr
        MOV EAX, [M]                ;EAX = matrix ptr
        MOV EBX, [RES]              ;EBX = destination vector ptr
        MOV ECX, [NUMVERTS]         ;ECX = number of vertices to transform

        ;3DNow! version of fully general 3D vertex tranformation.
        ;Optimal for AMD Athlon (completes in 16 cycles)

        FEMMS ;clear MMX state
        ALIGN 16 ;for optimal branch alignment
    }
}
Chapter 10 3DNow!™ and MMX™ Optimizations

$$xform:

ADD EBX, 16 ; res++
MOVQ MM0, QWORD PTR [EDX] ; v->y | v->x
MOVQ MM1, QWORD PTR [EDX+8] ; v->w | v->z
ADD EDX, 16 ; v++
MOVQ MM2, MM0 ; v->y | v->x
MOVQ MM3, QWORD PTR [EAX+M00] ; m[0][1] | m[0][0]
PUNPCKLDQ MM0, MM0 ; v->x | v->x
MOVQ MM4, QWORD PTR [EAX+M10] ; m[1][1] | m[1][0]
PFMUL MM3, MM0 ; v->x*m[0][1] | v->y*m[0][0]
PUNPCKHDQ MM2, MM2 ; v->y | v->y
PFMUL MM4, MM2 ; v->y*m[1][1] | v->y*m[1][0]
MOVQ MM5, QWORD PTR [EAX+M02] ; m[0][3] | m[0][2]
MOVQ MM7, QWORD PTR [EAX+M12] ; m[1][3] | m[1][2]
MOVQ MM6, MM1 ; v->w | v->z
PFMUL MM5, MM0 ; v->x*m[0][3] | v->x*m[0][0]
MOVQ MM0, QWORD PTR [EAX+M20] ; m[2][1] | m[2][0]
PUNPCKLDQ MM1, MM1 ; v->z | v->z
PFMUL MM7, MM2 ; v->y*m[1][3] | v->y*m[1][2]
MOVQ MM2, QWORD PTR [EAX+M22] ; m[2][3] | m[2][2]
PFMUL MM0, MM1 ; v->z*m[2][1] | v->z*m[2][0]
PFMUL MM3, MM4 ; v->x*m[0][1]+v->y*m[1][1] | v->x*m[0][0]+v->y*m[1][0]
MOVQ MM4, QWORD PTR [EAX+M30] ; m[3][1] | m[3][0]
PFMUL MM2, MM1 ; v->z*m[2][3] | v->z*m[2][2]
PFADD MM5, MM7 ; v->x*m[0][3]+v->y*m[1][3] | v->x*m[0][2]+v->y*m[1][2]
MOVQ MM1, QWORD PTR [EAX+M32] ; m[3][3] | m[3][2]
PUNPCKHDQ MM6, MM6 ; v->w | v->w
PFADD MM3, MM0 ; v->x*m[0][1]+v->y*m[1][1]+v->z*m[2][1] | v->x*m[0][0]+v->y*m[1][0]+v->z*m[2][0]
PFMUL MM4, MM6 ; v->w*m[3][1] | v->w*m[3][0]
PFMUL MM1, MM6 ; v->w*m[3][3] | v->w*m[3][2]
PFADD MM5, MM2 ; v->x*m[0][3]+v->y*m[1][3]+v->z*m[2][3] | v->x*m[0][2]+v->y*m[1][2]+v->z*m[2][2]
PFADD MM3, MM4 ; v->x*m[0][1]+v->y*m[1][1]+v->z*m[2][1]+ v->w*m[3][1] | v->x*m[0][0]+v->y*m[1][0]+v->z*m[2][0] + v->w*m[3][0]

MOVQ [EBX-16], MM3 ; store res->y | res->x
PFADD MM5, MM1 ; v->x*m[0][3]+v->y*m[1][3]+v->z*m[2][3] + v->w*m[3][3] | v->x*m[0][2]+v->y*m[1][2]+v->z*m[2][2] + v->w*m[3][2]
MOVQ [EBX-8], MM5 ; store res->w | res->z
DEC ECX ; numverts--
JNZ $$XFORM ; until numverts == 0

FEMMS ; clear MMX state

}
Efficient 3D-Clipping Code Computation Using 3DNow!™ Instructions

Clipping is one of the major activities occurring in a 3D graphics pipeline. In many instances, this activity is split into two parts, which do not necessarily have to occur consecutively:

- Computation of the clip code for each vertex, where each bit of the clip code indicates whether the vertex is outside the frustum with regard to a specific clip plane.
- Examination of the clip code for a vertex and clipping if the clip code is non-zero.

The following example shows how to use 3DNow! instructions to efficiently implement a clip code computation for a frustum that is defined by:

- \(-w \leq x \leq w\)
- \(-w \leq y \leq w\)
- \(-w \leq z \leq w\)

3D-Clipping Code Sample

```
.DAT
RIGHT   EQU 01h
LEFT    EQU 02h
ABOVE   EQU 04h
BELOW   EQU 08h
BEHIND  EQU 10h
BEFORE  EQU 20h
ALIGN 8

ABOVE_RIGHT  DD  RIGHT
             DD  ABOVE
BELOW_LEFT   DD  LEFT
             DD  BELOW
BEHIND_BEFORE DD  BEFORE
             DD  BEHIND

.CODE
```
:: Generalized computation of 3D clip code (out code)
::
:: Register usage: IN MM5 y | x
:: MM6 w | z
::
:: OUT MM2 clip code (out code)
::
:: DESTROYS MM0, MM1, MM2, MM3, MM4

PXOR MM0, MM0 ; 0 | 0
MOVQ MM1, MM6 ; w | z
MOVQ MM4, MM5 ; y | x
PUNPCKHDQ MM1, MM1 ; w | w
MOVQ MM3, MM6 ; w | z
MOVQ MM2, MM5 ; y | x
PFSUBR MM3, MM0 ; -w | -z
PFSUBR MM2, MM0 ; -y | -x
PUNPCKLDQ MM3, MM6 ; z | -z
PFCMPGT MM4, MM1 ; y > w ? FFFFFFFF:0 | x > w ? FFFFFFFF:0
MOVQ MM0, QWORD PTR [ABOVE_RIGHT] ; ABOVE | RIGHT
PFCMPGT MM3, MM1 ; z > w ? FFFFFFFF:0 | -z > w ? FFFFFFFF:0
PFCMPGT MM2, MM1 ; -y > w ? FFFFFFFF:0 | -x > w ? FFFFFFFF:0
MOVQ MM1, QWORD PTR [BEHIND_BEFORE] ; BEHIND | BEFORE
PAND MM4, MM0 ; y > w ? ABOVE:0 | x > w ? RIGHT:0
MOVQ MM0, QWORD PTR [BELOW_LEFT] ; BELOW | LEFT
PAND MM3, MM1 ; z > w ? BEHIND:0 | -z > w ? BEFORE:0
PAND MM2, MM0 ; -y > w ? BELOW:0 | -x > w ? LEFT:0
POR MM2, MM4 ; BELOW, ABOVE | LEFT, RIGHT
POR MM2, MM3 ; BELOW, ABOVE, BEHIND | LEFT, RIGHT, BEFORE
POR MM2, MM3 ; BELOW, ABOVE, BEHIND | LEFT, RIGHT, BEFORE
PUNPCKHDQ MM2, MM2 ; BELOW, ABOVE, BEHIND | BELOW, ABOVE, BEHIND
POR MM2, MM1 ; zclip, yclip, xclip = clip code
Efficiently Determining Similarity Between RGBA Pixels

The manipulation of 32-bit RGBA pixels commonly occurs in graphics imaging. Each of the components (red, blue, green, and Alpha-channel) occupies one byte of the 32-bit pixel. The order of the components within the pixel can vary between systems. This optimization guideline shows how to efficiently determine whether two RGBA pixels are similar (i.e., approximately equal) regardless of the actual component ordering. The example also demonstrates techniques of general utility:

- Computing absolute differences using unsigned saturating arithmetic
- Performing unsigned comparisons in MMX using unsigned saturating arithmetic
- Combining multiple conditions into one condition for branching

The pixels similar() function determines whether two RGBA pixels are similar by computing the absolute difference between identical components of each pixel. If any absolute difference is greater than or equal to some cutoff value, TOL, the pixels are found to be dissimilar and the function returns 0. If all absolute differences for all components are less than TOL, the pixels are found to be similar and the function returns 1. The following algorithm is used in the MMX implementation.

**Step 1:** Absolute differences can be computed efficiently by taking advantage of unsigned saturation. Under unsigned saturation, if the difference between two entities is negative the result is zero. By performing subtraction on two operands in both directions, and ORing the results, the absolute difference is computed.

**Example 1:**

\[
\text{sub\_unsigned\_sat}(i,k) = (i > k) ? i-k : 0 \\
\text{sub\_unsigned\_sat}(k,i) = (i <= k) ? k-i : 0 \\
(sub\_unsigned\_sat(i,k) | sub\_unsigned\_sat(k,i))) = (i > k) ? i-k : k-i = abs(i-k)
\]

In this case, the source operands are bytes, so the MMX instruction PSUBUSB is used to subtract with unsigned
saturation, then the results are merged using POR. Note that on the AMD Athlon processor, the above computation should not be used in algorithms that require the sum of absolute differences, such as motion vector computations during MPEG-2 encoding as an extended MMX instruction, PSADBW, specifically exists for that purpose. For the AMD-K6 processor family or for blended code, the above computation can be used as a building block in the computation of sums of absolute differences.

**Step 2:** The absolute difference of each component is in the range 0...255. In order to compare these against the cutoff value, do not use MMX compare instructions, as these implement signed comparisons, which would flag any input above 127 to be less than the (positive) cutoff value. Instead, we turn again to unsigned saturated arithmetic. In order to test whether a value is below a cutoff value TOL, (TOL-1) is subtracted using unsigned saturation. An input less than TOL results in an output of zero, an input equal to or greater than TOL results in a non-zero output. Since the operands are byte size, again use the PSUBUSB instruction.

**Step 3:** According to the similarity metric chosen, two pixels are similar if the absolute differences of all components are less than the cutoff value. In other words, the pixels are similar if all results of the previous step are zero. This can be tested easily by concatenating and if the concatenation is equal to zero, all absolute differences are below the cutoff value and the pixels are thus similar under the chosen metric. MMX instructions do not require explicit concatenation, instead, the four byte operands can simply be viewed as a DWORD operand and compared against zero using the PCMPEQD instruction.

Note that the implementation of unsigned comparison in step 2 does not produce "clean" masks of all 1s or all 0s like the MMX compare instructions since this is not needed in the example code. Where this is required, the output for an unsigned comparison may be created as follows:
Example 2:

\[
\begin{align*}
    a > b & \implies \text{MOVQ MMreg1, [a]} \quad \text{PSUBUS* MMreg1, [b]} \quad \text{PCMPGT* MMreg1, [zero]} \\
    a \leq b & \implies \text{MOVQ MMreg1, [a]} \quad \text{PSUBUS* MMreg1, [b]} \quad \text{PCMPEQ* MMreg1, [zero]} \\
    a < b & \implies \text{MOVQ MMreg1, [b]} \quad \text{PSUBUS* MMreg1, [a]} \quad \text{PCMPGT* MMreg1, [zero]}
\end{align*}
\]

Example 3:

```c
#include <stdlib.h>
#include <stdio.h>

#define TOL 5

typedef struct {
    unsigned char r, g, b, a;
} PIXEL;

#ifdef C_VERSION
int _stdcall pixels_similar (PIXEL pixel1, PIXEL pixel2)
{
    return ((labs(((int)pixel1.r) - ((int)pixel2.r)) < TOL) &&
            (labs(((int)pixel1.g) - ((int)pixel2.g)) < TOL) &&
            (labs(((int)pixel1.b) - ((int)pixel2.b)) < TOL) &&
            (labs(((int)pixel1.a) - ((int)pixel2.a)) < TOL)) ? 1 : 0;
}
#else /* !C_VERSION */
static unsigned int tolerance = {((TOL-1) << 24) | ((TOL-1) << 16) |
                                ((TOL-1) << 8)  | ((TOL-1) <<  0) };
__declspec (naked) int _stdcall pixels_similar (PIXEL pixel1, PIXEL pixel2)
{
    __asm {
        MOVD MM0, [ESP+8]    ; a1 ... r1
        MOVD MM1, [ESP+4]    ; a2 ... r2
        MOQV MM2, MM0        ; a1 ... r1
        PSUBUSB MM0, MM1     ; a1>a2?a1-a2:0 ... r1>r2?r1-r2:0
        PSUBUSB MM1, MM2     ; a1<a2?a2-a1:0 ... r1<r2?r2-r1:0
        MOVD MM2, [tolerance]; TOL-1 TOL-1 TOL-1 TOL-1
        POR MM0, MM1         ; da=labs(a1-a2) ... dr=labs(r1-r2)
    }
}
#endif /* !C_VERSION */
```

Since MMX defines subtraction with unsigned saturation only for byte and WORD sized operands, the above code does not work for comparisons of unsigned DWORD operands.
Use the 3DNow! PAVGUSB instruction for MPEG-2 motion compensation. The PAVGUSB instruction produces the rounded averages of the eight unsigned 8-bit integer values in the source operand (an MMX register or a 64-bit memory location) and the eight corresponding unsigned 8-bit integer values in the destination operand (an MMX register). The PAVGUSB instruction is extremely useful in DVD (MPEG-2) decoding where motion compensation performs a lot of byte averaging between and within macroblocks. The PAVGUSB instruction helps speed up these operations. In addition, PAVGUSB can free up some registers and make unrolling the averaging loops possible.

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

**Example 1 (Avoid):**

```
MOV  ES1, DWORD PTR Src_MB
MOV  EDI, DWORD PTR Dst_MB
MOV  EDX, DWORD PTR SrcStride
MOV  EBX, DWORD PTR DstStride
MOVQ MM7, QWORD PTR [ConstFEFE]
MOVQ MM6, QWORD PTR [Const0101]
MOV  ECX, 16

L1:    
MOVQ MM0, [ESI]                    ;MM0=QWORD1
MOVQ MM1, [EDI]                    ;MM1=QWORD3
MOVQ MM2, MM0
MOVQ MM3, MM1
PAND MM2, MM6
PAND MM3, MM6
```
The following code fragment uses the 3DNow! PAVGUSB instruction to perform averaging between the source macroblock and destination macroblock:

**Example 1 (Preferred):**

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

L1:
    MOVQ MM0, [EAX] ; MM0 = QWORD1
    MOVQ MM1, [EAX+8] ; MM1 = QWORD2
    PAVGUSB MM0, [EDI] ; (QWORD1 + QWORD3)/2 with adjustment
    PAVGUSB MM1, [EDI+8]; (QWORD2 + QWORD4)/2 with adjustment
    ADD EAX, EDX
    MOVQ [EDI], MM0
    MOVQ [EDI+8], MM1
    LOOP L1
```
Efficient Implementation of floor() Using 3DNow!™ Instructions

The function floor() returns the greatest integer less than or equal to a given floating-point argument x. The integer result is returned as a floating-point number. In other words, floor() implements a floating-point-to-integer conversion that rounds towards negative infinity and then converts the result back to a floating-point number.

The 3DNow! instruction set supports only one type of floating-point to integer conversion, namely truncation, i.e., a conversion that rounds toward zero. For arguments greater than or equal to zero, rounding towards zero and rounding towards negative infinity returns identical results. For negative arguments, rounding towards negative infinity produces results that are smaller by 1 than results from rounding towards zero, unless the input is an integer.

The following code efficiently computes floor() based on the definition that floor(x) <= x. It uses PF2ID and PI2FD to compute float(trunc(x)). If the result is greater than x, it conditionally decrements the result by 1, thus computing floor(x). This computation transfers the input into the integer domain during the intermediate computation, which leads to incorrect results due to integer overflow with saturation if abs(x) > 2^31. This issue is addressed by observing that for single-precision numbers with absolute value > 2^24, the number contains only integer bits, and therefore floor(x) = x. The computation below, therefore, returns x for x > 2^24.

**Example:**

```assembly
MABS  DQ 7FFFFFFF7FFFFFFFh
TTTF  DQ 4B8000004B800000h
;; IN:  mm0 = x
;; OUT: mm0 = floor(x)
MOVQ  MM3, [MABS] ;mask for absolute value
PF2ID MM1, MM0 ;trunc(x)
MOVQ  MM4, [TTTF] ;2^24
PAND  MM3, MM0 ;abs(x)
PI2FD MM2, MM1 ;float(trunc(x))
PCMPGT D MM3, MM4 ;abs(x) > 2^24
; 0xffffffff : 0
```

Chapter 10  3DNow!™ and MMX™ Optimizations  197
Stream of Packed Unsigned Bytes

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

**Example 1:**
outside loop:
PXOR MM0, MM0

inside loop:
MOVQ MM4, MM0 ;x
PFCMPGT MM2, MM4 ;float(trunc(x)) > x ? 0xffffffff : 0
PAND MM0, MM3 ;abs(x) > 2^24 ? x : 0
PADD MM1, MM2 ;float(trunc(x)) > x ? trunc(x)-1 : trunc(x)
PI2FD MM4, MM1 ;floor(x)
PANDN MM3, MM4 ;abs(x) > 2^24 ? 0 : floor(x)
POR MM0, MM3 ;abs(x) > 2^24 ? x : floor(x)
Complex Number Arithmetic

Complex numbers have a “real” part and an “imaginary” part. Multiplying complex numbers (ex. 3 + 4i) is an integral part of many algorithms such as Discrete Fourier Transform (DFT) and complex FIR filters. Complex number multiplication is shown below:

\[(\text{src0.real} + \text{src0.imag}) \times (\text{src1.real} + \text{src1.imag}) = \text{result}\]
\[\text{result.real} = \text{src0.real} \times \text{src1.real} - \text{src0.imag} \times \text{src1.imag}\]
\[\text{result.imag} = \text{src0.real} \times \text{src1.imag} + \text{src0.imag} \times \text{src1.real}\]

Example 1:
\[(1+2i) \times (3+4i) \Rightarrow \text{result.real} + \text{result.imag}\]
\[\text{result.real} = 1 \times 3 - 2 \times 4 = -5\]
\[\text{result.imag} = 1 \times 4i + 2i \times 3 = 10i\]
\[\text{result} = -5 + 10i\]

Assuming that complex numbers are represented as two element vectors \([v.\text{real}, v.\text{imag}]\), one can see the need for swapping the elements of src1 to perform the multiplies for result.imag, and the need for a mixed positive/negative accumulation to complete the parallel computation of result.real and result.imag.

PSWAPD performs the swapping of elements for src1 and PFPNACC performs the mixed positive/negative accumulation to complete the computation. The code example below summarizes the computation of a complex number multiply.

Example 2:

\[; \text{MM0} = s0.\text{imag} | s0.\text{real} ; \text{reg_hi} | \text{reg_lo}\]
\[; \text{MM1} = s1.\text{imag} | s1.\text{real}\]

PSWAPD MM2, MM0 ; M2 = s0.\text{real} | s0.\text{imag}
PFMUL MM0, MM1 ; M0 = s0.\text{imag} * s1.\text{imag} | s0.\text{real} * s1.\text{real}
PFMUL MM1, MM2 ; M1 = s0.\text{real} * s1.\text{imag} | s0.\text{imag} * s1.\text{real}
PFPNACC MM0, MM1 ; M0 = res.\text{imag} | res.\text{real}\]

PSWAPD supports independent source and result operands and enables PSWAPD to also perform a copy function. In the above example, this eliminates the need for a separate “MOVQ MM2, MM0” instruction.
11

General x86 Optimization Guidelines

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

Short Forms

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

Example 1 (Avoid):

CMP REG, 0

Example 2 (Preferred):

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

Dependencies

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

Register Operands

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

Stack Allocation

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

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

The term microarchitecture refers to the design techniques used in the processor to reach the target cost, performance, and functionality goals. The AMD Athlon processor microarchitecture is a decoupled decode/execution design approach. In other words, the decoders essentially operate independent of the execution units, and the execution core uses a small number of instructions and simplified circuit design for fast single-cycle execution and fast operating frequencies.

The term design implementation refers to the actual logic and circuit designs from which the processor is created according to the microarchitecture specifications.
 AMD Athlon™ Processor Microarchitecture

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

Superscalar Processor

The AMD Athlon processor is an aggressive, out-of-order, three-way superscalar x86 processor. It can fetch, decode, and issue up to three x86 instructions per cycle with a centralized instruction control unit (ICU) and two independent instruction schedulers—an integer scheduler and a floating-point scheduler. These two schedulers can simultaneously issue up to nine OPs to the three general-purpose integer execution units (IEUs), three address-generation units (AGUs), and three floating-point/3DNow!™/MMX™ execution units. The AMD Athlon moves integer instructions down the integer execution pipeline, which consists of the integer scheduler and the IEUs, as shown in Figure 1 on page 205. Floating-point instructions are handled by the floating-point execution pipeline, which consists of the floating-point scheduler and the x87/3DNow!/MMX execution units.
Figure 1. AMD Athlon™ Processor Block Diagram

Instruction Cache

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

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

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

**Predecode**

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

**Branch Prediction**

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

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

The BTB is a 2048-entry table that caches in each entry the predicted target address of a branch.
In addition, the AMD Athlon processor implements a 12-entry return address stack to predict return addresses from a near or far call. As CALLs are fetched, the next EIP is pushed onto the return stack. Subsequent RETs pop a predicted return address off the top of the stack.

**Early Decoding**

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

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

**DirectPath Decoder**

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

**VectorPath Decoder**

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

The instruction control unit (ICU) is the control center for the AMD Athlon processor. The ICU controls the following resources—the centralized in-flight reorder buffer, the integer scheduler, and the floating-point scheduler. In turn, the ICU is responsible for the following functions—MacroOP dispatch, MacroOP retirement, register and flag dependency resolution and renaming, execution resource management, interrupts, exceptions, and branch mispredictions.

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

Data Cache

The L1 data cache contains two 64-bit ports. It is a write-allocate and writeback cache that uses an LRU replacement policy. The data cache and instruction cache are both two-way set-associative and 64-Kbytes in size. It is divided into 8 banks where each bank is 8 bytes wide. In addition, this cache supports the MOESI (Modified, Owner, Exclusive, Shared, and Invalid) cache coherency protocol and data parity.

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

The integer scheduler is based on a three-wide queuing system (also known as a reservation station) that feeds three integer execution positions or pipes. The reservation stations are six entries deep, for a total queuing system of 18 integer MacroOPs. Each reservation station divides the MacroOPs into integer and address generation OPs, as required.

### Integer Execution Unit

The integer execution pipeline consists of three identical pipes—0, 1, and 2. Each integer pipe consists of an integer execution unit (IEU) and an address generation unit (AGU). The integer execution pipeline is organized to match the three MacroOP dispatch pipes in the ICU as shown in Figure 2 on page 209. MacroOPs are broken down into OPs in the schedulers. OPs issue when their operands are available either from the register file or result buses.

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

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

All integer operations can be handled within any of the three IEUs with the exception of multiplies. Multiplies are handled by a pipelined multiplier that is attached to the pipeline at pipe 0. See Figure 2 on page 209. 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 OPs from issuing at the appropriate time.

**Floating-Point Scheduler**

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

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

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

Figure 3. Floating-Point Unit Block Diagram

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

The load-store unit (LSU) manages data load and store accesses to the L1 data cache and, if required, to the L2 cache or system memory. The 44-entry LSU provides a data interface for both the integer scheduler and the floating-point scheduler. It consists of two queues—a 12-entry queue for L1 cache load and store accesses and a 32-entry queue for L2 cache or system memory load and store accesses. The 12-entry queue can request a maximum of two L1 cache loads and two L1 cache (32-bit) stores per cycle. The 32-entry queue effectively holds requests that missed in the L1 cache probe by the 12-entry queue. Finally, the LSU ensures that the architectural load and store ordering rules are preserved (a requirement for x86 architecture compatibility).

Figure 4. Load/Store Unit
L2 Cache

The AMD Athlon processor Models 1 and 2 contain a very flexible onboard L2 controller. It uses an independent bus to access up to 8 Mbytes of industry-standard SRAMs. There are full on-chip tags for a 512-Kbyte cache, while larger sizes use a partial tag system. In addition, there is a two-level data TLB structure. The first-level TLB is fully associative and contains 32 entries (24 that map 4-Kbyte pages and eight that map 2-Mbyte or 4-Mbyte pages). The second-level TLB is four-way set associative and contains 256 entries, which can map 4-Kbyte pages.

Newer Athlon processor models (such as Models 3 and 4) contain an integrated L2 Cache. This full-speed on-die L2 cache features an exclusive cache architecture as opposed to the inclusive cache architecture utilized by previous-generation x86 processors. This 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 AMD Athlon processor victim buffer contains data evicted from the L1 cache. It features up to eight 64-byte entries, where each entry corresponds to a 64-byte cache line. The new AMD Athlon processor 16-way set associative cache is eight times more associative than previous AMD Athlon processors which feature a 2-way set associative cache. Increasing the set associativity increases the hit rate by reducing data conflicts. This translates into more possible locations in which important data can reside without having to throw away other often-used data in the L2 cache.

Write Combining

See Appendix C, “Implementation of Write Combining,” for detailed information about write combining.
**AMD Athlon™ System Bus**

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

Pipeline and Execution Unit Resources Overview

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

Fetch and Decode Pipeline Stages

Figure 5 and Figure 6 on page 216 show the AMD Athlon processor instruction fetch and decoding pipeline stages. The pipeline consists of one cycle for instruction fetches and four cycles of instruction alignment and decoding. The three ports in stage 5 provide a maximum bandwidth of three MacroOPs per cycle for dispatching to the instruction control unit (ICU).
The most common x86 instructions flow through the DirectPath pipeline stages and are decoded by hardware. The less common instructions, which require microcode assistance, flow through the VectorPath. Although the DirectPath decodes the common x86 instructions, it also contains VectorPath instruction data, which allows it to maintain dispatch order at the end of cycle 5.
Appendix B Pipeline and Execution Unit Resources Overview

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

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

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

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

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

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

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

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

Cycle 6: IDEC/Rename
At the instruction decoder (IDEC)/rename stage, integer and floating-point MacroOPs diverge in the pipeline. Integer MacroOPs are scheduled for execution in the next cycle. Floating-point MacroOPs have their floating-point stack operands mapped to registers. Both integer and floating-point MacroOPs are placed into the ICU.
Integer Pipeline Stages

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

Figure 7. Integer Execution Pipeline

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

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

**Cycle 8: EXEC**

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

**Cycle 9: ADDGEN**

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

**Cycle 10: DCACC**

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

**Cycle 11: RESP**

In the response (RESP) pipeline stage, the data cache returns hit/miss status and data for the request from DCACC.
Floating-Point Pipeline Stages

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

Figure 9. Floating-Point Unit Block Diagram

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

Figure 10. Floating-Point Pipeline Stages
Cycle 7: STKREN

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

Cycle 8: REGREN

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

Cycle 9: SCHEDW

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

Cycle 10: SCHED

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

Cycle 11: FREG

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

Cycles 12–15: Floating-Point Execution (FEXEC1–4)

The FPU has three logical pipes—FADD, FMUL, and FSTORE. Each pipe may have several associated execution units. MMX execution is in both the FADD and FMUL pipes, with the exception of MMX instructions involving multiplies, which are limited to the FMUL pipe. The FMUL pipe has special support for long latency operations.

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

Terminology

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

Operands

The three types of operands are as follows:

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

Results

The two types of results are as follows:

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

Examples

The following examples illustrate the operand and result definitions:

```assembly
ADD    EAX, EBX
```

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

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

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

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

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

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

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

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

Table 2. Integer Pipeline Operation Types

<table>
<thead>
<tr>
<th>Category</th>
<th>Execution Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Memory Load or Store Operations</td>
<td>L/S</td>
</tr>
<tr>
<td>Address Generation Operations</td>
<td>AGU</td>
</tr>
<tr>
<td>Integer Execution Unit Operations</td>
<td>IEU</td>
</tr>
<tr>
<td>Integer Multiply Operations</td>
<td>IMUL</td>
</tr>
</tbody>
</table>

Table 3. Integer Decode Types

<table>
<thead>
<tr>
<th>x86 Instruction</th>
<th>Decode Type</th>
<th>OPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV CX, [SP+4]</td>
<td>DirectPath</td>
<td>AGU, L/S</td>
</tr>
<tr>
<td>ADD AX, BX</td>
<td>DirectPath</td>
<td>IEU</td>
</tr>
<tr>
<td>CMP CX, [AX]</td>
<td>VectorPath</td>
<td>AGU, L/S, IEU</td>
</tr>
<tr>
<td>JZ Addr</td>
<td>DirectPath</td>
<td>IEU</td>
</tr>
</tbody>
</table>

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

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

Table 4. Floating-Point Pipeline Operation Types

<table>
<thead>
<tr>
<th>Category</th>
<th>Execution Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPU/3DNow!/MMX Load/store or Miscellaneous Operations</td>
<td>FSTORE</td>
</tr>
<tr>
<td>FPU/3DNow!/MMX Multiply Operation</td>
<td>FMUL</td>
</tr>
<tr>
<td>FPU/3DNow!/MMX Arithmetic Operation</td>
<td>FADD</td>
</tr>
</tbody>
</table>

Table 5. Floating-Point Decode Types

<table>
<thead>
<tr>
<th>x86 Instruction</th>
<th>Decode Type</th>
<th>OPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD ST, ST(i)</td>
<td>DirectPath</td>
<td>FADD</td>
</tr>
<tr>
<td>FSIN</td>
<td>VectorPath</td>
<td>various</td>
</tr>
<tr>
<td>PFACC</td>
<td>DirectPath</td>
<td>FADD</td>
</tr>
<tr>
<td>PFRSQRT</td>
<td>DirectPath</td>
<td>FMUL</td>
</tr>
</tbody>
</table>

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

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

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

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

### Table 6. Load/Store Unit Stages

<table>
<thead>
<tr>
<th>Stage 1 (Cycle 8)</th>
<th>Stage 2 (Cycle 9)</th>
<th>Stage 3 (Cycle 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Calculation / LS1 Scan</td>
<td>Transport Address to Data Cache</td>
<td>Data Cache Access / LS2 Data Forward</td>
</tr>
</tbody>
</table>

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

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

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

The samples in Table 7 on page 227 and Table 8 on page 228 show the execution behavior of several series of instructions as a function of decode constraints, dependencies, and execution resource constraints.

The sample tables show the x86 instructions, the decode pipe in the integer execution pipeline, the decode type, the clock counts, and a description of the events occurring within the processor. The decode pipe gives the specific IEU used (see Figure 7 on page 218). The decode type specifies either VectorPath (VP) or DirectPath (DP).

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

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

**Note:** Instructions execute more efficiently (that is, without delays) when scheduled apart by suitable distances based on dependencies. In general, the samples in this section show poorly scheduled code in order to illustrate the resultant effects.
### Table 7. Sample 1—Integer Register Operations

<table>
<thead>
<tr>
<th>Instruction Number</th>
<th>Instruction</th>
<th>Decode Pipe</th>
<th>Decode Type</th>
<th>Clocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IMUL EAX, ECX</td>
<td>0</td>
<td>VP</td>
<td>D</td>
</tr>
<tr>
<td>2</td>
<td>INC ESI</td>
<td>0</td>
<td>DP</td>
<td>D</td>
</tr>
<tr>
<td>3</td>
<td>MOV EDI, 0x07F4</td>
<td>1</td>
<td>DP</td>
<td>D</td>
</tr>
<tr>
<td>4</td>
<td>ADD EDI, EBX</td>
<td>2</td>
<td>DP</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>SHL EAX, 8</td>
<td>0</td>
<td>DP</td>
<td>D</td>
</tr>
<tr>
<td>6</td>
<td>OR EAX, 0x0F</td>
<td>1</td>
<td>DP</td>
<td>D</td>
</tr>
<tr>
<td>7</td>
<td>INC EBX</td>
<td>2</td>
<td>DP</td>
<td>D</td>
</tr>
<tr>
<td>8</td>
<td>ADD ESI, EDX</td>
<td>0</td>
<td>DP</td>
<td>D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMUL EAX, ECX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC ESI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV EDI, 0x07F4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD EDI, EBX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHL EAX, 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR EAX, 0x0F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC EBX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD ESI, EDX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Comments for Each Instruction Number

1. The IMUL is a VectorPath instruction. It cannot be decoded or paired with other operations and, therefore, dispatches alone in pipe 0. The multiply latency is four cycles.

2. The simple INC operation is paired with instructions 3 and 4. The INC executes in IEU0 in cycle 4.

3. The MOV executes in IEU1 in cycle 4.

4. The ADD operation depends on instruction 3. It executes in IEU2 in cycle 5.

5. The SHL operation depends on the multiply result (instruction 1). The MacroOP waits in a reservation station and is eventually scheduled to execute in cycle 7 after the multiply result is available.

6. This operation executes in cycle 8 in IEU1.

7. This simple operation has a resource contention for execution in IEU2 in cycle 5. Therefore, the operation does not execute until cycle 6.

8. The ADD operation executes immediately in IEU0 after dispatching.
### Table 8. Sample 2—Integer Register and Memory Load Operations

<table>
<thead>
<tr>
<th>Instr Num</th>
<th>Instruction</th>
<th>Decode Pipe</th>
<th>Decode Type</th>
<th>Clocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DEC EDX</td>
<td>0</td>
<td>DP</td>
<td>D I E</td>
</tr>
<tr>
<td>2</td>
<td>MOV EDI, [ECX]</td>
<td>1</td>
<td>DP</td>
<td>D I &amp;/S A $</td>
</tr>
<tr>
<td>3</td>
<td>SUB EAX, [EDX+20]</td>
<td>2</td>
<td>DP</td>
<td>D I &amp;/S A $/I E</td>
</tr>
<tr>
<td>4</td>
<td>SAR EAX, 5</td>
<td>0</td>
<td>DP</td>
<td>D I E</td>
</tr>
<tr>
<td>5</td>
<td>ADD ECX, [EDI+4]</td>
<td>1</td>
<td>DP</td>
<td>D I &amp;/S A $</td>
</tr>
<tr>
<td>6</td>
<td>AND EBX, 0x1F</td>
<td>2</td>
<td>DP</td>
<td>D I E</td>
</tr>
<tr>
<td>7</td>
<td>MOV ESI, [0x0F100]</td>
<td>0</td>
<td>DP</td>
<td>D I &amp; S A $</td>
</tr>
<tr>
<td>8</td>
<td>OR ECX, [ESI+EAX*4+8]</td>
<td>1</td>
<td>DP</td>
<td>I &amp;/S A $ E</td>
</tr>
</tbody>
</table>

**Comments for Each Instruction Number**

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

Implementation of Write Combining

Introduction

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

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

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

This appendix uses the following definitions and abbreviations:

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

What is Write Combining?

Write combining is the merging of multiple memory write cycles that target locations within the address range of a write buffer. The AMD Athlon processor combines multiple memory-write cycles to a 64-byte buffer whenever the memory address is within a WC or WT memory type region. The processor continues to combine writes to this buffer without writing the data to the system, as long as certain rules apply (see Table 9 on page 232 for more information).

Programming Details

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

1. Verify the presence of an AMD Athlon processor by using the CPUID instruction to check for the instruction family code and vendor identification of the processor. Standard function 0 on AMD processors returns a vendor identification string of “AuthenticAMD” in registers EBX, EDX, and ECX. Standard function 1 returns the processor signature in
register EAX, where EAX[11–8] contains the instruction family code. For the AMD Athlon processor, the instruction family code is six.

2. In addition, the presence of the MTRRs is indicated by bit 12 and the presence of the PAT extension is indicated by bit 16 of the extended features bits returned in the EDX register by CPUID function 8000_0001h. See the AMD Processor Recognition Application Note, order no. 20734 for more details on the CPUID instruction.

3. Write combining is controlled by the MTRRs and PAT. Write combining should be enabled for the appropriate memory ranges. The AMD Athlon processor MTRRs and PAT are compatible with the Pentium® II.

Write-Combining Operations

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

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

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

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

Combining is able to continue until interrupted by one of the conditions listed in Table 9 on page 232. When combining is interrupted, one or more bus commands are issued to the system for that write buffer, as described by Table 10 on page 233.
### Table 9. Write Combining Completion Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-WB write outside of current buffer</td>
<td>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.</td>
</tr>
<tr>
<td>I/O Read or Write</td>
<td>Any IN/INS or OUT/OUTS instruction closes combining. The implied memory type for all IN/OUT instructions is UC, which cannot be combined.</td>
</tr>
<tr>
<td>Serializing instructions</td>
<td>Any serializing instruction closes combining. These instructions include: MOVCRx, MOVDRx, WRMSR, INVD, INVLPG, WBINVD, LGDT, LLDT, LIDT, LTR, CPUID, IRET, RSM, INIT, HALT.</td>
</tr>
<tr>
<td>Flushing instructions</td>
<td>Any flush instruction causes the WC to complete.</td>
</tr>
<tr>
<td>Locks</td>
<td>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.</td>
</tr>
<tr>
<td>Uncacheable Read</td>
<td>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.</td>
</tr>
<tr>
<td>Different memory type</td>
<td>Any WT write while write-combining for WC memory or any WC write while write combining for WT memory closes write combining.</td>
</tr>
<tr>
<td>Buffer full</td>
<td>Write combining is closed if all 64 bytes of the write buffer are valid.</td>
</tr>
<tr>
<td>WT time-out</td>
<td>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.</td>
</tr>
<tr>
<td>WT write fills byte 7</td>
<td>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.</td>
</tr>
<tr>
<td>WT Nonsequential</td>
<td>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.</td>
</tr>
<tr>
<td>TLB AD bit set</td>
<td>Write combining is closed whenever a TLB reload sets the accessed (A) or dirty (D) bits of a Pde or Pte.</td>
</tr>
</tbody>
</table>
Sending Write-Buffer Data to the System

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

Table 10. AMD Athlon™ System Bus Command Generation Rules

1. If all eight quadwords are either full (8 bytes valid) or empty (0 bytes valid), a Write-Quadword system command is issued, with an 8-byte mask representing which of the eight quadwords are valid. If this case is true, do not proceed to the next rule.

2. If all longwords are either full (4 bytes valid) or empty (0 bytes valid), a Write-Longword system command is issued for each 32-byte buffer half that contains at least one valid longword. The mask for each Write-Longword system command indicates which longwords are valid in that 32-byte write buffer half. If this case is true, do not proceed to the next rule.

3. Sequence through all eight quadwords of the write buffer, from quadword 0 to quadword 7. Skip over a quadword if no bytes are valid. Issue a Write-Quad system command if all bytes are valid, asserting one mask bit. Issue a Write-Longword system command if the quadword contains one aligned longword, asserting one mask bit. Otherwise, issue a Write-Byte system command if there is at least one valid byte, asserting a mask bit for each valid byte.
Appendix D

Performance-Monitoring Counters

This chapter describes how to use the AMD Athlon™ processor performance monitoring counters.

Overview

The AMD Athlon processor provides four 48-bit performance counters, which allows four types of events to be monitored simultaneously. These counters can either count events or measure duration. When counting events, a counter is incremented each time a specified event takes place or a specified number of events takes place. When measuring duration, a counter counts the number of processor clocks that occur while a specified condition is true. The counters can count events or measure durations that occur at any privilege level. Table 11 on page 238 lists the events that can be counted with the performance monitoring counters.

The performance counters are not guaranteed to be fully accurate and should be used as a relative measure of performance to assist in application tuning. Unlisted event numbers are reserved and their results undefined.
Performance Counter Usage

The performance monitoring counters are supported by eight MSRs—PerfEvtSel[3:0] are the performance event select MSRs, and PerfCtr[3:0] are the performance counter MSRs. These registers can be read from and written to using the RDMSR and WRMSR instructions, respectively.

The PerfEvtSel[3:0] registers are located at MSR locations C001_0000h to C001_0003h. The PerfCtr[3:0] registers are located at MSR locations C001_0004h to C0001_0007h and are 64-byte registers.

The PerfEvtSel[3:0] registers can be accessed using the RDMSR/WRMSR instructions only when operating at privilege level 0. The PerfCtr[3:0] MSRs can be read from any privilege level using the RDPMC (read performance-monitoring counters) instruction, if the PCE flag in CR4 is set.

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

The PerfEvtSel[3:0] MSRs, shown in Figure 11, control the operation of the performance-monitoring counters, with one register used to set up each counter. These MSRs specify the events to be counted, how they should be counted, and the privilege levels at which counting should take place. The functions of the flags and fields within these MSRs are as are described in the following sections.
Figure 11. PerfEvtSel[3:0] Registers

**Event Select Field**
(Bits 0–7)

These bits are used to select the event to be monitored. See Table 11 on page 238 for a list of event masks and their 8-bit codes.

**Unit Mask Field**
(Bits 8–15)

These bits are used to further qualify the event selected in the event select field. For example, for some cache events, the mask is used as a MESI-protocol qualifier of cache states. See Table 11 on page 238 for a list of unit masks and their 8-bit codes.

**USR (User Mode) Flag**
(Bit 16)

Events are counted only when the processor is operating at privilege levels 1, 2 or 3. This flag can be used in conjunction with the OS flag.

**OS (Operating System Mode) Flag**
(Bit 17)

Events are counted only when the processor is operating at privilege level 0. This flag can be used in conjunction with the USR flag.

**E (Edge Detect) Flag**
(Bit 18)

When this flag is set, edge detection of events is enabled. The processor counts the number of negated-to-asserted transitions of any condition that can be expressed by the other fields. The mechanism is limited in that it does not permit back-to-back assertions to be distinguished. This mechanism allows software to measure not only the fraction of time spent in a particular state, but also the average length of time spent in such a state (for example, the time spent waiting for an interrupt to be serviced).
PC (Pin Control) Flag (Bit 19)  
When this flag is set, the processor toggles the PMi pins when the counter overflows. When this flag is clear, the processor toggles the PMi pins and increments the counter when performance monitoring events occur. The toggling of a pin is defined as assertion of the pin for one bus clock followed by negation.

INT (APIC Interrupt Enable) Flag (Bit 20)  
When this flag is set, the processor generates an interrupt through its local APIC on counter overflow.

EN (Enable Counter) Flag (Bit 22)  
This flag enables/disables the PerfEvtSeln MSR. When set, performance counting is enabled for this counter. When clear, this counter is disabled.

INV (Invert) Flag (Bit 23)  
By inverting the Counter Mask Field, this flag inverts the result of the counter comparison, allowing both greater than and less than comparisons.

Counter Mask Field (Bits 31–24)  
For events which can have multiple occurrences within one clock, this field is used to set a threshold. If the field is non-zero, the counter increments each time the number of events is greater than or equal to the counter mask. Otherwise if this field is zero, then the counter increments by the total number of events.

Table 11. Performance-Monitoring Counters

<table>
<thead>
<tr>
<th>Event Number</th>
<th>Source Unit</th>
<th>Notes / Unit Mask (bits 15–8)</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>40h</td>
<td>DC</td>
<td></td>
<td>Data cache accesses</td>
</tr>
<tr>
<td>41h</td>
<td>DC</td>
<td></td>
<td>Data cache misses</td>
</tr>
<tr>
<td>42h</td>
<td>DC</td>
<td>xxx1_xxxxb = Modified (M)</td>
<td>Data cache refills from L2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxxx_1xxxxb = Owner (O)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxxx_x1xxxb = Exclusive (E)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxxx_xx1xb = Shared (S)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxxx_xxx1b = Invalid (I)</td>
<td></td>
</tr>
<tr>
<td>43h</td>
<td>DC</td>
<td>xxx1_xxxxb = Modified (M)</td>
<td>Data cache refills from system</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxxx_1xxxxb = Owner (O)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxxx_x1xxxb = Exclusive (E)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxxx_xx1xb = Shared (S)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxxx_xxx1b = Invalid (I)</td>
<td></td>
</tr>
</tbody>
</table>
### Table 11. Performance-Monitoring Counters (Continued)

<table>
<thead>
<tr>
<th>Event Number</th>
<th>Source Unit</th>
<th>Notes / Unit Mask (bits 15–8)</th>
<th>Event Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>44h</td>
<td>DC</td>
<td>xxx1_xxxxxb = Modified (M)</td>
<td>Data cache writebacks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxxx_1xxxb = Owner (O)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxxx_xlxxb = Exclusive (E)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxxx_xx1xb = Shared (S)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>xxxx_xxx1b = Invalid (I)</td>
<td></td>
</tr>
<tr>
<td>45h</td>
<td>DC</td>
<td></td>
<td>L1 DTLB misses and L2 DTLB hits</td>
</tr>
<tr>
<td>46h</td>
<td>DC</td>
<td></td>
<td>L1 and L2 DTLB misses</td>
</tr>
<tr>
<td>47h</td>
<td>DC</td>
<td></td>
<td>Misaligned data references</td>
</tr>
<tr>
<td>80h</td>
<td>PC</td>
<td></td>
<td>Instruction cache fetches</td>
</tr>
<tr>
<td>81h</td>
<td>PC</td>
<td></td>
<td>Instruction cache misses</td>
</tr>
<tr>
<td>84h</td>
<td>PC</td>
<td></td>
<td>L1 ITLB misses (and L2 ITLB hits)</td>
</tr>
<tr>
<td>85h</td>
<td>PC</td>
<td></td>
<td>(L1 and) L2 ITLB misses</td>
</tr>
<tr>
<td>C0h</td>
<td>FR</td>
<td></td>
<td>Retired instructions (includes exceptions, interrupts, resyncs)</td>
</tr>
<tr>
<td>C1h</td>
<td>FR</td>
<td></td>
<td>Retired Ops</td>
</tr>
<tr>
<td>C2h</td>
<td>FR</td>
<td></td>
<td>Retired branches (conditional, unconditional, exceptions, interrupts)</td>
</tr>
<tr>
<td>C3h</td>
<td>FR</td>
<td></td>
<td>Retired branches mispredicted</td>
</tr>
<tr>
<td>C4h</td>
<td>FR</td>
<td></td>
<td>Retired taken branches</td>
</tr>
<tr>
<td>C5h</td>
<td>FR</td>
<td></td>
<td>Retired taken branches mispredicted</td>
</tr>
<tr>
<td>C6h</td>
<td>FR</td>
<td></td>
<td>Retired far control transfers</td>
</tr>
<tr>
<td>C7h</td>
<td>FR</td>
<td></td>
<td>Retired resync branches (only non-control transfer branches counted)</td>
</tr>
<tr>
<td>CDh</td>
<td>FR</td>
<td></td>
<td>Interrupts masked cycles (IF=0)</td>
</tr>
<tr>
<td>CEh</td>
<td>FR</td>
<td></td>
<td>Interrupts masked while pending cycles (INTR while IF=0)</td>
</tr>
<tr>
<td>CFh</td>
<td>FR</td>
<td></td>
<td>Number of taken hardware interrupts</td>
</tr>
</tbody>
</table>
PerfCtr[3:0] MSRs (MSR Addresses C001_0004h–C001_0007h)

The PerfCtr registers are model-specific registers that can be read using a special read performance-monitoring counter instruction, RDPMC. RDPMC loads the contents of the PerfCtrn register specified by the ECX register (which contains the MSR number for the performance counter), into the EDX register and the EAX register. The high-32 bits are loaded into EDX, and the low-32 bits are loaded into EAX. RDPMC can be executed only at CPL=0, unless system software enables use of the instruction at all privilege levels. RDPMC can be enabled for use at all privilege levels by setting CR4.PCE (the performance-monitor counter-enable bit) to 1. When CR4.PCE is 0, attempts to execute RDPMC result in a general-protection exception (#GP) if CPL is higher than 0.

The performance counters can also be read and written by system software running at CPL=0 using the RDMSR and WRMSR instructions, respectively. Writing the performance counters can be useful if software wants to count a specific number of events, and then trigger an interrupt when that count is reached. An interrupt can be triggered when a performance counter overflows. Software should use the WRMSR instruction to load the count as a two’s-complement negative number into the performance counter. This causes the counter to overflow after counting the appropriate number of times.

The performance counters are not guaranteed to produce identical measurements each time they are used to measure a particular instruction sequence, and they should not be used to take measurements of very-small instruction-sequences. The RDPMC instruction is not serializing, and it can be executed out-of-order with respect to other instructions around it. Even when bound by serializing instructions, the system environment at the time the instruction is executed can cause events to be counted before the counter value is loaded into EDX:EAX.
**Using Performance Counters**

Performance counts are started and stopped in a PerfCtr register by setting the corresponding PerfEvtSeln.EN bit to 1. Counting is stopped by clearing PerfEvtSeln.EN to 0. Software must initialize the remaining PerfEvtSel fields with the appropriate setup information prior to or at the same time EN is set. Counting begins when the WRMSR instruction that sets PerfEvtSeln.EN to 1 completes execution. Likewise, counting stops when the WRMSR instruction that clears PerfEvtSeln.EN to 0 completes execution.

**Counter Overflow**

Some processor implementations support an interrupt-on-overflow capability that allows an interrupt to occur when one of the PerfCtr registers overflows. The source and type of interrupt is implementation dependent. Some implementations cause a debug interrupt to occur, while others make use of the local APIC to specify the interrupt vector and trigger the interrupt when an overflow occurs. Software controls the triggering of an interrupt by setting or clearing the PerfEvtSeln.INT bit.

If system software makes use of the interrupt-on-overflow capability, an interrupt handler must be provided that can record information relevant to the counter overflow. Prior to returning from the interrupt handler, the performance counter can be re-initialized to its previous state so that another interrupt occurs when the appropriate number of events are counted.
Appendix E

Programming the MTRR and PAT

Introduction

The AMD Athlon™ processor includes a set of memory type and range registers (MTRRs) to control cacheability and access to specified memory regions. The processor also includes the Page Address Table for defining attributes of pages. This chapter documents the use and capabilities of this feature.

The purpose of the MTRRs is to provide system software with the ability to manage the memory mapping of the hardware. Both the BIOS software and operating systems utilize this capability. The AMD Athlon processor’s implementation is compatible with the Pentium® II. Prior to the MTRR mechanism, chipsets usually provided this capability.

Memory Type Range Register (MTRR) Mechanism

The memory type and range registers allow the processor to determine cacheability of various memory locations prior to bus access and to optimize access to the memory system. The AMD Athlon processor implements the MTRR programming model in a manner compatible with Pentium II.
There are two types of address ranges: fixed and variable (see Figure 12 on page 245). For each address range, there is a memory type. For each 4K, 16K or 64K segment within the first 1 Mbyte of memory, there is one fixed address MTRR. The fixed address ranges all exist in the first 1 Mbyte. There are eight variable address ranges above 1 Mbytes. Each is programmed to a specific memory starting address, size and alignment. If a variable range overlaps the lower 1 MByte and the fixed MTRRs are enabled, then the fixed-memory type dominates.

The address regions have the following priority with respect to each other:

1. Fixed address ranges
2. Variable address ranges
3. Default memory type (UC at reset)
Figure 12. MTRR Mapping of Physical Memory
Memory Types

Five standard memory types are defined by the AMD Athlon processor: writethrough (WT), writeback (WB), write-protect (WP), write-combining (WC), and uncacheable (UC). These are described in Table 12.

Table 12. Memory Type Encodings

<table>
<thead>
<tr>
<th>Type Number</th>
<th>Type Name</th>
<th>Type Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>UC—Uncacheable</td>
<td>Uncacheable for reads or writes. Cannot be combined. Must be non-speculative for reads or writes.</td>
</tr>
<tr>
<td>01h</td>
<td>WC—Write-Combining</td>
<td>Uncacheable for reads or writes. Can be combined. Can be speculative for reads. Writes can never be speculative.</td>
</tr>
<tr>
<td>04h</td>
<td>WT—Writethrough</td>
<td>Reads allocate on a miss, but only to the S-state. Writes do not allocate on a miss and, for a hit, writes update the cached entry and main memory.</td>
</tr>
<tr>
<td>05h</td>
<td>WP—Write-Protect</td>
<td>WP is functionally the same as the WT memory type, except stores do not actually modify cached data and do not cause an exception.</td>
</tr>
<tr>
<td>06h</td>
<td>WB—Writeback</td>
<td>Reads will allocate on a miss, and will allocate to: S-state if returned with a ReadDataShared command. M-state if returned with a ReadDataDirty command. Writes allocate to the M state, if the read allows the line to be marked E.</td>
</tr>
</tbody>
</table>

MTRR Capability Register Format

The MTRR capability register is a read-only register that defines the specific MTRR capability of the processor and is defined as follows.

Figure 13. MTRR Capability Register Format

For the AMD Athlon processor, the MTRR capability register should contain 0508h (write-combining, fixed MTRRs supported, and eight variable MTRRs defined).
**MTRR Default Type Register Format.** The MTRR default type register is defined as follows.

![MTRR Default Type Register Format](image)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>MTRRs Enabled</td>
<td>11</td>
</tr>
<tr>
<td>FE</td>
<td>Fixed Range Enabled</td>
<td>10</td>
</tr>
<tr>
<td>Type</td>
<td>Default Memory Type</td>
<td>7–0</td>
</tr>
</tbody>
</table>

**Figure 14. MTRR Default Type Register Format**

- **E**  MTRRs are enabled when set. All MTRRs (both fixed and variable range) are disabled when clear, and all of physical memory is mapped as uncacheable memory (reset state = 0).

- **FE**  Fixed-range MTRRs are enabled when set. All MTRRs are disabled when clear. When the fixed-range MTRRs are enabled and an overlap occurs with a variable-range MTRR, the fixed-range MTRR takes priority (reset state = 0).

- **Type**  Defines the default memory type (reset state = 0). See Table 13 for more details.
Note that if two or more variable memory ranges match then the interactions are defined as follows:

1. If the memory types are identical, then that memory type is used.

2. If one or more of the memory types is UC, the UC memory type is used.

3. If one or more of the memory types is WT and the only other matching memory type is WB then the WT memory type is used.

4. Otherwise, if the combination of memory types is not listed above then the behavior of the processor is undefined.

**MTRR Overlapping**

The Intel documentation (P6/PII) states that the mapping of large pages into regions that are mapped with differing memory types can result in undefined behavior. However, testing shows that these processors decompose these large pages into 4-Kbyte pages.

When a large page (2 Mbytes/4 Mbytes) mapping covers a region that contains more than one memory type (as mapped by the MTRRs), the AMD Athlon processor does not suppress the caching of that large page mapping and only caches the mapping for just that 4-Kbyte piece in the 4-Kbyte TLB. Therefore, the AMD Athlon processor does not decompose large pages under these conditions. The fixed range MTRRs are
not affected by this issue, only the variable range (and MTRR DefType) registers are affected.

**Page Attribute Table (PAT)**

The Page Attribute Table (PAT) is an extension of the page table entry format, which allows the specification of memory types to regions of physical memory based on the linear address. The PAT provides the same functionality as MTRRs with the flexibility of the page tables. It provides the operating systems and applications to determine the desired memory type for optimal performance. PAT support is detected in the feature flags (bit 16) of the CPUID instruction.

**MSR Access**

The PAT is located in a 64-bit MSR at location 277h. It is illustrated in Figure 15. Each of the eight PA<sub>n</sub> fields can contain the memory type encodings as described in Table 12 on page 246. An attempt to write an undefined memory type encoding into the PAT will generate a GP fault.

![Page Attribute Table (MSR 277h)](image)

*Figure 15. Page Attribute Table (MSR 277h)*
**Accessing the PAT**

A 3-bit index consisting of the PATi, PCD, and PWT bits of the page table entry, is used to select one of the seven PAT register fields to acquire the memory type for the desired page (PATi is defined as bit 7 for 4-Kbyte PTEs and bit 12 for PDEs which map to 2-Mbyte or 4-Mbyte pages). The memory type from the PAT is used instead of the PCD and PWT for the effective memory type.

A 2-bit index consisting of PCD and PWT bits of the page table entry is used to select one of four PAT register fields when PAE (page address extensions) is enabled, or when the PDE doesn’t describe a large page. In the latter case, the PATi bit for a PTE (bit 7) corresponds to the page size bit in a PDE. Therefore, the OS should only use PA0-3 when setting the memory type for a page table that is also used as a page directory. See Table 14.

**Table 14. PATi 3-Bit Encodings**

<table>
<thead>
<tr>
<th>PATi</th>
<th>PCD</th>
<th>PWT</th>
<th>PAT Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

**MTRRs and PAT**

The processor contains MTRRs as described earlier which provide a limited way of assigning memory types to specific regions. However, the page tables allow memory types to be assigned to the pages used for linear to physical translation.

The memory type as defined by PAT and MTRRs are combined to determine the effective memory type as listed in Table 15 and Table 16. Shaded areas indicate reserved settings.
## Table 15. Effective Memory Type Based on PAT and MTRRs

<table>
<thead>
<tr>
<th>PAT Memory Type</th>
<th>MTRR Memory Type</th>
<th>Effective Memory Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>UC</td>
<td>WB, WT, WP, WC</td>
<td>UC-Page</td>
</tr>
<tr>
<td></td>
<td>UC</td>
<td>UC-MTRR</td>
</tr>
<tr>
<td>WC</td>
<td>x</td>
<td>WC</td>
</tr>
<tr>
<td>WT</td>
<td>WB, WT</td>
<td>WT</td>
</tr>
<tr>
<td></td>
<td>UC</td>
<td>UC</td>
</tr>
<tr>
<td></td>
<td>WC</td>
<td>CD</td>
</tr>
<tr>
<td></td>
<td>WP</td>
<td>CD</td>
</tr>
<tr>
<td>WP</td>
<td>WB, WP</td>
<td>WP</td>
</tr>
<tr>
<td></td>
<td>UC</td>
<td>UC-MTRR</td>
</tr>
<tr>
<td></td>
<td>WC, WT</td>
<td>CD</td>
</tr>
<tr>
<td>WB</td>
<td>WB</td>
<td>WB</td>
</tr>
<tr>
<td></td>
<td>UC</td>
<td>UC</td>
</tr>
<tr>
<td></td>
<td>WC</td>
<td>WC</td>
</tr>
<tr>
<td></td>
<td>WT</td>
<td>WT</td>
</tr>
<tr>
<td></td>
<td>WP</td>
<td>WP</td>
</tr>
</tbody>
</table>

Notes:

1. UC-MTRR indicates that the UC attribute came from the MTRRs and that the processor caches should not be probed for performance reasons.
2. UC-Page indicates that the UC attribute came from the page tables and that the processor caches must be probed due to page aliasing.
3. All reserved combinations default to CD.
Table 16. Final Output Memory Types

<table>
<thead>
<tr>
<th>Input Memory Type</th>
<th>Output Memory Type</th>
<th>AMD-751™</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RdMem</td>
<td>WrMem</td>
</tr>
<tr>
<td>● ● UC</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● CD</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● WC</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● WT</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● WP</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● WB</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● -</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● UC</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● CD</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● WC</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● WT</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● WP</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● WB</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● -</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● UC</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● CD</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● WC</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● WT</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● WP</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● WB</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● -</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>● ● UC</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
### Table 16. Final Output Memory Types (Continued)

<table>
<thead>
<tr>
<th>Input Memory Type</th>
<th>Output Memory Type</th>
<th>AMD-751™</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RdMem</td>
<td>WrMem</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RdMem</td>
<td>WrMem</td>
<td>Effective. MType</td>
<td>forceCD</td>
</tr>
<tr>
<td>●</td>
<td>●</td>
<td>CD</td>
<td>-</td>
</tr>
<tr>
<td>●</td>
<td>●</td>
<td>WC</td>
<td>-</td>
</tr>
<tr>
<td>●</td>
<td>●</td>
<td>WT</td>
<td>-</td>
</tr>
<tr>
<td>●</td>
<td>●</td>
<td>WP</td>
<td>-</td>
</tr>
<tr>
<td>●</td>
<td>●</td>
<td>WB</td>
<td>-</td>
</tr>
<tr>
<td>●</td>
<td>●</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

**Notes:**

1. WP is not functional for RdMem/WrMem.
2. ForceCD must cause the MTRR memory type to be ignored in order to avoid x’s.
3. D-I should always be WP because the BIOS will only program RdMem-WrIO for WP. CD is forced to preserve the write-protect intent.
4. Since cached IO lines cannot be copied back to IO, the processor forces WB to WT to prevent cached IO from going dirty.
5. ForceCD. The memory type is forced CD due to (1) CR0[CD]=1, (2) memory type is for the ITLB and the I-Cache is disabled or for the DTLB and the D-Cache is disabled, (3) when clean victims must be written back and RdIO and WrIO and WT, WB, or WP, or (4) access to Local APIC space.
6. The processor does not support this memory type.
MTRR Fixed-Range Register Format

Table 17 lists the memory segments defined in each of the MTRR fixed-range registers. (See also Table 13 on page 248).

<table>
<thead>
<tr>
<th>Address Range (in hexadecimal)</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:56</td>
<td></td>
</tr>
<tr>
<td>55:48</td>
<td></td>
</tr>
<tr>
<td>47:40</td>
<td></td>
</tr>
<tr>
<td>39:32</td>
<td></td>
</tr>
<tr>
<td>31:24</td>
<td></td>
</tr>
<tr>
<td>23:16</td>
<td></td>
</tr>
<tr>
<td>15:8</td>
<td></td>
</tr>
<tr>
<td>7:0</td>
<td></td>
</tr>
<tr>
<td>70000-7FFFF</td>
<td>MTRR_fix64K_00000</td>
</tr>
<tr>
<td>9C000-9FFFF</td>
<td>MTRR_fix16K_80000</td>
</tr>
<tr>
<td>BC000-BFFFF</td>
<td>MTRR_fix16K_A0000</td>
</tr>
<tr>
<td>C7000-C7FFF</td>
<td>MTRR_fix4K_C0000</td>
</tr>
<tr>
<td>CF000-CFFF</td>
<td>MTRR_fix4K_C8000</td>
</tr>
<tr>
<td>D7000-D7FFF</td>
<td>MTRR_fix4K_D0000</td>
</tr>
<tr>
<td>DF000-DDFFF</td>
<td>MTRR_fix4K_D8000</td>
</tr>
<tr>
<td>E7000-E7FFF</td>
<td>MTRR_fix4K_E0000</td>
</tr>
<tr>
<td>EF000-EFFFF</td>
<td>MTRR_fix4K_E8000</td>
</tr>
<tr>
<td>F7000-F7FFF</td>
<td>MTRR_fix4K_F0000</td>
</tr>
<tr>
<td>FF000-FFFF</td>
<td>MTRR_fix4K_F8000</td>
</tr>
</tbody>
</table>
Variable-Range MTRRs

A variable MTRR can be programmed to start at address 0000_0000h because the fixed MTRRs always override the variable ones. However, it is recommended not to create an overlap.

The upper two variable MTRRs should not be used by the BIOS and are reserved for operating system use.

Variable-Range MTRR Register Format

The variable address range is power of 2 sized and aligned. The range of supported sizes is from $2^{12}$ to $2^{36}$ in powers of 2. The AMD Athlon processor does not implement $A[35:32]$.

Figure 16. MTRRphysBase

Note: A software attempt to write to reserved bits will generate a general protection exception.

- **Physical Base**: Specifies a 24-bit value which is extended by 12 bits to form the base address of the region defined in the register pair.
- **Type**: Table 13 on page 248.
Figure 17. **MTRRphysMask** Register Format

*Note: A software attempt to write to reserved bits will generate a general protection exception.*

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Mask</td>
<td>24-Bit Mask</td>
<td>35–12</td>
</tr>
<tr>
<td>V</td>
<td>Variable Range Register Pair Enabled</td>
<td>11</td>
</tr>
</tbody>
</table>

(V = 0 at reset)

Physical Mask 24-Bit Mask

V Enables the register pair when set (V = 0 at reset).

Mask values can represent discontinuous ranges (when the mask defines a lower significant bit as zero and a higher significant bit as one). In a discontinuous range, the memory area not mapped by the mask value is set to the default type. Discontinuous ranges should not be used.

The range that is mapped by the variable-range MTRR register pair must meet the following range size and alignment rule:

- Each defined memory range must have a size equal to \(2^n\) (11 < \(n\) < 36).
- The base address for the address pair must be aligned to a similar \(2^n\) boundary.

An example of a variable MTRR pair is as follows:

To map the address range from 8 Mbytes (0080_0000h) to 16 Mbytes (00FF_FFFFh) as writeback memory, the base register should be loaded with 80_0006h, and the mask should be loaded with FFF8_00800h.
**MTRR MSR Format**

This table defines the model-specific registers related to the memory type range register implementation. All MTRRs are defined to be 64 bits.

<table>
<thead>
<tr>
<th>Register Address</th>
<th>Register Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0FEh</td>
<td>MTRRcap</td>
<td>See “MTRR Capability Register Format” on page 246.</td>
</tr>
<tr>
<td>200h</td>
<td>MTRR Base0</td>
<td>See “MTRRphysBasen Register Format” on page 255.</td>
</tr>
<tr>
<td>201h</td>
<td>MTRR Mask0</td>
<td>See “MTRRphysMaskn Register Format” on page 256.</td>
</tr>
<tr>
<td>202h</td>
<td>MTRR Base1</td>
<td></td>
</tr>
<tr>
<td>203h</td>
<td>MTRR Mask1</td>
<td></td>
</tr>
<tr>
<td>204h</td>
<td>MTRR Base2</td>
<td></td>
</tr>
<tr>
<td>205h</td>
<td>MTRR Mask2</td>
<td></td>
</tr>
<tr>
<td>206h</td>
<td>MTRR Base3</td>
<td></td>
</tr>
<tr>
<td>207h</td>
<td>MTRR Mask3</td>
<td></td>
</tr>
<tr>
<td>208h</td>
<td>MTRR Base4</td>
<td></td>
</tr>
<tr>
<td>209h</td>
<td>MTRR Mask4</td>
<td></td>
</tr>
<tr>
<td>20Ah</td>
<td>MTRR Base5</td>
<td></td>
</tr>
<tr>
<td>208h</td>
<td>MTRR Mask5</td>
<td></td>
</tr>
<tr>
<td>20Ch</td>
<td>MTRR Base6</td>
<td></td>
</tr>
<tr>
<td>20Dh</td>
<td>MTRR Mask6</td>
<td></td>
</tr>
<tr>
<td>20Eh</td>
<td>MTRR Base7</td>
<td></td>
</tr>
<tr>
<td>20Fh</td>
<td>MTRR Mask7</td>
<td></td>
</tr>
<tr>
<td>250h</td>
<td>MTRRFIX64k_00000</td>
<td></td>
</tr>
<tr>
<td>258h</td>
<td>MTRRFIX16k_80000</td>
<td></td>
</tr>
<tr>
<td>259h</td>
<td>MTRRFIX16k_A0000</td>
<td></td>
</tr>
<tr>
<td>268h</td>
<td>MTRRFIX4k_C0000</td>
<td></td>
</tr>
<tr>
<td>269h</td>
<td>MTRRFIX4k_C8000</td>
<td>See “MTRR Fixed-Range Register Format” on page 254.</td>
</tr>
<tr>
<td>26Ah</td>
<td>MTRRFIX4k_D0000</td>
<td></td>
</tr>
<tr>
<td>268h</td>
<td>MTRRFIX4k_D8000</td>
<td></td>
</tr>
<tr>
<td>26Ch</td>
<td>MTRRFIX4k_E0000</td>
<td></td>
</tr>
<tr>
<td>26Dh</td>
<td>MTRRFIX4k_E8000</td>
<td></td>
</tr>
<tr>
<td>26Eh</td>
<td>MTRRFIX4k_F0000</td>
<td></td>
</tr>
<tr>
<td>26Fh</td>
<td>MTRRFIX4k_F8000</td>
<td></td>
</tr>
<tr>
<td>2FFh</td>
<td>MTRRdefType</td>
<td>See “MTRR Default Type Register Format” on page 247.</td>
</tr>
</tbody>
</table>
Appendix F

Instruction Dispatch and Execution Resources/Timing

This chapter describes the MacroOPs generated by each decoded instruction, along with the relative static execution latencies of these groups of operations. Tables 19 through 25 starting on page 261 define the following instructions: integer, MMX, MMX extensions, floating-point, 3DNow!, and 3DNow! extensions, and new instructions available with 3DNow! Professional.

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

- **`reg8`**—byte integer register defined by instruction byte(s) or bits 5, 4, and 3 of the modR/M byte
- **`mreg8`**—byte integer register defined by bits 2, 1, and 0 of the modR/M byte
- **`reg16/32`**—word and doubleword integer register defined by instruction byte(s) or bits 5, 4, and 3 of the modR/M byte
- **`mreg16/32`**—word and doubleword integer register defined by bits 2, 1, and 0 of the modR/M byte
- **`mem8`**—byte memory location
- **`mem16/32`**—word or doubleword memory location
- **`mem32/48`**—doubleword or 6-byte memory location
- **`mem48`**—48-bit integer value in memory
- **`mem64`**—64-bit value in memory
- **`mem128`**—128-bit value in memory
- imm8/16/32—8-bit, 16-bit or 32-bit immediate value
- disp8—8-bit displacement value
- disp16/32—16-bit or 32-bit displacement value
- disp32/48—32-bit or 48-bit displacement value
- eXX—register width depending on the operand size
- mem32real—32-bit floating-point value in memory
- mem64real—64-bit floating-point value in memory
- mem80real—80-bit floating-point value in memory
- mmreg—MMX/3DNow! register
- mmreg1—MMX/3DNow! register defined by bits 5, 4, and 3 of the modR/M byte
- mmreg2—MMX/3DNow! register defined by bits 2, 1, and 0 of the modR/M byte
- xmmreg—XMM register
- xmmreg1—XMM register defined by bits 5, 4, and 3 of the modR/M byte
- xmmreg2—XMM register defined by bits 2, 1, and 0 of the modR/M byte

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

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

The fifth column lists the type of instruction decode—DirectPath or VectorPath (see “DirectPath Decoder” and “VectorPath Decoder” under “Early Decoding” on page 207 for more information). The AMD Athlon processor enhanced decode logic can process three instructions per clock.

The FPU, MMX, and 3DNow! instruction tables have an additional column that lists the possible FPU execution pipelines available for use by any particular DirectPath decoded operation. Typically, VectorPath instructions require more than one execution pipe resource.
The sixth column lists the static execution latency. The static execution latency is defined as the number of clocks it takes to execute an instruction, or, more directly, the time it takes to execute the serially-dependent sequence of OPs that comprise each instruction. It is assumed that the instruction is an L1 hit that has already been fetched, decoded, and the operations loaded into the scheduler. It is the best case scenario which assumes no other instructions executing in the processor. The following format is used to describe the static execution latency:

- $x$—singular clock count
- $x$–$y$—possible latency range from $x$ to $y$ clocks
- $x/y$—$x$ equals the 16-bit timing and $y$ equals the 32-bit timing
- $y$: latency from address register operand(s)
- ~—clock count is not available

### Table 19. Integer Instructions

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA</td>
<td></td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>AAD imm8</td>
<td>D3h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>AAM imm8</td>
<td>D4h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>AAS</td>
<td>D8h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>ADC mreg8, reg8</td>
<td>10h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC mem8, reg8</td>
<td>10h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC mreg16/32, reg16/32</td>
<td>11h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC mem16/32, reg16/32</td>
<td>11h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is $20+3(n)$ for $n \geq 2$.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC reg8, mreg8</td>
<td>12h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC reg8, mem8</td>
<td>12h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC reg16/32, mreg16/32</td>
<td>13h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC reg16/32, mem16/32</td>
<td>13h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC AL, imm8</td>
<td>14h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC EAX, imm16/32</td>
<td>15h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC mreg8, imm8</td>
<td>80h</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC mem8, imm8</td>
<td>80h</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC mreg16/32, imm16/32</td>
<td>81h</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC mem16/32, imm16/32</td>
<td>81h</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC mreg16/32, imm8 (sign extended)</td>
<td>83h</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADC mem16/32, imm8 (sign extended)</td>
<td>83h</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD mreg8, reg8</td>
<td>00h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD mem8, reg8</td>
<td>00h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD mreg16/32, reg16/32</td>
<td>01h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD mem16/32, reg16/32</td>
<td>01h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD reg8, mreg8</td>
<td>02h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD reg8, mem8</td>
<td>02h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD reg16/32, mreg16/32</td>
<td>03h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD reg16/32, mem16/32</td>
<td>03h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD AL, imm8</td>
<td>04h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD EAX, imm16/32</td>
<td>05h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD mreg8, imm8</td>
<td>80h</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD mem8, imm8</td>
<td>80h</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD mreg16/32, imm16/32</td>
<td>81h</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is \(20 + (3^n)\) for \(n \geq 2\).
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
# Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD mem16/32, imm16/32</td>
<td>81h</td>
<td>mm-000-xxx</td>
<td>DirectPath 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD mreg16/32, imm8 (sign extended)</td>
<td>83h</td>
<td>11-000-xxx</td>
<td>DirectPath 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD mem16/32, imm8 (sign extended)</td>
<td>83h</td>
<td>mm-000-xxx</td>
<td>DirectPath 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND mreg8, reg8</td>
<td>20h</td>
<td>11-xxx-xxx</td>
<td>DirectPath 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND mem8, reg8</td>
<td>20h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND mreg16/32, reg16/32</td>
<td>21h</td>
<td>11-xxx-xxx</td>
<td>DirectPath 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND mem16/32, reg16/32</td>
<td>21h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND reg8, mreg8</td>
<td>22h</td>
<td>11-xxx-xxx</td>
<td>DirectPath 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND reg8, mem8</td>
<td>22h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND reg16/32, mreg16/32</td>
<td>23h</td>
<td>11-xxx-xxx</td>
<td>DirectPath 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND reg16/32, mem16/32</td>
<td>23h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND AL, imm8</td>
<td>24h</td>
<td></td>
<td>DirectPath 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND EAX, imm16/32</td>
<td>25h</td>
<td></td>
<td>DirectPath 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND mreg8, imm8</td>
<td>80h</td>
<td>11-100-xxx</td>
<td>DirectPath 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND mem8, imm8</td>
<td>80h</td>
<td>mm-100-xxx</td>
<td>DirectPath 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND mreg16/32, imm16/32</td>
<td>81h</td>
<td>11-100-xxx</td>
<td>DirectPath 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND mem16/32, imm16/32</td>
<td>81h</td>
<td>mm-100-xxx</td>
<td>DirectPath 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND mreg16/32, imm8 (sign extended)</td>
<td>83h</td>
<td>11-100-xxx</td>
<td>DirectPath 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AND mem16/32, imm8 (sign extended)</td>
<td>83h</td>
<td>mm-100-xxx</td>
<td>DirectPath 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARPL mreg16, reg16</td>
<td>63h</td>
<td>11-xxx-xxx</td>
<td>VectorPath 15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARPL mem16, reg16</td>
<td>63h</td>
<td>mm-xxx-xxx</td>
<td>VectorPath 19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOUND reg16/32, mem16/32:mem16/32</td>
<td>62h</td>
<td>mm-xxx-xxx</td>
<td>VectorPath 6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSF reg16/32, mreg16/32</td>
<td>0Fh</td>
<td>BCh</td>
<td>VectorPath 8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSF reg16/32, mem16/32</td>
<td>0Fh</td>
<td>BCh</td>
<td>VectorPath 12/11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BSR reg16/32, mreg16/32</td>
<td>0Fh</td>
<td>BDh</td>
<td>VectorPath 10</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSR reg16/32, mem16/32</td>
<td>0Fh</td>
<td>BDh</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>14/13</td>
<td></td>
</tr>
<tr>
<td>BSWAP EAX</td>
<td>0Fh</td>
<td>C8h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>BSWAP ECX</td>
<td>0Fh</td>
<td>C9h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>BSWAP EDX</td>
<td>0Fh</td>
<td>CAh</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>BSWAP EBX</td>
<td>0Fh</td>
<td>CBh</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>BSWAP ESP</td>
<td>0Fh</td>
<td>CCh</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>BSWAP EBP</td>
<td>0Fh</td>
<td>CDh</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>BSWAP ESI</td>
<td>0Fh</td>
<td>CEh</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>BSWAP EDI</td>
<td>0Fh</td>
<td>CFh</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>BT mreg16/32, reg16/32</td>
<td>0Fh</td>
<td>A3h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>BT mem16/32, reg16/32</td>
<td>0Fh</td>
<td>A3h</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>BT mreg16/32, imm8</td>
<td>0Fh</td>
<td>BAh</td>
<td>11-100-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>BT mem16/32, imm8</td>
<td>0Fh</td>
<td>BAh</td>
<td>mm-100-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>BTC mreg16/32, reg16/32</td>
<td>0Fh</td>
<td>BBh</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>BTC mem16/32, reg16/32</td>
<td>0Fh</td>
<td>BBh</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>BTC mreg16/32, imm8</td>
<td>0Fh</td>
<td>BAh</td>
<td>11-111-xxx</td>
<td>VectorPath</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>BTC mem16/32, imm8</td>
<td>0Fh</td>
<td>BAh</td>
<td>mm-111-xxx</td>
<td>VectorPath</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>BTR mreg16/32, reg16/32</td>
<td>0Fh</td>
<td>B3h</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>BTR mem16/32, reg16/32</td>
<td>0Fh</td>
<td>B3h</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>BTR mreg16/32, imm8</td>
<td>0Fh</td>
<td>BAh</td>
<td>11-110-xxx</td>
<td>VectorPath</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>BTR mem16/32, imm8</td>
<td>0Fh</td>
<td>BAh</td>
<td>mm-110-xxx</td>
<td>VectorPath</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>BTS mreg16/32, reg16/32</td>
<td>0Fh</td>
<td>ABh</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>BTS mem16/32, reg16/32</td>
<td>0Fh</td>
<td>ABh</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>BTS mreg16/32, imm8</td>
<td>0Fh</td>
<td>BAh</td>
<td>11-101-xxx</td>
<td>VectorPath</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>BTS mem16/32, imm8</td>
<td>0Fh</td>
<td>BAh</td>
<td>mm-101-xxx</td>
<td>VectorPath</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL full pointer</td>
<td>9Ah</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>CALL near imm16/32</td>
<td>E8h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>3 2</td>
<td></td>
</tr>
<tr>
<td>CALL near mreg32 (indirect)</td>
<td>FFh</td>
<td>11-010-xxx</td>
<td></td>
<td>VectorPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CALL near mem32 (indirect)</td>
<td>FFh</td>
<td>mm-010-xxx</td>
<td></td>
<td>VectorPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CALL mem16:16/32</td>
<td>FFh</td>
<td>11-011-xxx</td>
<td></td>
<td>VectorPath</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>CBW/CWDE</td>
<td>98h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CLC</td>
<td>F8h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CLD</td>
<td>FCh</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CLI</td>
<td>FAh</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CLTS</td>
<td>0Fh 06h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>CMCC</td>
<td>F5h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CMOVA/CMOVNBE reg16/32, reg16/32</td>
<td>0Fh</td>
<td>47h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CMOVA/CMOVNBE reg16/32, mem16/32</td>
<td>0Fh</td>
<td>47h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMOVAE/CMOVNB/CMOVNC reg16/32, mem16/32</td>
<td>0Fh</td>
<td>43h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CMOVAE/CMOVNB/CMOVNC mem16/32, mem16/32</td>
<td>0Fh</td>
<td>43h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMOVB/CMOVC/CMOVNAE reg16/32, reg16/32</td>
<td>0Fh</td>
<td>42h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CMOVB/CMOVC/CMOVNAE mem16/32, reg16/32</td>
<td>0Fh</td>
<td>42h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMOVBE/CMOVNA reg16/32, reg16/32</td>
<td>0Fh</td>
<td>46h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CMOVBE/CMOVNA reg16/32, mem16/32</td>
<td>0Fh</td>
<td>46h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMOVE/CMOVZ reg16/32, reg16/32</td>
<td>0Fh</td>
<td>44h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CMOVE/CMOVZ reg16/32, mem16/32</td>
<td>0Fh</td>
<td>44h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMOVG/CMOVNLE reg16/32, reg16/32</td>
<td>0Fh</td>
<td>4Fh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CMOVG/CMOVNLE reg16/32, mem16/32</td>
<td>0Fh</td>
<td>4Fh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMOVGE/CMOVNL reg16/32, reg16/32</td>
<td>0Fh</td>
<td>4Dh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMOVGE/CMOVNLE reg16/32, mem16/32</td>
<td>0Fh</td>
<td>4Dh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMOVL/CMOVNGE reg16/32, reg16/32</td>
<td>0Fh</td>
<td>4Ch</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CMOVL/CMOVNGE reg16/32, mem16/32</td>
<td>0Fh</td>
<td>4Ch</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMOVLE/CMOVNGE reg16/32, reg16/32</td>
<td>0Fh</td>
<td>4Eh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CMOVLE/CMOVNGE reg16/32, mem16/32</td>
<td>0Fh</td>
<td>4Eh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMOVNE/CMOVNZ reg16/32, reg16/32</td>
<td>0Fh</td>
<td>45h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CMOVNE/CMOVNZ reg16/32, mem16/32</td>
<td>0Fh</td>
<td>45h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMOVNO reg16/32, reg16/32</td>
<td>0Fh</td>
<td>41h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CMOVNO reg16/32, mem16/32</td>
<td>0Fh</td>
<td>41h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMOVNP/CMOVPO reg16/32, reg16/32</td>
<td>0Fh</td>
<td>4Bh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CMOVNP/CMOVPO reg16/32, mem16/32</td>
<td>0Fh</td>
<td>4Bh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMOVNS reg16/32, reg16/32</td>
<td>0Fh</td>
<td>49h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CMOVNS reg16/32, mem16/32</td>
<td>0Fh</td>
<td>49h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMODO reg16/32, reg16/32</td>
<td>0Fh</td>
<td>40h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CMODO reg16/32, mem16/32</td>
<td>0Fh</td>
<td>40h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMOVPE/CMOVPE reg16/32, reg16/32</td>
<td>0Fh</td>
<td>4Ah</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CMOVPE/CMOVPE reg16/32, mem16/32</td>
<td>0Fh</td>
<td>4Ah</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMOVSP reg16/32, reg16/32</td>
<td>0Fh</td>
<td>48h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CMOVSP reg16/32, mem16/32</td>
<td>0Fh</td>
<td>48h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMP mreg8, reg8</td>
<td>38h</td>
<td></td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CMP mreg8, reg8</td>
<td>38h</td>
<td></td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMP mreg16/32, reg16/32</td>
<td>39h</td>
<td></td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CMP mreg16/32, reg16/32</td>
<td>39h</td>
<td></td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CMP reg8, mreg8</td>
<td>3Ah</td>
<td></td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CMP reg8, mem8</td>
<td>3Ah</td>
<td></td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
### Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMP reg16/32, mreg16/32</td>
<td>3Bh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP reg16/32, mem16/32</td>
<td>3Bh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP AL, imm8</td>
<td>3Ch</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP EAX, imm16/32</td>
<td>3Dh</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP mreg8, imm8</td>
<td>80h</td>
<td>11-111-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP mem8, imm8</td>
<td>80h</td>
<td>mm-111-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP mreg16/32, imm16/32</td>
<td>81h</td>
<td>11-111-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP mem16/32, imm16/32</td>
<td>81h</td>
<td>mm-111-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP mreg16/32, imm8 (sign extended)</td>
<td>83h</td>
<td>11-111-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMP mem16/32, imm8 (sign extended)</td>
<td>83h</td>
<td>mm-111-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPSB mem8,mem8</td>
<td>A6h</td>
<td></td>
<td>VectorPath</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPSW mem16, mem32</td>
<td>A7h</td>
<td></td>
<td>VectorPath</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPSD mem32, mem32</td>
<td>A7h</td>
<td></td>
<td>VectorPath</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMPXCHG mreg8, reg8</td>
<td>0Fh</td>
<td>B0h</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>CMPXCHG mem8, reg8</td>
<td>0Fh</td>
<td>B0h</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>CMPXCHG mreg16/32, reg16/32</td>
<td>0Fh</td>
<td>B1h</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>CMPXCHG mem16/32, reg16/32</td>
<td>0Fh</td>
<td>B1h</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>CMPXCHG8B mem64</td>
<td>0Fh</td>
<td>C7h</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>CPUID</td>
<td>0Fh</td>
<td>A2h</td>
<td></td>
<td>VectorPath</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>CWD/CDQ</td>
<td>99h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAA</td>
<td>27h</td>
<td></td>
<td>VectorPath</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DAS</td>
<td>2Fh</td>
<td></td>
<td>VectorPath</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC EAX</td>
<td>48h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC ECX</td>
<td>49h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEC EDX</td>
<td>4Ah</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC EBX</td>
<td>4Bh</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>DEC ESP</td>
<td>4Ch</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>DEC EBP</td>
<td>4Dh</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>DEC ESI</td>
<td>4Eh</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>DEC EDI</td>
<td>4Fh</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>DEC mreg8</td>
<td>FEh</td>
<td>11-001-xxx</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>DEC mem8</td>
<td>FEh</td>
<td>mm-001-xxx</td>
<td></td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>DEC mreg16/32</td>
<td>FFh</td>
<td>11-001-xxx</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>DEC mem16/32</td>
<td>FFh</td>
<td>mm-001-xxx</td>
<td></td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>DIV mreg8</td>
<td>F6h</td>
<td>11-110-xxx</td>
<td></td>
<td>VectorPath</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>DIV AL, mem8</td>
<td>F6h</td>
<td>mm-110-xxx</td>
<td></td>
<td>VectorPath</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>DIV mreg16/32</td>
<td>F7h</td>
<td>11-110-xxx</td>
<td></td>
<td>VectorPath</td>
<td>24/40</td>
<td></td>
</tr>
<tr>
<td>DIV EAX, mem16/32</td>
<td>F7h</td>
<td>mm-110-xxx</td>
<td></td>
<td>VectorPath</td>
<td>24/40</td>
<td></td>
</tr>
<tr>
<td>ENTER</td>
<td>C8h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>13/17/19/22</td>
<td>6</td>
</tr>
<tr>
<td>IDIV mreg8</td>
<td>F6h</td>
<td>11-111-xxx</td>
<td></td>
<td>VectorPath</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>IDIV mem8</td>
<td>F6h</td>
<td>mm-111-xxx</td>
<td></td>
<td>VectorPath</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>IDIV mreg16/32</td>
<td>F7h</td>
<td>11-111-xxx</td>
<td></td>
<td>VectorPath</td>
<td>26/42</td>
<td></td>
</tr>
<tr>
<td>IDIV EAX, mem16/32</td>
<td>F7h</td>
<td>mm-111-xxx</td>
<td></td>
<td>VectorPath</td>
<td>27/43</td>
<td></td>
</tr>
<tr>
<td>IMUL reg16/32, imm16/32</td>
<td>69h</td>
<td>11-xxx-xxx</td>
<td></td>
<td>VectorPath</td>
<td>4/5</td>
<td></td>
</tr>
<tr>
<td>IMUL reg16/32, mreg16/32, imm16/32</td>
<td>69h</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>4/5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMUL reg16/32, mem16/32, imm16/32</td>
<td>69h</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>7/8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMUL reg16/32, imm8 (sign extended)</td>
<td>6Bh</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMUL reg16/32, mreg16/32, imm8 (signed)</td>
<td>6Bh</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>4/5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMUL reg16/32, mem16/32, imm8 (signed)</td>
<td>6Bh</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMUL mreg8</td>
<td>F6h</td>
<td>11-101-xxx</td>
<td></td>
<td>VectorPath</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
### Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMUL AX, AL, mem8</td>
<td>F6h</td>
<td>mm-101-xxx</td>
<td>VectorPath</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMUL mreg16/32</td>
<td>F7h</td>
<td>11-101-xxx</td>
<td>VectorPath</td>
<td>5/6</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>IMUL EDX:EAX, EAX, mem16/32</td>
<td>F7h</td>
<td>mm-101-xxx</td>
<td>VectorPath</td>
<td>8/9</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>IMUL reg16/32, mreg16/32</td>
<td>0Fh</td>
<td>AFh</td>
<td>VectorPath</td>
<td>3/4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMUL reg16/32, mem16/32</td>
<td>0Fh</td>
<td>AFh</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>6/7</td>
<td></td>
</tr>
<tr>
<td>IN AL, imm8</td>
<td>E4h</td>
<td>VectorPath</td>
<td>~</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN AX, imm8</td>
<td>E5h</td>
<td>VectorPath</td>
<td>~</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN EAX, imm8</td>
<td>E5h</td>
<td>VectorPath</td>
<td>~</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN AL, DX</td>
<td>EDh</td>
<td>VectorPath</td>
<td>~</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN AX, DX</td>
<td>EDh</td>
<td>VectorPath</td>
<td>~</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IN EAX, DX</td>
<td>EDh</td>
<td>VectorPath</td>
<td>~</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC EAX</td>
<td>40h</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC ECX</td>
<td>41h</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC EDX</td>
<td>42h</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC EBX</td>
<td>43h</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC ESP</td>
<td>44h</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC EBP</td>
<td>45h</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC ESI</td>
<td>46h</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC EDI</td>
<td>47h</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC mreg8</td>
<td>FEh</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC mem8</td>
<td>FEh</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC mreg16/32</td>
<td>FFh</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INC mem16/32</td>
<td>FFh</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INVD</td>
<td>0Fh</td>
<td>08h</td>
<td>VectorPath</td>
<td>~</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INVLPD</td>
<td>0Fh</td>
<td>01h</td>
<td>mm-111-xxx</td>
<td>VectorPath</td>
<td>106</td>
<td></td>
</tr>
</tbody>
</table>

**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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
### Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>JO short disp8</td>
<td>70h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JNO short disp8</td>
<td>71h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JB/JNAE/JC short disp8</td>
<td>72h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JNB/JAC/JNC short disp8</td>
<td>73h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JZ/JE short disp8</td>
<td>74h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JNZ/JNE short disp8</td>
<td>75h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JBE/JNA short disp8</td>
<td>76h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JNBE/JA short disp8</td>
<td>77h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JS short disp8</td>
<td>78h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JNS short disp8</td>
<td>79h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JP/JPE short disp8</td>
<td>7Ah</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JNP/JPO short disp8</td>
<td>7Bh</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JL/JNE short disp8</td>
<td>7Ch</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JNL/JGE short disp8</td>
<td>7Dh</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JLE/JNG short disp8</td>
<td>7Eh</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JNLE/JG short disp8</td>
<td>7Fh</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JCXZ/JEC short disp8</td>
<td>E3h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>JO near disp16/32</td>
<td>0Fh</td>
<td>80h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JNO near disp16/32</td>
<td>0Fh</td>
<td>81h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JB/JNAE near disp16/32</td>
<td>0Fh</td>
<td>82h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JNB/JAE near disp16/32</td>
<td>0Fh</td>
<td>83h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JZ/JE near disp16/32</td>
<td>0Fh</td>
<td>84h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JNZ/JNE near disp16/32</td>
<td>0Fh</td>
<td>85h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JBE/JNA near disp16/32</td>
<td>0Fh</td>
<td>86h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JNBE/JA near disp16/32</td>
<td>0Fh</td>
<td>87h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 "Use 32-Bit LEA Rather than 16-Bit LEA Instruction" on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
### Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>JS near disp16/32</td>
<td>0Fh</td>
<td>88h</td>
<td>DirectPath 1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>JNS near disp16/32</td>
<td>0Fh</td>
<td>89h</td>
<td>DirectPath 1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>JP/JPE near disp16/32</td>
<td>0Fh</td>
<td>8Ah</td>
<td>DirectPath 1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>JNP/JPO near disp16/32</td>
<td>0Fh</td>
<td>88h</td>
<td>DirectPath 1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>JL/JNGE near disp16/32</td>
<td>0Fh</td>
<td>8Ch</td>
<td>DirectPath 1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>JNL/JGE near disp16/32</td>
<td>0Fh</td>
<td>8Dh</td>
<td>DirectPath 1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>JLE/JNG near disp16/32</td>
<td>0Fh</td>
<td>8Eh</td>
<td>DirectPath 1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>JNLE/JG near disp16/32</td>
<td>0Fh</td>
<td>8Fh</td>
<td>DirectPath 1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>JMP near disp16/32 (direct)</td>
<td>E9h</td>
<td></td>
<td>DirectPath 1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP far disp32/48 (direct)</td>
<td>EAh</td>
<td></td>
<td>VectorPath 16</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP disp8 (short)</td>
<td>EBh</td>
<td></td>
<td>DirectPath 1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP far mem32 (indirect)</td>
<td>EFh</td>
<td>mm-101-xxx</td>
<td>VectorPath 18</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP far mreg32 (indirect)</td>
<td>FFh</td>
<td>mm-101-xxx</td>
<td>VectorPath 18</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP near mreg16/32 (indirect)</td>
<td>FFh</td>
<td>11-100-xxx</td>
<td>DirectPath 1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JMP near mem16/32 (indirect)</td>
<td>FFh</td>
<td>mm-100-xxx</td>
<td>DirectPath 4</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAHF</td>
<td>9Fh</td>
<td></td>
<td>VectorPath 3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAR reg16/32, mreg16/32</td>
<td>0Fh</td>
<td>02h 11-xxx-xxx</td>
<td>VectorPath 23</td>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAR reg16/32, mem16/32</td>
<td>0Fh</td>
<td>02h mm-xxx-xxx</td>
<td>VectorPath 25</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDS reg16/32, mem32/48</td>
<td>C5h</td>
<td>mm-xxx-xxx</td>
<td>VectorPath 14</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEA reg16, mem16/32</td>
<td>8Dh</td>
<td>mm-xxx-xxx</td>
<td>VectorPath 3</td>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>LEA reg32, mem16/32</td>
<td>8Dh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath 2</td>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>LEAVE</td>
<td>C9h</td>
<td></td>
<td>VectorPath 3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LES reg16/32, mem32/48</td>
<td>C4h</td>
<td>mm-xxx-xxx</td>
<td>VectorPath 14</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LFS reg16/32, mem32/48</td>
<td>0Fh</td>
<td>B4h</td>
<td>VectorPath 14</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LGDT mem48</td>
<td>0Fh</td>
<td>01h mm-010-xxx</td>
<td>VectorPath 35</td>
<td>35</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for \( n \geq 2 \).
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>LGS reg16/32, mem32/48</td>
<td>0Fh</td>
<td>B5h</td>
<td></td>
<td>VectorPath</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>LIDT mem48</td>
<td>0Fh</td>
<td>01h</td>
<td>mm-011-xxx</td>
<td>VectorPath</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>LLDT mreg16</td>
<td>0Fh</td>
<td>00h</td>
<td>11-010-xxx</td>
<td>VectorPath</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>LLDT mem16</td>
<td>0Fh</td>
<td>00h</td>
<td>mm-010-xxx</td>
<td>VectorPath</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>LMSW mreg16</td>
<td>0Fh</td>
<td>01h</td>
<td>11-100-xxx</td>
<td>VectorPath</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>LMSW mem16</td>
<td>0Fh</td>
<td>01h</td>
<td>mm-100-xxx</td>
<td>VectorPath</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>LODSB AL, mem8</td>
<td>ACh</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>5, 8</td>
<td></td>
</tr>
<tr>
<td>LODSW AX, mem16</td>
<td>ADh</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>5, 8</td>
<td></td>
</tr>
<tr>
<td>LODSD EAX, mem32</td>
<td>ADh</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>4, 8</td>
<td></td>
</tr>
<tr>
<td>LOOP disp8</td>
<td>E2h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>LOOPE/LOOPZ disp8</td>
<td>E1h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>LOOPNE/LOOPNZ disp8</td>
<td>E0h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>LSL reg16/32, mreg16/32</td>
<td>0Fh</td>
<td>03h</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>LSL reg16/32, mem16/32</td>
<td>0Fh</td>
<td>03h</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>LSS reg16/32, mem32/48</td>
<td>0Fh</td>
<td>B2h</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>LTR mreg16</td>
<td>0Fh</td>
<td>00h</td>
<td>11-011-xxx</td>
<td>VectorPath</td>
<td>91</td>
<td></td>
</tr>
<tr>
<td>LTR mem16</td>
<td>0Fh</td>
<td>00h</td>
<td>mm-011-xxx</td>
<td>VectorPath</td>
<td>94</td>
<td></td>
</tr>
<tr>
<td>MOV mreg8, reg8</td>
<td>88h</td>
<td></td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV mem8, reg8</td>
<td>88h</td>
<td></td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MOV mreg16/32, reg16/32</td>
<td>89h</td>
<td></td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV mem16/32, reg16/32</td>
<td>89h</td>
<td></td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MOV reg8, mreg8</td>
<td>8Ah</td>
<td></td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV reg8, mem8</td>
<td>8Ah</td>
<td></td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MOV reg16/32, mreg16/32</td>
<td>8Bh</td>
<td></td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV reg16/32, mem16/32</td>
<td>8Bh</td>
<td></td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
### Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV mreg16, segment reg</td>
<td>8Ch</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV mem16, segment reg</td>
<td>8Ch</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV segment reg, mreg16</td>
<td>8Eh</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV segment reg, mem16</td>
<td>8Eh</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV AL, mem8</td>
<td>A0h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MOV EAX, mem16/32</td>
<td>A1h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MOV mem8, AL</td>
<td>A2h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MOV mem16/32, EAX</td>
<td>A3h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MOV AL, imm8</td>
<td>B0h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV CL, imm8</td>
<td>B1h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV DL, imm8</td>
<td>B2h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV BL, imm8</td>
<td>B3h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV AH, imm8</td>
<td>B4h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV CH, imm8</td>
<td>B5h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV DH, imm8</td>
<td>B6h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV BH, imm8</td>
<td>B7h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV EAX, imm16/32</td>
<td>B8h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV ECX, imm16/32</td>
<td>B9h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV EDX, imm16/32</td>
<td>BAh</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV EBX, imm16/32</td>
<td>BBh</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV ESP, imm16/32</td>
<td>BCb</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV EBP, imm16/32</td>
<td>BDh</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV ESI, imm16/32</td>
<td>BEh</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV EDI, imm16/32</td>
<td>BFh</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV mreg8, imm8</td>
<td>C6h</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
### Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV mem8, imm8</td>
<td>C6h</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV mreg16/32, imm16/32</td>
<td>C7h</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV mem16/32, imm16/32</td>
<td>C7h</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVSB mem8,mem8</td>
<td>A4h</td>
<td>VectorPath</td>
<td>5</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVSD mem16, mem16</td>
<td>A5h</td>
<td>VectorPath</td>
<td>5</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVSW mem32, mem32</td>
<td>A5h</td>
<td>VectorPath</td>
<td>5</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVZX reg16/32, mreg8</td>
<td>0Fh</td>
<td>BEh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOVZX reg16/32, mem8</td>
<td>0Fh</td>
<td>BEh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>MOVZX reg32, mreg16</td>
<td>0Fh</td>
<td>BFh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOVZX reg32, mem16</td>
<td>0Fh</td>
<td>BFh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>MOVZX reg16/32, mreg8</td>
<td>0Fh</td>
<td>B6h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOVZX reg16/32, mem8</td>
<td>0Fh</td>
<td>B6h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>MOVZX reg32, mreg16</td>
<td>0Fh</td>
<td>B7h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOVZX reg32, mem16</td>
<td>0Fh</td>
<td>B7h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>MUL mreg8</td>
<td>F6h</td>
<td>11-100-xxx</td>
<td>VectorPath</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL AL, mem8</td>
<td>F6h</td>
<td>mm-100-xx</td>
<td>VectorPath</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL mreg16</td>
<td>F7h</td>
<td>11-100-xxx</td>
<td>VectorPath</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL AX, mem16</td>
<td>F7h</td>
<td>mm-100-xx</td>
<td>VectorPath</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL mreg32</td>
<td>F7h</td>
<td>11-100-xxx</td>
<td>VectorPath</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL EAX, mem32</td>
<td>F7h</td>
<td>mm-100-xx</td>
<td>VectorPath</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEG mreg8</td>
<td>F6h</td>
<td>11-011-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEG mem8</td>
<td>F6h</td>
<td>mm-011-xx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEG mreg16/32</td>
<td>F7h</td>
<td>11-011-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEG mem16/32</td>
<td>F7h</td>
<td>mm-011-xx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOP (XCHG EAX, EAX)</td>
<td>90h</td>
<td>DirectPath</td>
<td>0</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
### Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT mreg8</td>
<td>F6h</td>
<td>11-010-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOT mem8</td>
<td>F6h</td>
<td>mm-010-xx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOT mreg16/32</td>
<td>F7h</td>
<td>11-010-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOT mem16/32</td>
<td>F7h</td>
<td>mm-010-xx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR mreg8, reg8</td>
<td>08h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR mem8, reg8</td>
<td>08h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR mreg16/32, reg16/32</td>
<td>09h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR mem16/32, reg16/32</td>
<td>09h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR reg8, mreg8</td>
<td>0Ah</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR reg8, mem8</td>
<td>0Ah</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR reg16/32, mreg16/32</td>
<td>0Bh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR reg16/32, mem16/32</td>
<td>0Bh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR AL, imm8</td>
<td>0Ch</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR EAX, imm16/32</td>
<td>0Dh</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR mreg8, imm8</td>
<td>80h</td>
<td>11-001-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR mem8, imm8</td>
<td>80h</td>
<td>mm-001-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR mreg16/32, imm16/32</td>
<td>81h</td>
<td>11-001-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR mem16/32, imm16/32</td>
<td>81h</td>
<td>mm-001-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR mreg16/32, imm8 (sign extended)</td>
<td>83h</td>
<td>11-001-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OR mem16/32, imm8 (sign extended)</td>
<td>83h</td>
<td>mm-001-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT imm8, AL</td>
<td>E6h</td>
<td></td>
<td>VectorPath</td>
<td>~</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT imm8, AX</td>
<td>E7h</td>
<td></td>
<td>VectorPath</td>
<td>~</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT imm8, EAX</td>
<td>E7h</td>
<td></td>
<td>VectorPath</td>
<td>~</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT DX, AL</td>
<td>EEh</td>
<td></td>
<td>VectorPath</td>
<td>~</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT DX, AX</td>
<td>EFh</td>
<td></td>
<td>VectorPath</td>
<td>~</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
### Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>OUT DX, EAX</td>
<td>EFh</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>POP ES</td>
<td>07h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>POP SS</td>
<td>17h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>POP DS</td>
<td>1Fh</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>POP FS</td>
<td>0Fh A1h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>POP GS</td>
<td>0Fh A9h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>POP EAX</td>
<td>58h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>POP ECX</td>
<td>59h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>POP EDX</td>
<td>5Ah</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>POP EBX</td>
<td>58h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>POP ESP</td>
<td>5Ch</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>POP EBP</td>
<td>5Dh</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>POP ESI</td>
<td>5Eh</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>POP EDI</td>
<td>5Fh</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>POP mreg 16/32</td>
<td>8Fh 11-000-xxx</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>POP mem 16/32</td>
<td>8Fh mm-000-xxx</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>POPA/POPAD</td>
<td>61h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>7/6</td>
<td></td>
</tr>
<tr>
<td>POPF/POPFD</td>
<td>9Dh</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>PUSH ES</td>
<td>06h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>3 2</td>
<td></td>
</tr>
<tr>
<td>PUSH CS</td>
<td>0Eh</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PUSH FS</td>
<td>0Fh A0h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PUSH GS</td>
<td>0Fh A8h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PUSH SS</td>
<td>16h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PUSH DS</td>
<td>1Eh</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>3 2</td>
<td></td>
</tr>
<tr>
<td>PUSH EAX</td>
<td>50h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
### Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH ECX</td>
<td>51h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>PUSH EDX</td>
<td>52h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>PUSH EBX</td>
<td>53h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>PUSH ESP</td>
<td>54h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>PUSH EBP</td>
<td>55h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>PUSH ESI</td>
<td>56h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>PUSH EDI</td>
<td>57h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>PUSH imm8</td>
<td>6Ah</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>PUSH imm16/32</td>
<td>68h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>PUSH mreg16/32</td>
<td>FFh</td>
<td>11-110-xxx</td>
<td>VectorPath</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSH mem16/32</td>
<td>FFh</td>
<td>mm-110-xxx</td>
<td>VectorPath</td>
<td>3</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>PUSHA/PUSHAD</td>
<td>60h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>PUSHF/PUSHFD</td>
<td>9Ch</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>RCL mreg8, imm8</td>
<td>C0h</td>
<td>11-010-xxx</td>
<td>DirectPath</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCL mem8, imm8</td>
<td>C0h</td>
<td>mm-010-xxx</td>
<td>VectorPath</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCL mreg16/32, imm8</td>
<td>C1h</td>
<td>11-010-xxx</td>
<td>DirectPath</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCL mem16/32, imm8</td>
<td>C1h</td>
<td>mm-010-xxx</td>
<td>VectorPath</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCL mreg8, 1</td>
<td>D0h</td>
<td>11-010-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCL mem8, 1</td>
<td>D0h</td>
<td>mm-010-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCL mreg16/32, 1</td>
<td>D1h</td>
<td>11-010-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCL mem16/32, 1</td>
<td>D1h</td>
<td>mm-010-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCL mreg8, CL</td>
<td>D2h</td>
<td>11-010-xxx</td>
<td>DirectPath</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCL mem8, CL</td>
<td>D2h</td>
<td>mm-010-xxx</td>
<td>VectorPath</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCL mreg16/32, CL</td>
<td>D3h</td>
<td>11-010-xxx</td>
<td>DirectPath</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RCL mem16/32, CL</td>
<td>D3h</td>
<td>mm-010-xxx</td>
<td>VectorPath</td>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCR mreg8, imm8</td>
<td>C0h</td>
<td>11-011-xxx</td>
<td>DirectPath</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>RCR mem8, imm8</td>
<td>C0h</td>
<td>mm-011-xxx</td>
<td>VectorPath</td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>RCR mreg16/32, imm8</td>
<td>C1h</td>
<td>11-011-xxx</td>
<td>DirectPath</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>RCR mem16/32, imm8</td>
<td>C1h</td>
<td>mm-011-xxx</td>
<td>VectorPath</td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>RCR mreg8, 1</td>
<td>D0h</td>
<td>11-011-xxx</td>
<td>DirectPath</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>RCR mem8, 1</td>
<td>D0h</td>
<td>mm-011-xxx</td>
<td>DirectPath</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>RCR mreg16/32, 1</td>
<td>D1h</td>
<td>11-011-xxx</td>
<td>DirectPath</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>RCR mem16/32, 1</td>
<td>D1h</td>
<td>mm-011-xxx</td>
<td>DirectPath</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>RCR mreg8, CL</td>
<td>D2h</td>
<td>11-011-xxx</td>
<td>DirectPath</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>RCR mem8, CL</td>
<td>D2h</td>
<td>mm-011-xxx</td>
<td>VectorPath</td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>RCR mreg16/32, CL</td>
<td>D3h</td>
<td>11-011-xxx</td>
<td>DirectPath</td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>RCR mem16/32, CL</td>
<td>D3h</td>
<td>mm-011-xxx</td>
<td>VectorPath</td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>RDMSR</td>
<td>0Fh</td>
<td>32h</td>
<td>VectorPath</td>
<td>~</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDPMC</td>
<td>0Fh</td>
<td>33h</td>
<td>VectorPath</td>
<td>~</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RDTSC</td>
<td>0Fh</td>
<td>31h</td>
<td>VectorPath</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RET near imm16</td>
<td>C2h</td>
<td></td>
<td>VectorPath</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RET near</td>
<td>C3h</td>
<td></td>
<td>VectorPath</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RET far imm16</td>
<td>CAh</td>
<td></td>
<td>VectorPath</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RET far</td>
<td>CBh</td>
<td></td>
<td>VectorPath</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROL mreg8, imm8</td>
<td>C0h</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ROL mem8, imm8</td>
<td>C0h</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ROL mreg16/32, imm8</td>
<td>C1h</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ROL mem16/32, imm8</td>
<td>C1h</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ROL mreg8, 1</td>
<td>D0h</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROL mem8, 1</td>
<td>D0h</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
### Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROL mreg16/32, 1</td>
<td>D1h</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROL mem16/32, 1</td>
<td>D1h</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROL mreg8, CL</td>
<td>D2h</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ROL mem8, CL</td>
<td>D2h</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ROL mreg16/32, CL</td>
<td>D3h</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ROL mem16/32, CL</td>
<td>D3h</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ROR mreg8, imm8</td>
<td>C0h</td>
<td>11-001-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ROR mem8, imm8</td>
<td>C0h</td>
<td>mm-001-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ROR mreg16/32, imm8</td>
<td>C1h</td>
<td>11-001-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ROR mem16/32, imm8</td>
<td>C1h</td>
<td>mm-001-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ROR mreg8, 1</td>
<td>D0h</td>
<td>11-001-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROR mem8, 1</td>
<td>D0h</td>
<td>mm-001-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROR mreg16/32, 1</td>
<td>D1h</td>
<td>11-001-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROR mem16/32, 1</td>
<td>D1h</td>
<td>mm-001-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROR mreg8, CL</td>
<td>D2h</td>
<td>11-001-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ROR mem8, CL</td>
<td>D2h</td>
<td>mm-001-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ROR mreg16/32, CL</td>
<td>D3h</td>
<td>11-001-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>ROR mem16/32, CL</td>
<td>D3h</td>
<td>mm-001-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SAHF</td>
<td>9Eh</td>
<td></td>
<td>VectorPath</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAR mreg8, imm8</td>
<td>C0h</td>
<td>11-111-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SAR mem8, imm8</td>
<td>C0h</td>
<td>mm-111-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SAR mreg16/32, imm8</td>
<td>C1h</td>
<td>11-111-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SAR mem16/32, imm8</td>
<td>C1h</td>
<td>mm-111-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SAR mreg8, 1</td>
<td>D0h</td>
<td>11-111-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAR mem8, 1</td>
<td>D0h</td>
<td>mm-111-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAR mreg16/32, 1</td>
<td>D1h</td>
<td>11-111-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAR mem16/32, 1</td>
<td>D1h</td>
<td>mm-111-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SAR mreg8, CL</td>
<td>D2h</td>
<td>11-111-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SAR mem8, CL</td>
<td>D2h</td>
<td>mm-111-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SAR mreg16/32, CL</td>
<td>D3h</td>
<td>11-111-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SAR mem16/32, CL</td>
<td>D3h</td>
<td>mm-111-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SBB mreg8, reg8</td>
<td>18h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB mem8, reg8</td>
<td>18h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB mreg16/32, reg16/32</td>
<td>19h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB mem16/32, reg16/32</td>
<td>19h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB reg8, mreg8</td>
<td>1Ah</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB reg8, mem8</td>
<td>1Ah</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB reg16/32, mreg16/32</td>
<td>1Bh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB reg16/32, mem16/32</td>
<td>1Bh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB AL, imm8</td>
<td>1Ch</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB EAX, imm16/32</td>
<td>1Dh</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB mreg8, imm8</td>
<td>80h</td>
<td>11-011-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB mem8, imm8</td>
<td>80h</td>
<td>mm-011-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB mreg16/32, imm16/32</td>
<td>81h</td>
<td>11-011-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB mem16/32, imm16/32</td>
<td>81h</td>
<td>mm-011-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB mreg16/32, imm8 (sign extended)</td>
<td>83h</td>
<td>11-011-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SBB mem16/32, imm8 (sign extended)</td>
<td>83h</td>
<td>mm-011-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCASB AL, mem8</td>
<td>AEh</td>
<td>VectorPath</td>
<td>4</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCASW AX, mem16</td>
<td>AFh</td>
<td>VectorPath</td>
<td>4</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCASD EAX, mem32</td>
<td>AFh</td>
<td>VectorPath</td>
<td>4</td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
### Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETO mreg8</td>
<td>0Fh</td>
<td>90h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SETO mem8</td>
<td>0Fh</td>
<td>90h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SETNO mreg8</td>
<td>0Fh</td>
<td>91h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SETNO mem8</td>
<td>0Fh</td>
<td>91h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SETB/SETC/SETNAE mreg8</td>
<td>0Fh</td>
<td>92h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SETB/SETC/SETNAE mem8</td>
<td>0Fh</td>
<td>92h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SETAE/SETNB/SETNC mreg8</td>
<td>0Fh</td>
<td>93h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SETAE/SETNB/SETNC mem8</td>
<td>0Fh</td>
<td>93h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SETE/SETE mreg8</td>
<td>0Fh</td>
<td>94h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SETE/SETE mem8</td>
<td>0Fh</td>
<td>94h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SETNE/SETEZ mreg8</td>
<td>0Fh</td>
<td>95h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SETNE/SETEZ mem8</td>
<td>0Fh</td>
<td>95h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SETBE/SETENA mreg8</td>
<td>0Fh</td>
<td>96h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SETBE/SETENA mem8</td>
<td>0Fh</td>
<td>96h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SETA/SETNEBE mreg8</td>
<td>0Fh</td>
<td>97h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SETA/SETNEBE mem8</td>
<td>0Fh</td>
<td>97h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SETS mreg8</td>
<td>0Fh</td>
<td>98h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SETS mem8</td>
<td>0Fh</td>
<td>98h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SETNS mreg8</td>
<td>0Fh</td>
<td>99h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SETNS mem8</td>
<td>0Fh</td>
<td>99h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SETP/SETPE mreg8</td>
<td>0Fh</td>
<td>9Ah</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SETP/SETPE mem8</td>
<td>0Fh</td>
<td>9Ah</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SETNP/SETPO mreg8</td>
<td>0Fh</td>
<td>9Bh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SETNP/SETPO mem8</td>
<td>0Fh</td>
<td>9Bh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SETL/SETNGE mreg8</td>
<td>0Fh</td>
<td>9Ch</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
### Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETL/SETNGE mem8</td>
<td>0Fh</td>
<td>9Ch</td>
<td>mm-xxx-xxx</td>
<td>DirectPath 1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SETGE/SETNL mreg8</td>
<td>0Fh</td>
<td>9Dh</td>
<td>11-xxx-xxx</td>
<td>DirectPath 1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SETGE/SETNL mem8</td>
<td>0Fh</td>
<td>9Dh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath 3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SETLE/SETNG mreg8</td>
<td>0Fh</td>
<td>9Eh</td>
<td>11-xxx-xxx</td>
<td>DirectPath 1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SETLE/SETNG mem8</td>
<td>0Fh</td>
<td>9Eh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath 3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SETG/SETNLLE mreg8</td>
<td>0Fh</td>
<td>9Fh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath 1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SETG/SETNLLE mem8</td>
<td>0Fh</td>
<td>9Fh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath 3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SGDT mem48</td>
<td>0Fh</td>
<td>01h</td>
<td>mm-000-xxx</td>
<td>VectorPath 17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIDT mem48</td>
<td>0Fh</td>
<td>01h</td>
<td>mm-001-xxx</td>
<td>VectorPath 17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHL/SAL mreg8, imm8</td>
<td>C0h</td>
<td>11-100-xxx</td>
<td>DirectPath 1</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHL/SAL mem8, imm8</td>
<td>C0h</td>
<td>mm-100-xxx</td>
<td>DirectPath 4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHL/SAL mreg16/32, imm8</td>
<td>C1h</td>
<td>11-100-xxx</td>
<td>DirectPath 1</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHL/SAL mem16/32, imm8</td>
<td>C1h</td>
<td>mm-100-xxx</td>
<td>DirectPath 4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHL/SAL mreg8, 1</td>
<td>D0h</td>
<td>11-100-xxx</td>
<td>DirectPath 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHL/SAL mem8, 1</td>
<td>D0h</td>
<td>mm-100-xxx</td>
<td>DirectPath 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHL/SAL mreg16/32, 1</td>
<td>D1h</td>
<td>11-100-xxx</td>
<td>DirectPath 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHL/SAL mem16/32, 1</td>
<td>D1h</td>
<td>mm-100-xxx</td>
<td>DirectPath 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHL/SAL mreg8, CL</td>
<td>D2h</td>
<td>11-100-xxx</td>
<td>DirectPath 1</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHL/SAL mem8, CL</td>
<td>D2h</td>
<td>mm-100-xxx</td>
<td>DirectPath 4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHL/SAL mreg16/32, CL</td>
<td>D3h</td>
<td>11-100-xxx</td>
<td>DirectPath 1</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHL/SAL mem16/32, CL</td>
<td>D3h</td>
<td>mm-100-xxx</td>
<td>DirectPath 4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHR mreg8, imm8</td>
<td>C0h</td>
<td>11-101-xxx</td>
<td>DirectPath 1</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHR mem8, imm8</td>
<td>C0h</td>
<td>mm-101-xxx</td>
<td>DirectPath 4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHR mreg16/32, imm8</td>
<td>C1h</td>
<td>11-101-xxx</td>
<td>DirectPath 1</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHR mem16/32, imm8</td>
<td>C1h</td>
<td>mm-101-xxx</td>
<td>DirectPath 4</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 "Use 32-Bit LEA Rather than 16-Bit LEA Instruction" on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHR mreg8, 1</td>
<td>D0h</td>
<td>11-101-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHR mem8, 1</td>
<td>D0h</td>
<td>mm-101-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHR mreg16/32, 1</td>
<td>D1h</td>
<td>11-101-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHR mem16/32, 1</td>
<td>D1h</td>
<td>mm-101-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHR mreg8, CL</td>
<td>D2h</td>
<td>11-101-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SHR mem8, CL</td>
<td>D2h</td>
<td>mm-101-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SHR mreg16/32, CL</td>
<td>D3h</td>
<td>11-101-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SHR mem16/32, CL</td>
<td>D3h</td>
<td>mm-101-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SHLD mreg16/32, reg16/32, imm8</td>
<td>0Fh</td>
<td>A4h</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>SHLD mem16/32, reg16/32, imm8</td>
<td>0Fh</td>
<td>A4h</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>SHLD mreg16/32, reg16/32, Cl</td>
<td>0Fh</td>
<td>A5h</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>SHLD mem16/32, reg16/32, Cl</td>
<td>0Fh</td>
<td>A5h</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>SHRD mreg16/32, reg16/32, imm8</td>
<td>0Fh</td>
<td>ACh</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>SHRD mem16/32, reg16/32, imm8</td>
<td>0Fh</td>
<td>ACh</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>SHRD mreg16/32, reg16/32, Cl</td>
<td>0Fh</td>
<td>ADh</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>SHRD mem16/32, reg16/32, Cl</td>
<td>0Fh</td>
<td>ADh</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>SLDT mreg16</td>
<td>0Fh</td>
<td>00h</td>
<td>11-000-xxx</td>
<td>VectorPath</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>SLDT mem16</td>
<td>0Fh</td>
<td>00h</td>
<td>mm-000-xxx</td>
<td>VectorPath</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>SMSW mreg16</td>
<td>0Fh</td>
<td>01h</td>
<td>11-100-xxx</td>
<td>VectorPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SMSW mem16</td>
<td>0Fh</td>
<td>01h</td>
<td>mm-100-xxx</td>
<td>VectorPath</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>STC</td>
<td>F9h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STD</td>
<td>FDh</td>
<td></td>
<td>VectorPath</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STI</td>
<td>FBh</td>
<td></td>
<td>VectorPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STOSB mem8, AL</td>
<td>AAh</td>
<td></td>
<td>VectorPath</td>
<td>4</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>STOSW mem16, AX</td>
<td>Abh</td>
<td></td>
<td>VectorPath</td>
<td>4</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is $20 + (3*n)$ for $n \geq 2$.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>STOSD mem32, EAX</td>
<td>ABh</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>STR mreg16</td>
<td>0Fh</td>
<td>00h</td>
<td>11-001-xxx</td>
<td>VectorPath</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>STR mem16</td>
<td>0Fh</td>
<td>00h</td>
<td>mm-001-xxx</td>
<td>VectorPath</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>SUB mreg8, reg8</td>
<td>28h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SUB mem8, reg8</td>
<td>28h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SUB mreg16/32, reg16/32</td>
<td>29h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SUB mem16/32, reg16/32</td>
<td>29h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SUB reg8, mreg8</td>
<td>2Ah</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SUB reg8, mem8</td>
<td>2Ah</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SUB reg16/32, mreg16/32</td>
<td>2Bh</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SUB reg16/32, mem16/32</td>
<td>2Bh</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SUB AL, imm8</td>
<td>2Ch</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SUB EAX, imm16/32</td>
<td>2Dh</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SUB mreg8, imm8</td>
<td>80h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SUB mem8, imm8</td>
<td>80h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SUB mreg16/32, imm16/32</td>
<td>81h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SUB mem16/32, imm16/32</td>
<td>81h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SUB mreg16/32, imm8 (sign extended)</td>
<td>83h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SUB mem16/32, imm8 (sign extended)</td>
<td>83h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SYSCALL</td>
<td>0Fh</td>
<td>05h</td>
<td></td>
<td>VectorPath</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>SYSENTER</td>
<td>0Fh</td>
<td>34h</td>
<td></td>
<td>VectorPath</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>SYSEXIT</td>
<td>0Fh</td>
<td>35h</td>
<td></td>
<td>VectorPath</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>SYSRET</td>
<td>0Fh</td>
<td>07h</td>
<td></td>
<td>VectorPath</td>
<td>~</td>
<td></td>
</tr>
<tr>
<td>TEST mreg8, reg8</td>
<td>84h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>TEST mem8, reg8</td>
<td>84h</td>
<td></td>
<td></td>
<td>DirectPath</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 "Use 32-Bit LEA Rather than 16-Bit LEA Instruction" on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
### Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST mreg16/32, reg16/32</td>
<td>85h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEST mem16/32, reg16/32</td>
<td>85h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEST AL, imm8</td>
<td>A8h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEST EAX, imm16/32</td>
<td>A9h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEST mreg8, imm8</td>
<td>F6h</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEST mem8, imm8</td>
<td>F6h</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEST mreg16/32, imm16/32</td>
<td>F7h</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TEST mem16/32, imm16/32</td>
<td>F7h</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VERR mreg16</td>
<td>0Fh</td>
<td>00h</td>
<td>11-100-xxx</td>
<td>VectorPath</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>VERR mem16</td>
<td>0Fh</td>
<td>00h</td>
<td>mm-100-xxx</td>
<td>VectorPath</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>VERW mreg16</td>
<td>0Fh</td>
<td>00h</td>
<td>11-101-xxx</td>
<td>VectorPath</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>VERW mem16</td>
<td>0Fh</td>
<td>00h</td>
<td>mm-101-xxx</td>
<td>VectorPath</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>WAIT</td>
<td>9Bh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>0</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>WBINVD</td>
<td>0Fh</td>
<td>09h</td>
<td>VectorPath</td>
<td>~</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRMSR</td>
<td>0Fh</td>
<td>30h</td>
<td>VectorPath</td>
<td>~</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XADD mreg8, reg8</td>
<td>0Fh</td>
<td>C0h</td>
<td>11-100-xxx</td>
<td>VectorPath</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>XADD mem8, reg8</td>
<td>0Fh</td>
<td>C0h</td>
<td>mm-100-xxx</td>
<td>VectorPath</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>XADD mreg16/32, reg16/32</td>
<td>0Fh</td>
<td>C1h</td>
<td>11-101-xxx</td>
<td>VectorPath</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>XADD mem16/32, reg16/32</td>
<td>0Fh</td>
<td>C1h</td>
<td>mm-101-xxx</td>
<td>VectorPath</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>XCHG reg8, mreg8</td>
<td>86h</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XCHG reg8, mem8</td>
<td>86h</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XCHG reg16/32, mreg16/32</td>
<td>87h</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XCHG reg16/32, mem16/32</td>
<td>87h</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XCHG EAX, EAX (NOP)</td>
<td>90h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>0</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>XCHG EAX, ECX</td>
<td>91h</td>
<td>VectorPath</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
Table 19. Integer Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCHG EAX, EDX</td>
<td>92h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>XCHG EAX, EBX</td>
<td>93h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>XCHG EAX, ESP</td>
<td>94h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>XCHG EAX, EBP</td>
<td>95h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>XCHG EAX, ESI</td>
<td>96h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>XCHG EAX, EDI</td>
<td>97h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>XLAT</td>
<td>D7h</td>
<td></td>
<td></td>
<td>VectorPath</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>XOR mreg8, reg8</td>
<td>30h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR mem8, reg8</td>
<td>30h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR mreg16/32, reg16/32</td>
<td>31h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR mem16/32, reg16/32</td>
<td>31h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR reg8, mreg8</td>
<td>32h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR reg8, mem8</td>
<td>32h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR reg16/32, mreg16/32</td>
<td>33h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR reg16/32, mem16/32</td>
<td>33h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR AL, imm8</td>
<td>34h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR EAX, imm16/32</td>
<td>35h</td>
<td></td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR mreg8, imm8</td>
<td>80h</td>
<td>11-110-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR mem8, imm8</td>
<td>80h</td>
<td>mm-110-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR mreg16/32, imm16/32</td>
<td>81h</td>
<td>11-110-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR mem16/32, imm16/32</td>
<td>81h</td>
<td>mm-110-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR mreg16/32, imm8 (sign extended)</td>
<td>83h</td>
<td>11-110-xxx</td>
<td>DirectPath</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR mem16/32, imm8 (sign extended)</td>
<td>83h</td>
<td>mm-110-xxx</td>
<td>DirectPath</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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. There is a lower latency for low half of products, and a higher latency for high half of product and/or setting of flags.
5. 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 “Use 32-Bit LEA Rather than 16-Bit LEA Instruction” on page 54.
6. Execution latencies for nesting levels 0/1/2/3. A general rule for latency is 20+(3*n) for n ≥ 2.
7. These instructions have an effective latency of that which is listed. They map to internal NOPs that can be executed at a rate of three per cycle and do not occupy execution resources.
8. The latency of repeated string instructions can be found in “Latency of Repeated String Instructions” on page 123.
Table 20. MMX™ Instructions

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Prefix Byte(s)</th>
<th>First Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>FPU Pipe(s)</th>
<th>Execute Latency</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMMS</td>
<td>0Fh</td>
<td>77h</td>
<td>DirectPath</td>
<td>FADD/FMUL/FSTORE</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MOVQ mmreg, reg32</td>
<td>0Fh</td>
<td>6Eh</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>3, 4</td>
<td></td>
</tr>
<tr>
<td>MOVQ mmreg, mem32</td>
<td>0Fh</td>
<td>6Eh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL/FSTORE</td>
<td>2, 3</td>
<td></td>
</tr>
<tr>
<td>MOVQ reg32, mmreg</td>
<td>0Fh</td>
<td>7Eh</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>5, 1, 4</td>
<td></td>
</tr>
<tr>
<td>MOVQ mem32, mmreg</td>
<td>0Fh</td>
<td>7Eh</td>
<td>mm-xxx-xxx</td>
<td>FSTORE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOVQ mmreg1, mmreg2</td>
<td>0Fh</td>
<td>6Fh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MOVQ mmreg, mem64</td>
<td>0Fh</td>
<td>6Fh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL/FSTORE</td>
<td>2, 3</td>
<td></td>
</tr>
<tr>
<td>MOVQ mmreg2, mmreg1</td>
<td>0Fh</td>
<td>7Fh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MOVQ mem64, mmreg</td>
<td>0Fh</td>
<td>7Fh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FSTORE</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PACKSSDW mmreg1, mmreg2</td>
<td>0Fh</td>
<td>6Bh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PACKSSDW mmreg, mem64</td>
<td>0Fh</td>
<td>6Bh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PACKSSWB mmreg1, mmreg2</td>
<td>0Fh</td>
<td>63h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PACKSSWB mmreg, mem64</td>
<td>0Fh</td>
<td>63h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PACKUSWB mmreg1, mmreg2</td>
<td>0Fh</td>
<td>67h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PACKUSWB mmreg, mem64</td>
<td>0Fh</td>
<td>67h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PADDDB mmreg1, mmreg2</td>
<td>0Fh</td>
<td>FCh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PADDDB mmreg, mem64</td>
<td>0Fh</td>
<td>FCh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PADDDB mmreg1, mmreg2</td>
<td>0Fh</td>
<td>FEh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PADDDB mmreg, mem64</td>
<td>0Fh</td>
<td>FEh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PADDDB mmreg1, mmreg2</td>
<td>0Fh</td>
<td>ECh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PADDDB mmreg, mem64</td>
<td>0Fh</td>
<td>ECh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PADDDB mmreg1, mmreg2</td>
<td>0Fh</td>
<td>EDh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PADDDB mmreg, mem64</td>
<td>0Fh</td>
<td>EDh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PADDUSB mmreg1, mmreg2</td>
<td>0Fh</td>
<td>DCh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PADDUSB mmreg, mem64</td>
<td>0Fh</td>
<td>DCh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PADDUSB mmreg1, mmreg2</td>
<td>0Fh</td>
<td>DDh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PADDUSB mmreg, mem64</td>
<td>0Fh</td>
<td>DDh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. Bits 2, 1, and 0 of the modR/M byte select the integer register.
2. The cycle count listed is purely for execution and does not take into account the time required by the instruction to access the Load/Store Unit. It is recommended that operations dependent on the result of this particular operation be pushed back by at least two additional cycles.
3. These instructions have an effective latency of that which is listed. 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.
4. The latency listed is the absolute minimum, while average latencies may be higher and are a function of internal pipeline conditions.
Table 20. MMX™ Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Prefix Byte(s)</th>
<th>First Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>FPU Pipe(s)</th>
<th>Execute Latency</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PADDW mmreg1, mmreg2</td>
<td>0Fh</td>
<td>FDh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PADDW mmreg, mem64</td>
<td>0Fh</td>
<td>FDh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PAND mmreg1, mmreg2</td>
<td>0Fh</td>
<td>DBh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PAND mmreg, mem64</td>
<td>0Fh</td>
<td>DBh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PANDN mmreg1, mmreg2</td>
<td>0Fh</td>
<td>DFh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PANDN mmreg, mem64</td>
<td>0Fh</td>
<td>DFh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PCMPEQB mmreg1, mmreg2</td>
<td>0Fh</td>
<td>74h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PCMPEQB mmreg, mem64</td>
<td>0Fh</td>
<td>74h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PCMPEQD mmreg1, mmreg2</td>
<td>0Fh</td>
<td>76h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PCMPEQD mmreg, mem64</td>
<td>0Fh</td>
<td>76h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PCMPGTQ mmreg1, mmreg2</td>
<td>0Fh</td>
<td>75h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PCMPGTQ mmreg, mem64</td>
<td>0Fh</td>
<td>75h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PCMGTB mmreg1, mmreg2</td>
<td>0Fh</td>
<td>64h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PCMGTB mmreg, mem64</td>
<td>0Fh</td>
<td>64h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PCMPGTQD mmreg1, mmreg2</td>
<td>0Fh</td>
<td>66h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PCMPGTQD mmreg, mem64</td>
<td>0Fh</td>
<td>66h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PCMPGTW mmreg1, mmreg2</td>
<td>0Fh</td>
<td>65h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PCMPGTW mmreg, mem64</td>
<td>0Fh</td>
<td>65h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PMADDWD mmreg1, mmreg2</td>
<td>0Fh</td>
<td>F5h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PMADDWD mmreg, mem64</td>
<td>0Fh</td>
<td>F5h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>PMULHW mmreg1, mmreg2</td>
<td>0Fh</td>
<td>E5h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PMULHW mmreg, mem64</td>
<td>0Fh</td>
<td>E5h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>PMULLW mmreg1, mmreg2</td>
<td>0Fh</td>
<td>D5h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PMULLW mmreg, mem64</td>
<td>0Fh</td>
<td>D5h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>POR mmreg1, mmreg2</td>
<td>0Fh</td>
<td>EBh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>POR mmreg, mem64</td>
<td>0Fh</td>
<td>EBh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PSLLD mmreg1, mmreg2</td>
<td>0Fh</td>
<td>F2h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. Bits 2, 1, and 0 of the modR/M byte select the integer register.
2. The cycle count listed is purely for execution and does not take into account the time required by the instruction to access the Load/Store Unit. It is recommended that operations dependent on the result of this particular operation be pushed back by at least two additional cycles.
3. These instructions have an effective latency of that which is listed. 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.
4. The latency listed is the absolute minimum, while average latencies may be higher and are a function of internal pipeline conditions.
### Table 20. MMX™ Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Prefix Byte(s)</th>
<th>First Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>FPU Pipe(s)</th>
<th>Execute Latency</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSLLD mmreg, mem64</td>
<td>0Fh</td>
<td>F2h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PSLLD mmreg, imm8</td>
<td>0Fh</td>
<td>72h</td>
<td>11-110-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSLLQ mmreg1, mmreg2</td>
<td>0Fh</td>
<td>F3h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSLLQ mmreg, mem64</td>
<td>0Fh</td>
<td>F3h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PSLLQ mmreg, imm8</td>
<td>0Fh</td>
<td>73h</td>
<td>11-110-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSLLW mmreg1, mmreg2</td>
<td>0Fh</td>
<td>F1h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSLLW mmreg, mem64</td>
<td>0Fh</td>
<td>71h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PSLLW mmreg, imm8</td>
<td>0Fh</td>
<td>E1h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSRAW mmreg1, mmreg2</td>
<td>0Fh</td>
<td>E1h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PSRAW mmreg, mem64</td>
<td>0Fh</td>
<td>71h</td>
<td>11-100-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSRAD mmreg1, mmreg2</td>
<td>0Fh</td>
<td>E2h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSRAD mmreg, mem64</td>
<td>0Fh</td>
<td>E2h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PSRAD mmreg, imm8</td>
<td>0Fh</td>
<td>72h</td>
<td>11-100-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSRLD mmreg1, mmreg2</td>
<td>0Fh</td>
<td>D2h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSRLD mmreg, mem64</td>
<td>0Fh</td>
<td>D2h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PSRLD mmreg, imm8</td>
<td>0Fh</td>
<td>72h</td>
<td>11-010-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSRLQ mmreg1, mmreg2</td>
<td>0Fh</td>
<td>D3h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSRLQ mmreg, mem64</td>
<td>0Fh</td>
<td>D3h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PSRLQ mmreg, imm8</td>
<td>0Fh</td>
<td>73h</td>
<td>11-010-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSRLW mmreg1, mmreg2</td>
<td>0Fh</td>
<td>D1h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSRLW mmreg, mem64</td>
<td>0Fh</td>
<td>D1h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PSRLW mmreg, imm8</td>
<td>0Fh</td>
<td>71h</td>
<td>11-010-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSUBB mmreg1, mmreg2</td>
<td>0Fh</td>
<td>F8h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSUBB mmreg, mem64</td>
<td>0Fh</td>
<td>F8h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PSUBD mmreg1, mmreg2</td>
<td>0Fh</td>
<td>FAh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSUBD mmreg, mem64</td>
<td>0Fh</td>
<td>FAh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

**Notes:**

1. Bits 2, 1, and 0 of the modR/M byte select the integer register.
2. The cycle count listed is purely for execution and does not take into account the time required by the instruction to access the Load/Store Unit. It is recommended that operations dependent on the result of this particular operation be pushed back by at least two additional cycles.
3. These instructions have an effective latency of that which is listed. 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.
4. The latency listed is the absolute minimum, while average latencies may be higher and are a function of internal pipeline conditions.
### Table 20. MMX™ Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Prefix Byte(s)</th>
<th>First Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>FPU Pipe(s)</th>
<th>Execute Latency</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSUBSB mmreg1, mmreg2</td>
<td>0Fh</td>
<td>E8h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSUBSB mmreg, mem64</td>
<td>0Fh</td>
<td>E8h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2 2</td>
<td></td>
</tr>
<tr>
<td>PSUBSW mmreg1, mmreg2</td>
<td>0Fh</td>
<td>E9h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSUBSW mmreg, mem64</td>
<td>0Fh</td>
<td>E9h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2 2</td>
<td></td>
</tr>
<tr>
<td>PSUSBW mmreg1, mmreg2</td>
<td>0Fh</td>
<td>D8h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSUSBW mmreg, mem64</td>
<td>0Fh</td>
<td>D8h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2 2</td>
<td></td>
</tr>
<tr>
<td>PUNPCKHBW mmreg1, mmreg2</td>
<td>0Fh</td>
<td>68h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PUNPCKHBW mmreg, mem64</td>
<td>0Fh</td>
<td>68h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2 2</td>
<td></td>
</tr>
<tr>
<td>PUNPCKHDQ mmreg1, mmreg2</td>
<td>0Fh</td>
<td>6Ah</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PUNPCKHDQ mmreg, mem64</td>
<td>0Fh</td>
<td>6Ah</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2 2</td>
<td></td>
</tr>
<tr>
<td>PUNPCKHWD mmreg1, mmreg2</td>
<td>0Fh</td>
<td>69h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PUNPCKHWD mmreg, mem64</td>
<td>0Fh</td>
<td>69h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2 2</td>
<td></td>
</tr>
<tr>
<td>PUNPCKLBW mmreg1, mmreg2</td>
<td>0Fh</td>
<td>60h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PUNPCKLBW mmreg, mem32</td>
<td>0Fh</td>
<td>60h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2 2</td>
<td></td>
</tr>
<tr>
<td>PUNPCKLDQ mmreg1, mmreg2</td>
<td>0Fh</td>
<td>62h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PUNPCKLDQ mmreg, mem32</td>
<td>0Fh</td>
<td>62h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2 2</td>
<td></td>
</tr>
<tr>
<td>PUNPCKLWD mmreg1, mmreg2</td>
<td>0Fh</td>
<td>61h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PUNPCKLWD mmreg, mem32</td>
<td>0Fh</td>
<td>61h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2 2</td>
<td></td>
</tr>
<tr>
<td>PXOR mmreg1, mmreg2</td>
<td>0Fh</td>
<td>EFh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PXOR mmreg, mem64</td>
<td>0Fh</td>
<td>EFh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2 2</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

1. Bits 2, 1, and 0 of the modR/M byte select the integer register.
2. The cycle count listed is purely for execution and does not take into account the time required by the instruction to access the Load/Store Unit. It is recommended that operations dependent on the result of this particular operation be pushed back by at least two additional cycles.
3. These instructions have an effective latency of that which is listed. 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.
4. The latency listed is the absolute minimum, while average latencies may be higher and are a function of internal pipeline conditions.
### Table 21. MMX™ Extensions

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Prefix Byte(s)</th>
<th>First Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>FPU Pipe(s)</th>
<th>Execute Latency</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASKMOVQ mmreg1, mmreg2</td>
<td>0Fh</td>
<td>F7h</td>
<td></td>
<td>VectorPath</td>
<td>FADD/FMUL/FSTORE</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>MOVNTQ mem64, mmreg</td>
<td>0Fh</td>
<td>E7h</td>
<td></td>
<td>DirectPath</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PAVGB mmreg1, mmreg2</td>
<td>0Fh</td>
<td>E0h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PAVGB mmreg, mem64</td>
<td>0Fh</td>
<td>E0h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PAVGW mmreg1, mmreg2</td>
<td>0Fh</td>
<td>E3h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PAVGW mmreg, mem64</td>
<td>0Fh</td>
<td>E3h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PEXTRW reg32, mmreg, imm8</td>
<td>0Fh</td>
<td>C5h</td>
<td></td>
<td>VectorPath</td>
<td>-</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>PINSRW mmreg, reg32, imm8</td>
<td>0Fh</td>
<td>C4h</td>
<td></td>
<td>VectorPath</td>
<td>-</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>PINSRW mmreg, mem16, imm8</td>
<td>0Fh</td>
<td>C4h</td>
<td></td>
<td>VectorPath</td>
<td>-</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>PMAXSW mmreg1, mmreg2</td>
<td>0Fh</td>
<td>EEh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PMAXSW mmreg, mem64</td>
<td>0Fh</td>
<td>EEh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PMAXUB mmreg1, mmreg2</td>
<td>0Fh</td>
<td>DEh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PMAXUB mmreg, mem64</td>
<td>0Fh</td>
<td>DEh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PMINSW mmreg1, mmreg2</td>
<td>0Fh</td>
<td>EAh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PMINSW mmreg, mem64</td>
<td>0Fh</td>
<td>EAh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PMINUB mmreg1, mmreg2</td>
<td>0Fh</td>
<td>DAh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PMINUB mmreg, mem64</td>
<td>0Fh</td>
<td>DAh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PMOVMSKB reg32, mmreg</td>
<td>0Fh</td>
<td>D7h</td>
<td></td>
<td>VectorPath</td>
<td>-</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>PMULHUW mmreg1, mmreg2</td>
<td>0Fh</td>
<td>E4h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PMULHUW mmreg, mem64</td>
<td>0Fh</td>
<td>E4h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PSADBW mmreg1, mmreg2</td>
<td>0Fh</td>
<td>F6h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PSADBW mmreg, mem64</td>
<td>0Fh</td>
<td>F6h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PSHUFW mmreg1, mmreg2, imm8</td>
<td>0Fh</td>
<td>70h</td>
<td></td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSHUFW mmreg, mem64, imm8</td>
<td>0Fh</td>
<td>70h</td>
<td></td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PREFETCHNTA mem8</td>
<td>0Fh</td>
<td>18h</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>-</td>
<td>~</td>
<td>1</td>
</tr>
<tr>
<td>PREFETCHT0 mem8</td>
<td>0Fh</td>
<td>18h</td>
<td>mm-001-xxx</td>
<td>DirectPath</td>
<td>-</td>
<td>~</td>
<td>1</td>
</tr>
<tr>
<td>PREFETCHT1 mem8</td>
<td>0Fh</td>
<td>18h</td>
<td>mm-010-xxx</td>
<td>DirectPath</td>
<td>-</td>
<td>~</td>
<td>1</td>
</tr>
<tr>
<td>PREFETCHT2 mem8</td>
<td>0Fh</td>
<td>18h</td>
<td>mm-011-xxx</td>
<td>DirectPath</td>
<td>-</td>
<td>~</td>
<td>1</td>
</tr>
<tr>
<td>SFENCE</td>
<td>0Fh</td>
<td>AEh</td>
<td></td>
<td>VectorPath</td>
<td>-</td>
<td>2/8</td>
<td>2</td>
</tr>
</tbody>
</table>

**Notes:**
1. For the PREFETCHNTA/T0/T1/T2 instructions, the mem8 value refers to an address in the 64-byte line that will be prefetched.
2. 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.
3. The latency listed is the absolute minimum, while average latencies may be higher and are a function of internal pipeline conditions.
Table 22. Floating-Point Instructions

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>FPU Pipe(s)</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2XM1</td>
<td>D9h</td>
<td>11-110-000</td>
<td>VectorPath</td>
<td>-</td>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FABS</td>
<td>D9h</td>
<td>11-100-001</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FADD ST, ST(i)</td>
<td>D8h</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FADD [mem32real]</td>
<td>D8h</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FADD ST(i), ST</td>
<td>DCh</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FADD [mem64real]</td>
<td>DCh</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FADDP ST(i), ST</td>
<td>DEh</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FBLD [mem80]</td>
<td>DFh</td>
<td>mm-100-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FBSTP [mem80]</td>
<td>DFh</td>
<td>mm-110-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>198</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCHS</td>
<td>D9h</td>
<td>11-100-000</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCLEX</td>
<td>DBh</td>
<td>11-100-010</td>
<td>VectorPath</td>
<td>-</td>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCMOVBE ST(0), ST(i)</td>
<td>DAh</td>
<td>11-000-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>FCMOVE ST(0), ST(i)</td>
<td>DAh</td>
<td>11-001-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>FCMOVST ST(0), ST(i)</td>
<td>DAh</td>
<td>11-010-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>FCMOVU ST(0), ST(i)</td>
<td>DAh</td>
<td>11-011-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>FCMOVNB ST(0), ST(i)</td>
<td>DBh</td>
<td>11-000-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>FCMOVNBE ST(0), ST(i)</td>
<td>DBh</td>
<td>11-001-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>FCMOVNBU ST(0), ST(i)</td>
<td>DBh</td>
<td>11-011-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>FCOM ST(i)</td>
<td>D8h</td>
<td>11-100-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FCOMP ST(i)</td>
<td>D8h</td>
<td>11-011-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FCOM [mem32real]</td>
<td>D8h</td>
<td>mm-010-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. The last three bits of the modR/M byte select the stack entry ST(i).
2. These instructions have an effective latency of that which is listed. 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. A VectorPath decoded operation that uses one execution pipe (one ROP).
4. The cycle count listed is purely for execution and does not take into account the time required by the instruction to access the Load/Store Unit. It is recommended that operations dependent on the result of this particular operation be pushed back by at least two additional cycles.
5. Three latency numbers refer to precision control settings of single precision, double precision, and extended precision, respectively.
6. There is additional latency associated with this instruction. “e” is the difference between the exponents of divisor and dividend. If “s” is the number of normalization shifts performed on the result, then n = (s+1)/2 where (0 <= n <= 32).
7. The latency provided for this operation is the best-case latency. See “Minimize Floating-Point-to-Integer Conversions” on page 154 for more information.
8. The first number is for when no error condition is present. The second number is for when there is an error condition.
Table 22. Floating-Point Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>FPU Pipe(s)</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCOM [mem64real]</td>
<td>DCh</td>
<td>mm-010-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FCOMI ST, ST(i)</td>
<td>DBh</td>
<td>11-110-xxx</td>
<td>VectorPath</td>
<td>FADD</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>FCOMIP ST, ST(i)</td>
<td>DFe</td>
<td>11-110-xxx</td>
<td>VectorPath</td>
<td>FADD</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>FCOMP [mem32real]</td>
<td>D8h</td>
<td>mm-011-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FCOMP [mem64real]</td>
<td>DCh</td>
<td>mm-011-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FCOMPP</td>
<td>DEh</td>
<td>11-011-001</td>
<td>DirectPath</td>
<td>FADD</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FCOS</td>
<td>D9h</td>
<td>11-111-111</td>
<td>VectorPath</td>
<td>-</td>
<td>97-196</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FDECSTP</td>
<td>D9h</td>
<td>11-110-110</td>
<td>DirectPath</td>
<td>FADD/FMUL/FSTORE</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FDIV ST, ST(i)</td>
<td>D8h</td>
<td>11-110-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>16/20/24</td>
<td>1, 5</td>
<td></td>
</tr>
<tr>
<td>FDIV ST(i), ST</td>
<td>DCh</td>
<td>11-111-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>16/20/24</td>
<td>1, 5</td>
<td></td>
</tr>
<tr>
<td>FDIV [mem32real]</td>
<td>D8h</td>
<td>mm-100-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>16/20/24</td>
<td>4, 5</td>
<td></td>
</tr>
<tr>
<td>FDIV [mem64real]</td>
<td>DCh</td>
<td>mm-111-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>16/20/24</td>
<td>4, 5</td>
<td></td>
</tr>
<tr>
<td>FDIVP ST(i), ST</td>
<td>DEh</td>
<td>11-111-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>16/20/24</td>
<td>1, 5</td>
<td></td>
</tr>
<tr>
<td>FDIVR ST(i), ST</td>
<td>D8h</td>
<td>11-110-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>16/20/24</td>
<td>1, 5</td>
<td></td>
</tr>
<tr>
<td>FDIVR [mem32real]</td>
<td>D8h</td>
<td>mm-111-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>16/20/24</td>
<td>4, 5</td>
<td></td>
</tr>
<tr>
<td>FDIVR [mem64real]</td>
<td>DCh</td>
<td>mm-111-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>16/20/24</td>
<td>4, 5</td>
<td></td>
</tr>
<tr>
<td>FDIVRP ST(i), ST</td>
<td>DEh</td>
<td>11-110-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>16/20/24</td>
<td>1, 5</td>
<td></td>
</tr>
<tr>
<td>FFREE ST(i)</td>
<td>DDh</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL/FSTORE</td>
<td>2</td>
<td>1, 2</td>
<td></td>
</tr>
<tr>
<td>FFREEP ST(i)</td>
<td>DFe</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL/FSTORE</td>
<td>2</td>
<td>1, 2</td>
<td></td>
</tr>
<tr>
<td>FIADD [mem32int]</td>
<td>DAh</td>
<td>mm-000-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>9</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FIADD [mem16int]</td>
<td>DEh</td>
<td>mm-000-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>9</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FICOM [mem32int]</td>
<td>DAh</td>
<td>mm-010-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>9</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. The last three bits of the modR/M byte select the stack entry ST(i).
2. These instructions have an effective latency of that which is listed. 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. A VectorPath decoded operation that uses one execution pipe (one ROP).
4. The cycle count listed is purely for execution and does not take into account the time required by the instruction to access the Load/Store Unit. It is recommended that operations dependent on the result of this particular operation be pushed back by at least two additional cycles.
5. Three latency numbers refer to precision control settings of single precision, double precision, and extended precision, respectively.
6. There is additional latency associated with this instruction. “n” is the difference between the exponents of divisor and dividend. If “s” is the number of normalization shifts performed on the result, then n = (s+1)/2 where (0 <= n <= 32).
7. The latency provided for this operation is the best-case latency. See “Minimize Floating-Point-to-Integer Conversions” on page 154 for more information.
8. The first number is for when no error condition is present. The second number is for when there is an error condition.
Table 22. Floating-Point Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>FPU Pipe(s)</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>FICOM [mem16int]</td>
<td>DEh</td>
<td>mm-010-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>9</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FICOMP [mem32int]</td>
<td>DAh</td>
<td>mm-011-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>9</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FICOMP [mem16int]</td>
<td>DEh</td>
<td>mm-011-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>9</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FIDIV [mem32int]</td>
<td>DAh</td>
<td>mm-110-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>21/25/29</td>
<td>4, 5</td>
<td></td>
</tr>
<tr>
<td>FIDIV [mem16int]</td>
<td>DEh</td>
<td>mm-110-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>21/25/29</td>
<td>4, 5</td>
<td></td>
</tr>
<tr>
<td>FIDIVR [mem32int]</td>
<td>DAh</td>
<td>mm-111-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>21/25/29</td>
<td>4, 5</td>
<td></td>
</tr>
<tr>
<td>FIDIVR [mem16int]</td>
<td>DEh</td>
<td>mm-111-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>21/25/29</td>
<td>4, 5</td>
<td></td>
</tr>
<tr>
<td>FILD [mem16int]</td>
<td>DFh</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>FSTORE</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FILD [mem32int]</td>
<td>DBh</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>FSTORE</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FILD [mem64int]</td>
<td>DFh</td>
<td>mm-101-xxx</td>
<td>DirectPath</td>
<td>FSTORE</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FIMUL [mem32int]</td>
<td>DAh</td>
<td>mm-001-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>9</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FIMUL [mem16int]</td>
<td>DEh</td>
<td>mm-001-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>9</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FINCSTP</td>
<td>D9h</td>
<td>11-110-111</td>
<td>DirectPath</td>
<td>FADD/FMUL/FSTORE</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>FINIT</td>
<td>DBh</td>
<td>11-100-011</td>
<td>VectorPath</td>
<td>-</td>
<td>91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIST [mem16int]</td>
<td>DFh</td>
<td>mm-010-xxx</td>
<td>DirectPath</td>
<td>FSTORE</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FIST [mem32int]</td>
<td>DBh</td>
<td>mm-010-xxx</td>
<td>DirectPath</td>
<td>FSTORE</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FISTP [mem16int]</td>
<td>DFh</td>
<td>mm-011-xxx</td>
<td>DirectPath</td>
<td>FSTORE</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FISTP [mem32int]</td>
<td>DBh</td>
<td>mm-011-xxx</td>
<td>DirectPath</td>
<td>FSTORE</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FISTP [mem64int]</td>
<td>DFh</td>
<td>mm-111-xxx</td>
<td>DirectPath</td>
<td>FSTORE</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FISUB [mem32int]</td>
<td>DAh</td>
<td>mm-100-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>9</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FISUB [mem16int]</td>
<td>DEh</td>
<td>mm-100-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>9</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FISUBR [mem32int]</td>
<td>DAh</td>
<td>mm-101-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>9</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FISUBR [mem16int]</td>
<td>DEh</td>
<td>mm-101-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>9</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. The last three bits of the modR/M byte select the stack entry ST(i).
2. These instructions have an effective latency of that which is listed. 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. A VectorPath decoded operation that uses one execution pipe (one ROP).
4. The cycle count listed is purely for execution and does not take into account the time required by the instruction to access the Load/Store Unit. It is recommended that operations dependent on the result of this particular operation be pushed back by at least two additional cycles.
5. Three latency numbers refer to precision control settings of single precision, double precision, and extended precision, respectively.
6. There is additional latency associated with this instruction. “e” is the difference between the exponents of divisor and dividend. If “s” is the number of normalization shifts performed on the result, then n = (s+1)/2 where (0 <= n <= 32).
7. The latency provided for this operation is the best-case latency. See “Minimize Floating-Point-to-Integer Conversions” on page 154 for more information.
8. The first number is for when no error condition is present. The second number is for when there is an error condition.
### Table 22. Floating-Point Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>FPU Pipe(s)</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD ST(i)</td>
<td>D9h</td>
<td>11-000-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FLD [mem32real]</td>
<td>D9h</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL/FSTORE</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FLD [mem64real]</td>
<td>DDh</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL/FSTORE</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FLD [mem80real]</td>
<td>DBh</td>
<td>mm-101-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>10</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FLD1</td>
<td>D9h</td>
<td>11-101-000</td>
<td>DirectPath</td>
<td>FSTORE</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDCW [mem16]</td>
<td>D9h</td>
<td>mm-101-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDEVN [mem14byte]</td>
<td>D9h</td>
<td>mm-100-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDEVN [mem28byte]</td>
<td>D9h</td>
<td>mm-100-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>129</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDLE2</td>
<td>D9h</td>
<td>11-101-010</td>
<td>DirectPath</td>
<td>FSTORE</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDL2T</td>
<td>D9h</td>
<td>11-101-001</td>
<td>DirectPath</td>
<td>FSTORE</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDL2G</td>
<td>D9h</td>
<td>11-101-100</td>
<td>DirectPath</td>
<td>FSTORE</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDLN2</td>
<td>D9h</td>
<td>11-101-101</td>
<td>DirectPath</td>
<td>FSTORE</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDP1</td>
<td>D9h</td>
<td>11-101-011</td>
<td>DirectPath</td>
<td>FSTORE</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLDZ</td>
<td>D9h</td>
<td>11-101-110</td>
<td>DirectPath</td>
<td>FSTORE</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMUL ST, ST(i)</td>
<td>D8h</td>
<td>11-001-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FMUL ST(i), ST</td>
<td>DCh</td>
<td>11-001-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FMUL [mem32real]</td>
<td>D8h</td>
<td>mm-001-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FMUL [mem64real]</td>
<td>DCh</td>
<td>mm-001-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FMULP ST(i), ST</td>
<td>DEh</td>
<td>11-001-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FNOP</td>
<td>D9h</td>
<td>11-010-000</td>
<td>DirectPath</td>
<td>FADD/FMUL/FSTORE</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>FPTAN</td>
<td>D9h</td>
<td>11-110-010</td>
<td>VectorPath</td>
<td>-</td>
<td>107-216</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPTAN</td>
<td>D9h</td>
<td>11-110-011</td>
<td>VectorPath</td>
<td>-</td>
<td>158-175</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FPREM</td>
<td>D9h</td>
<td>11-111-000</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>9+e+n</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

1. The last three bits of the modR/M byte select the stack entry ST(i).
2. These instructions have an effective latency of that which is listed. 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. A VectorPath decoded operation that uses one execution pipe (one ROP).
4. The cycle count listed is purely for execution and does not take into account the time required by the instruction to access the Load/Store Unit. It is recommended that operations dependent on the result of this particular operation be pushed back by at least two additional cycles.
5. Three latency numbers refer to precision control settings of single precision, double precision, and extended precision, respectively.
6. There is additional latency associated with this instruction. “e” is the difference between the exponents of divisor and dividend. If “s” is the number of normalization shifts performed on the result, then $n = (s+1)/2$ where $(0 \leq n \leq 32)$.
7. The latency provided for this operation is the best-case latency. See “Minimize Floating-Point-to-Integer Conversions” on page 154 for more information.
8. The first number is for when no error condition is present. The second number is for when there is an error condition.
### Table 22. Floating-Point Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>FPU Pipe(s)</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPREM1</td>
<td>D9h</td>
<td>11-110-101</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>9+e+n 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FRNDINT</td>
<td>D9h</td>
<td>11-111-100</td>
<td>VectorPath</td>
<td></td>
<td>- 10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FRSTOR [mem94byte]</td>
<td>DDh</td>
<td>mm-100-xxx</td>
<td>VectorPath</td>
<td></td>
<td>- 138</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FRSTOR [mem108byte]</td>
<td>DDh</td>
<td>mm-100-xxx</td>
<td>VectorPath</td>
<td></td>
<td>- 138</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSAVE [mem94byte]</td>
<td>DDh</td>
<td>mm-110-xxx</td>
<td>VectorPath</td>
<td></td>
<td>- 159</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSAVE [mem108byte]</td>
<td>DDh</td>
<td>mm-110-xxx</td>
<td>VectorPath</td>
<td></td>
<td>- 159</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSQRT</td>
<td>D9h</td>
<td>11-111-101</td>
<td>VectorPath</td>
<td></td>
<td>- 8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSIN</td>
<td>D9h</td>
<td>11-111-110</td>
<td>VectorPath</td>
<td></td>
<td>96-192</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSINCOS</td>
<td>D9h</td>
<td>11-111-011</td>
<td>VectorPath</td>
<td></td>
<td>107-211</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSQRT</td>
<td>D9h</td>
<td>11-111-010</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>19/27/35 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FST [mem32real]</td>
<td>D9h</td>
<td>mm-010-xxx</td>
<td>DirectPath</td>
<td>FSTORE</td>
<td>2 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FST [mem64real]</td>
<td>DDh</td>
<td>mm-010-xxx</td>
<td>DirectPath</td>
<td>FSTORE</td>
<td>2 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FST ST(i)</td>
<td>DDh</td>
<td>11-010xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTCW [mem16]</td>
<td>D9h</td>
<td>mm-111-xxx</td>
<td>VectorPath</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTENV [mem14byte]</td>
<td>D9h</td>
<td>mm-110-xxx</td>
<td>VectorPath</td>
<td></td>
<td>89</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTENV [mem28byte]</td>
<td>D9h</td>
<td>mm-110-xxx</td>
<td>VectorPath</td>
<td></td>
<td>89</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTP [mem32real]</td>
<td>D9h</td>
<td>mm-011-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>4 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTP [mem64real]</td>
<td>DDh</td>
<td>mm-011-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>4 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTP [mem80real]</td>
<td>D9h</td>
<td>mm-111-xxx</td>
<td>VectorPath</td>
<td></td>
<td>8 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTP ST(i)</td>
<td>DDh</td>
<td>11-011-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTSW AX</td>
<td>DFh</td>
<td>11-100-000</td>
<td>VectorPath</td>
<td></td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSTSW [mem16]</td>
<td>DDh</td>
<td>mm-111-xxx</td>
<td>VectorPath</td>
<td>FSTORE</td>
<td>8 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FSUB [mem32real]</td>
<td>D8h</td>
<td>mm-100-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

1. The last three bits of the modR/M byte select the stack entry ST(i).
2. These instructions have an effective latency of that which is listed. 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. A VectorPath decoded operation that uses one execution pipe (one ROP).
4. The cycle count listed is purely for execution and does not take into account the time required by the instruction to access the Load/Store Unit. It is recommended that operations dependent on the result of this particular operation be pushed back by at least two additional cycles.
5. Three latency numbers refer to precision control settings of single precision, double precision, and extended precision, respectively.
6. There is additional latency associated with this instruction. “e” is the difference between the exponents of divisor and dividend. If “s” is the number of normalization shifts performed on the result, then n = (5s+1)/2 where (0 ≤ n ≤ 32).
7. The latency provided for this operation is the best-case latency. See “Minimize Floating-Point-to-Integer Conversions” on page 154 for more information.
8. The first number is for when no error condition is present. The second number is for when there is an error condition.
### Table 22. Floating-Point Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>First Byte</th>
<th>Second Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>FPU Pipe(s)</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSUB [mem64real]</td>
<td>DCh</td>
<td>mm-100-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FSUB ST, ST(i)</td>
<td>D8h</td>
<td>11-100-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FSUB ST(i), ST</td>
<td>DCh</td>
<td>11-101-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FSUBP ST(i), ST</td>
<td>DEh</td>
<td>11-101-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FSUBR [mem32real]</td>
<td>D8h</td>
<td>mm-101-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FSUBR [mem64real]</td>
<td>DCh</td>
<td>mm-101-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>FSUBR ST, ST(i)</td>
<td>D8h</td>
<td>11-100-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FSUBRP ST(i), ST</td>
<td>DEh</td>
<td>11-100-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FTST</td>
<td>D9h</td>
<td>11-100-100</td>
<td>DirectPath</td>
<td>FADD</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FUCOM</td>
<td>DDh</td>
<td>11-100-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FUCOMI ST, ST(i)</td>
<td>DBh</td>
<td>11-101-xxx</td>
<td>VectorPath</td>
<td>FADD</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>FUCOMIP ST, ST(i)</td>
<td>DFh</td>
<td>11-101-xxx</td>
<td>VectorPath</td>
<td>FADD</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>FUCOMP</td>
<td>DDh</td>
<td>11-101-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FUCOMPP</td>
<td>DAh</td>
<td>11-101-001</td>
<td>DirectPath</td>
<td>FADD</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FWAIT</td>
<td>9Bh</td>
<td></td>
<td>DirectPath</td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>FXAM</td>
<td>D9h</td>
<td>11-100-101</td>
<td>VectorPath</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>FXCHR</td>
<td>D9h</td>
<td>11-001-xxx</td>
<td>VectorPath</td>
<td>FADD/FMUL/FSTORE</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>FXRSTOR [mem512byte]</td>
<td>0Fh</td>
<td>AEh</td>
<td>mm-001-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>68/108</td>
<td>8</td>
</tr>
<tr>
<td>FSXSAVE [mem512byte]</td>
<td>0Fh</td>
<td>AEh</td>
<td>mm-000-xxx</td>
<td>VectorPath</td>
<td>-</td>
<td>31/79</td>
<td>8</td>
</tr>
<tr>
<td>FXTRACT</td>
<td>D9h</td>
<td>11-110-100</td>
<td>VectorPath</td>
<td></td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>FYL2X</td>
<td>D9h</td>
<td>11-110-001</td>
<td>VectorPath</td>
<td></td>
<td>-</td>
<td>116-126</td>
<td></td>
</tr>
<tr>
<td>FYL2XPI</td>
<td>D9h</td>
<td>11-111-001</td>
<td>VectorPath</td>
<td></td>
<td>-</td>
<td>126</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
1. The last three bits of the modR/M byte select the stack entry ST(i).
2. These instructions have an effective latency of that which is listed. 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. A VectorPath decoded operation that uses one execution pipe (one ROP).
4. The cycle count listed is purely for execution and does not take into account the time required by the instruction to access the Load/Store Unit. It is recommended that operations dependent on the result of this particular operation be pushed back by at least two additional cycles.
5. Three latency numbers refer to precision control settings of single precision, double precision, and extended precision, respectively.
6. There is additional latency associated with this instruction. “e” is the difference between the exponents of divisor and dividend. If “s” is the number of normalization shifts performed on the result, then n = (s + 1)/2 where (0 <= n <= 32).
7. The latency provided for this operation is the best-case latency. See “Minimize Floating-Point-to-Integer Conversions” on page 154 for more information.
8. The first number is for when no error condition is present. The second number is for when there is an error condition.
### Table 23. 3DNow!™ Instructions

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Prefix Byte(s)</th>
<th>imm8</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>FPU Pipe(s)</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEMMS</td>
<td>0Fh</td>
<td>0Eh</td>
<td></td>
<td>DirectPath</td>
<td>FADD/FMUL/FSTORE</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PAVGUSB mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>BFh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PAVGUSB mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>BFh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>PF2ID mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>1Dh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PF2ID mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>1Dh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>PFACC mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>AEh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PFACC mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>AEh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>PFADD mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>9Eh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PFADD mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>9Eh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>PFCMPEQ mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>B0h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PFCMPEQ mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>B0h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>PFCMPGE mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>90h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PFCMPGE mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>90h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>PFCMPGT mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>A0h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PFCMPGT mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>A0h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>PFMAX mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>A4h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PFMAX mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>A4h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>PFMUL mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>94h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PFMUL mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>94h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>PFRCP mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>96h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PFRCP mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>96h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>PFRCPIT1 mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>A6h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PFRCPIT1 mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>A6h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>PFRCPIT2 mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>B6h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PFRCPIT2 mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>B6h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

**Notes:**

1. For the PREFETCH and PREFETCHW instructions, the mem8 value refers to an address in the 64-byte line that will be prefetched.
2. The byte listed in the column titled ‘imm8’ is actually the Opcode Byte.
3. The cycle count listed is purely for execution and does not take into account the time required by the instruction to access the Load/Store Unit. It is recommended that operations dependent on the result of this particular operation be pushed back by at least two additional cycles.
4. This instruction has an effective latency of that which is listed. However, it generates an internal NOP with a latency of two cycles but no related dependencies. These internal NOP(s) can be executed at a rate of three per cycle and can use any of the three execution resources.
### Table 23. 3DNow!™ Instructions (Continued)

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Prefix Byte(s)</th>
<th>imm8</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>FPU Pipe(s)</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>PFRSQIT1 mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>A7h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PFRSQIT1 mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>A7h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>4, 3</td>
<td></td>
</tr>
<tr>
<td>PFRSQR T mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>97h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PFRSQR T mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>97h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>3, 3</td>
<td></td>
</tr>
<tr>
<td>PFSUB mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>9Ah</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PFSUB mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>9Ah</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4, 3</td>
<td></td>
</tr>
<tr>
<td>PFSUBR mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>AAh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PFSUBR mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>AAh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4, 3</td>
<td></td>
</tr>
<tr>
<td>PI2FD mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>0Dh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PI2FD mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>0Dh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4, 3</td>
<td></td>
</tr>
<tr>
<td>PMULHRW mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>B7h</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>PMULHRW mmreg1, mem64</td>
<td>0Fh, 0Fh</td>
<td>B7h</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>3, 3</td>
<td></td>
</tr>
<tr>
<td>PREFETCH mem8</td>
<td>0Fh</td>
<td>0Dh</td>
<td>mm-000-xxx</td>
<td>DirectPath</td>
<td>-</td>
<td>~</td>
<td>1, 2</td>
</tr>
<tr>
<td>PREFETCHW mem8</td>
<td>0Fh</td>
<td>0Dh</td>
<td>mm-001-xxx</td>
<td>DirectPath</td>
<td>-</td>
<td>~</td>
<td>1, 2</td>
</tr>
</tbody>
</table>

**Notes:**

1. For the PREFETCH and PREFETCHW instructions, the mem8 value refers to an address in the 64-byte line that will be prefetched.
2. The byte listed in the column titled ‘imm8’ is actually the Opcode Byte.
3. The cycle count listed is purely for execution and does not take into account the time required by the instruction to access the Load/Store Unit. It is recommended that operations dependent on the result of this particular operation be pushed back by at least two additional cycles.
4. This instruction has an effective latency of that which is listed. However, it generates an internal NOP with a latency of two cycles but no related dependencies. These internal NOP(s) can be executed at a rate of three per cycle and can use any of the three execution resources.
### Table 24. 3DNow!™ Extensions

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Prefix Byte(s)</th>
<th>imm8</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>FPU Pipe(s)</th>
<th>Execute Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>PF2IW mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>1Ch</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PF2IW mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>1Ch</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4 1</td>
<td></td>
</tr>
<tr>
<td>PFNACC mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>8Ah</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PFNACC mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>8Ah</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4 1</td>
<td></td>
</tr>
<tr>
<td>PFPNACC mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>8Eh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PFPNACC mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>8Eh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4 1</td>
<td></td>
</tr>
<tr>
<td>PI2FW mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>0Ch</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>PI2FW mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>0Ch</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4 1</td>
<td></td>
</tr>
<tr>
<td>PSWAPD mmreg1, mmreg2</td>
<td>0Fh, 0Fh</td>
<td>BBh</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PSWAPD mmreg, mem64</td>
<td>0Fh, 0Fh</td>
<td>BBh</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD/FMUL</td>
<td>2 1</td>
<td></td>
</tr>
</tbody>
</table>

**Note:**

1. The cycle count listed is purely for execution and does not take into account the time required by the instruction to access the Load/Store Unit. It is recommended that operations dependent on the result of this particular operation be pushed back by at least two additional cycles.
## Table 25. Instructions Introduced with 3DNow!™ Professional

<table>
<thead>
<tr>
<th>Instruction Mnemonic</th>
<th>Prefix Byte</th>
<th>First Byte</th>
<th>2nd Byte</th>
<th>ModR/M Byte</th>
<th>Decode Type</th>
<th>FPU Pipe</th>
<th>Latency</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDPS xmmreg1, xmmreg2</td>
<td>0F</td>
<td>58</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>FADD</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ADDDS xmmreg, mem128</td>
<td>0F</td>
<td>58</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FADD</td>
<td>5</td>
<td>1,4</td>
<td></td>
</tr>
<tr>
<td>ADDDS xmmreg1, xmmreg2</td>
<td>F3</td>
<td>0F</td>
<td>58</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>ADDSS xmmreg, mem128</td>
<td>F3</td>
<td>0F</td>
<td>58</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>ANDNPS xmmreg1, xmmreg2</td>
<td>0F</td>
<td>55</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>3</td>
<td>1,4</td>
<td></td>
</tr>
<tr>
<td>ANDNPS xmmreg, mem128</td>
<td>0F</td>
<td>55</td>
<td></td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>3</td>
<td>1,4</td>
</tr>
<tr>
<td>ANDPS xmmreg1, xmmreg2</td>
<td>0F</td>
<td>54</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>3</td>
<td>1,4</td>
<td></td>
</tr>
<tr>
<td>ANDPS xmmreg, mem128</td>
<td>0F</td>
<td>54</td>
<td></td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>3</td>
<td>1,4</td>
</tr>
<tr>
<td>CMPPS xmmreg1, xmmreg2, imm8</td>
<td>0F</td>
<td>C2</td>
<td></td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>CMPPS xmmreg, mem128, imm8</td>
<td>0F</td>
<td>C2</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>3</td>
<td>1,4</td>
<td></td>
</tr>
<tr>
<td>CMPSS xmmreg1, xmmreg2, imm8</td>
<td>F3</td>
<td>0F</td>
<td>C2</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>CMPSS xmmreg, mem32, imm8</td>
<td>F3</td>
<td>0F</td>
<td>C2</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>COMISS xmmreg1, xmmreg2</td>
<td>0F</td>
<td>2F</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>COMISS xmmreg, mem32</td>
<td>0F</td>
<td>2F</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td></td>
<td></td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>CVTPi2PS xmmreg, mmreg</td>
<td>0F</td>
<td>2A</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CVTPi2PS xmmreg, mem64</td>
<td>0F</td>
<td>2A</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CVTSS2SI reg32, xmmreg</td>
<td>F3</td>
<td>0F</td>
<td>2A</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>~</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CVTSS2SI reg32, xmmreg</td>
<td>F3</td>
<td>0F</td>
<td>2A</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>~</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CVTSS2SI reg32, mem32</td>
<td>F3</td>
<td>0F</td>
<td>2D</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>7/11</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>CVTSS2SI reg32, mem32</td>
<td>F3</td>
<td>0F</td>
<td>2D</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>7/11</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CVTTPS2I xmmreg, xmmreg</td>
<td>0F</td>
<td>2C</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>CVTTPS2I xmmreg, mem128</td>
<td>0F</td>
<td>2C</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>DIVPS xmmreg1, xmmreg2</td>
<td>0F</td>
<td>5E</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>29/16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIVPS xmmreg, mem128</td>
<td>0F</td>
<td>5E</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>29/16</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>DIVPS xmmreg, mem128</td>
<td>0F</td>
<td>5E</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>29/16</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>DIVS xmmreg1, xmmreg2</td>
<td>F3</td>
<td>0F</td>
<td>5E</td>
<td>11-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>DIVSS xmmreg, mem32</td>
<td>F3</td>
<td>0F</td>
<td>5E</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>LDMXCSR mem32</td>
<td>0F</td>
<td>AE</td>
<td>mm-010-xxx</td>
<td>VectorPath</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAXPS xmmreg1, xmmreg2</td>
<td>0F</td>
<td>5F</td>
<td>11-xxx-xxx</td>
<td>VectorPath</td>
<td>FADD</td>
<td>37</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MAXPS xmmreg, mem128</td>
<td>0F</td>
<td>5F</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FADD</td>
<td>3</td>
<td>1,4</td>
<td></td>
</tr>
</tbody>
</table>
Table 25. Instructions Introduced with 3DNow!™ Professional (Continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>F3/F0</th>
<th>5F</th>
<th>Type</th>
<th>Path</th>
<th>FMUL</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXSS xmmreg1, xmmreg2</td>
<td>F3</td>
<td>0F</td>
<td>5F</td>
<td>DirectPath</td>
<td>FADD</td>
</tr>
<tr>
<td>MAXSS xmmreg, mem32</td>
<td>F3</td>
<td>0F</td>
<td>5F</td>
<td>mm-mm-mm-mm</td>
<td>DirectPath</td>
</tr>
<tr>
<td>MINPS xmmreg1, xmmreg2</td>
<td>0F</td>
<td>5D</td>
<td>11-mm-mm-mm</td>
<td>VectorPath</td>
<td>FADD</td>
</tr>
<tr>
<td>MINPS xmmreg, mem128</td>
<td>0F</td>
<td>5D</td>
<td>mm-mm-mm-mm</td>
<td>VectorPath</td>
<td>FADD</td>
</tr>
<tr>
<td>MINSS xmmreg1, xmmreg2</td>
<td>F3</td>
<td>0F</td>
<td>5D</td>
<td>DirectPath</td>
<td>FADD</td>
</tr>
<tr>
<td>MINSS xmmreg, mem32</td>
<td>F3</td>
<td>0F</td>
<td>5D</td>
<td>mm-mm-mm-mm</td>
<td>DirectPath</td>
</tr>
<tr>
<td>MOVAPS xmmreg1, xmmreg2</td>
<td>0F</td>
<td>28</td>
<td>11-mm-mm-mm</td>
<td>VectorPath</td>
<td>2</td>
</tr>
<tr>
<td>MOVAPS xmmreg, mem128</td>
<td>0F</td>
<td>28</td>
<td>mm-mm-mm-mm</td>
<td>VectorPath</td>
<td>2</td>
</tr>
<tr>
<td>MOVAPS xmmreg1, xmmreg2</td>
<td>0F</td>
<td>29</td>
<td>11-mm-mm-mm</td>
<td>VectorPath</td>
<td>2</td>
</tr>
<tr>
<td>MOVAPS mem128, xmmreg</td>
<td>0F</td>
<td>29</td>
<td>mm-mm-mm-mm</td>
<td>VectorPath</td>
<td>3</td>
</tr>
<tr>
<td>MOVHLPS xmmreg1, xmmreg2</td>
<td>0F</td>
<td>16</td>
<td>11-mm-mm-mm</td>
<td>VectorPath</td>
<td>2</td>
</tr>
<tr>
<td>MOVHPS xmmreg, mem64</td>
<td>0F</td>
<td>16</td>
<td>mm-mm-mm-mm</td>
<td>VectorPath</td>
<td>2</td>
</tr>
<tr>
<td>MOVHLPS xmmreg1, xmmreg2</td>
<td>0F</td>
<td>17</td>
<td>mm-mm-mm-mm</td>
<td>VectorPath</td>
<td>2</td>
</tr>
<tr>
<td>MOVHPS mem64, xmmreg</td>
<td>0F</td>
<td>16</td>
<td>11-mm-mm-mm</td>
<td>VectorPath</td>
<td>2</td>
</tr>
<tr>
<td>MOVAPS xmmreg1, xmmreg2</td>
<td>0F</td>
<td>12</td>
<td>mm-mm-mm-mm</td>
<td>VectorPath</td>
<td>2</td>
</tr>
<tr>
<td>MOVAPS mem128, xmmreg</td>
<td>0F</td>
<td>13</td>
<td>mm-mm-mm-mm</td>
<td>VectorPath</td>
<td>2</td>
</tr>
<tr>
<td>MOVMSKPS reg32, xmmreg</td>
<td>0F</td>
<td>50</td>
<td>11-mm-mm-mm</td>
<td>VectorPath</td>
<td>7/11</td>
</tr>
<tr>
<td>MOVNTPS mem128, xmmreg</td>
<td>0F</td>
<td>28</td>
<td>mm-mm-mm-mm</td>
<td>VectorPath</td>
<td>5</td>
</tr>
<tr>
<td>MOVSS xmmreg1, xmmreg2</td>
<td>F3</td>
<td>0F</td>
<td>10</td>
<td>11-mm-mm-mm</td>
<td>VectorPath</td>
</tr>
<tr>
<td>MOVSS xmmreg, mem32</td>
<td>F3</td>
<td>0F</td>
<td>10</td>
<td>mm-mm-mm-mm</td>
<td>VectorPath</td>
</tr>
<tr>
<td>MOVSS xmmreg1, xmmreg2</td>
<td>F3</td>
<td>0F</td>
<td>11</td>
<td>11-mm-mm-mm</td>
<td>VectorPath</td>
</tr>
<tr>
<td>MOVSS mmreg1, xmmreg</td>
<td>F3</td>
<td>0F</td>
<td>11</td>
<td>mm-mm-mm-mm</td>
<td>VectorPath</td>
</tr>
<tr>
<td>MOVUPS xmmreg1, xmmreg2</td>
<td>0F</td>
<td>10</td>
<td>11-mm-mm-mm</td>
<td>VectorPath</td>
<td>2</td>
</tr>
<tr>
<td>MOVUPS xmmreg, mem128</td>
<td>0F</td>
<td>10</td>
<td>mm-mm-mm-mm</td>
<td>VectorPath</td>
<td>~5</td>
</tr>
<tr>
<td>MOVUPS xmmreg1, xmmreg2</td>
<td>0F</td>
<td>11</td>
<td>11-mm-mm-mm</td>
<td>VectorPath</td>
<td>2</td>
</tr>
<tr>
<td>MOVUPS mem128, xmmreg</td>
<td>0F</td>
<td>11</td>
<td>mm-mm-mm-mm</td>
<td>VectorPath</td>
<td>~5</td>
</tr>
<tr>
<td>MULPS xmmreg1, xmmreg2</td>
<td>0F</td>
<td>59</td>
<td>11-mm-mm-mm</td>
<td>VectorPath</td>
<td>FMUL</td>
</tr>
<tr>
<td>MULPS xmmreg, mem128</td>
<td>0F</td>
<td>59</td>
<td>mm-mm-mm-mm</td>
<td>VectorPath</td>
<td>FMUL</td>
</tr>
<tr>
<td>MULSS xmmreg1, xmmreg2</td>
<td>F3</td>
<td>0F</td>
<td>59</td>
<td>11-mm-mm-mm</td>
<td>VectorPath</td>
</tr>
<tr>
<td>MULSS xmmreg, mem32</td>
<td>F3</td>
<td>0F</td>
<td>59</td>
<td>mm-mm-mm-mm</td>
<td>VectorPath</td>
</tr>
<tr>
<td>ORPS xmmreg1, xmmreg2</td>
<td>0F</td>
<td>56</td>
<td>11-mm-mm-mm</td>
<td>VectorPath</td>
<td>FMUL</td>
</tr>
<tr>
<td>ORPS xmmreg, mem128</td>
<td>0F</td>
<td>56</td>
<td>mm-mm-mm-mm</td>
<td>VectorPath</td>
<td>FMUL</td>
</tr>
<tr>
<td>RCPSS xmmreg1, xmmreg2</td>
<td>0F</td>
<td>53</td>
<td>11-mm-mm-mm</td>
<td>VectorPath</td>
<td>FMUL</td>
</tr>
<tr>
<td>RCPSS xmmreg, mem128</td>
<td>0F</td>
<td>53</td>
<td>mm-mm-mm-mm</td>
<td>VectorPath</td>
<td>FMUL</td>
</tr>
<tr>
<td>RCPSS xmmreg1, xmmreg2</td>
<td>F3</td>
<td>0F</td>
<td>53</td>
<td>11-mm-mm-mm</td>
<td>DirectPath</td>
</tr>
</tbody>
</table>
### Table 25. Instructions Introduced with 3DNow!™ Professional (Continued)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Prefix</th>
<th>Type</th>
<th>Source</th>
<th>Destination</th>
<th>Path</th>
<th>Result Latency</th>
<th>First Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCPSS xmmreg, mem32</td>
<td>F3 0F 53</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>RSRTPS xmmreg1, xmmreg2</td>
<td>0F 52</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>RSRTPS xmmreg, mem128</td>
<td>0F 52</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>4</td>
<td>1,4</td>
<td></td>
</tr>
<tr>
<td>RSRTPS xmmreg1, xmmreg2</td>
<td>F3 0F 52</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>RSRTPS xmmreg, mem32</td>
<td>F3 0F 52</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SHUFPS xmmreg1, xmmreg2, imm8</td>
<td>0F C6</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>~4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SHUFPS xmmreg, mem128, imm8</td>
<td>0F C6</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>3/1</td>
<td>2,4</td>
<td></td>
</tr>
<tr>
<td>SQRTPS xmmreg1, xmmreg2</td>
<td>0F 51</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>35/19</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SQRTPS xmmreg, mem128</td>
<td>0F 51</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>35/19</td>
<td>2,4</td>
<td></td>
</tr>
<tr>
<td>SQRTSS xmmreg1, xmmreg2</td>
<td>F3 0F 51</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRTSS xmmreg, mem32</td>
<td>F3 0F 51</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FMUL</td>
<td>19</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>STMXCSR mem32</td>
<td>0F AE</td>
<td>mm-011-xxx</td>
<td>VectorPath</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUBPS xmmreg1, xmmreg2</td>
<td>0F 5C</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FADD</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SUBPS xmmreg, mem128</td>
<td>0F 5C</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FADD</td>
<td>5</td>
<td>1,4</td>
<td></td>
</tr>
<tr>
<td>SUBSS xmmreg1, xmmreg2</td>
<td>F3 0F 5C</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SUBSS xmmreg, mem32</td>
<td>F3 0F 5C</td>
<td>mm-xxx-xxx</td>
<td>DirectPath</td>
<td>FADD</td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>UCOMISS xmmreg1, xmmreg2</td>
<td>0F 2E</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UCOMISS xmmreg, mem32</td>
<td>0F 2E</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNPCKHPS xmmreg1, xmmreg2</td>
<td>0F 15</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>UNPCKHPS xmmreg, mem128</td>
<td>0F 15</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>3/4</td>
<td>1,4</td>
<td></td>
</tr>
<tr>
<td>UNPCKLPS xmmreg1, xmmreg2</td>
<td>0F 14</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>UNPCKLPS xmmreg, mem128</td>
<td>0F 14</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>3/4</td>
<td>3,4</td>
<td></td>
</tr>
<tr>
<td>XORPS xmmreg1, xmmreg2</td>
<td>0F 57</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>XORPS xmmreg, mem128</td>
<td>0F 57</td>
<td>mm-xxx-xxx</td>
<td>VectorPath</td>
<td>FMUL</td>
<td>3</td>
<td>1,4</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**

1. The low half of the result is available 1 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 1 cycle earlier than listed.
4. The cycle count listed is purely for execution and does not take into account the time required by the instruction to access the Load/Store Unit. It is recommended that operations dependent on the result of this particular operation be pushed back by at least two additional cycles.
Index

Numerics
3D Clipping ............................................ 190
3DNow™ and MMX™ Intra-Operand Swapping .... 169
3DNow™ Instructions .............................. 13, 161

A
Address Generation Interlocks .................. 110
AMD Athlon™ Processor
Branch-Free Code ................................. 94
Code Padding ......................................... 59
Compute Upper Half of Unsigned Products ... 168
Family ...................................................... 3
Microarchitecture ................................. 4, 203–204
AMD Athlon™ System Bus .................... 214
Command Generation Rules .................. 233

B
Binary-to-ASCII Decimal Conversion .............. 139
Blended Code, AMD-K6 and AMD Athlon Processors
3DNow! and MMX Intra-Operand Swapping ... 169
Block Copies and Block Fills .................. 174
Branch Examples ..................................... 94
Code Padding ......................................... 59
Compute Upper Half of Unsigned Products ... 168
Branch Target Buffer (BTB) ................. 104, 206
Branches
Align Branch Targets ............................... 54
Based on Comparisons Between Floats ........ 42
Compound Branch Conditions .................. 27
Dependent on Random Data ................. 14, 93
Prediction ............................................... 206
Replace with Computation in 3DNow! Code ... 98

C
C Language ............................................... 17, 27
Array-Style Over Pointer-Style Code .......... 20
C Code to 3DNow! Code Examples ........... 100–103
Structure Component Considerations .......... 36, 91
Cache ......................................................... 4
64-Byte Cache Line ................................. 15, 85
Cache and Memory Optimizations .............. 63
CALL and RETURN Instructions .............. 96
Code Padding Using Neutral Code Fillers ..... 58
Code Sample Analysis ............................. 226
Complex Number Arithmetic ................... 199
Const Type Qualifier ............................... 29
Constant Control Code, Multiple ............ 30

D
Data Cache ............................................... 208
Decoding ................................................. 49, 207
Dependencies .......................................... 202
DirectPath
Decoder .................................................. 207
DirectPath Over VectorPath Instructions ..... 10, 50
Displacements, 8-Bit Sign-Extended .......... 58
Division .................................................. 115–118, 144
Fast ......................................................... 162
Replace Divides with Multiplies, Integer ..... 44, 115
Using 3DNow! Instructions .................. 162–163
Dynamic Memory Allocation Consideration .... 33

E
Execution Unit Resources ....................... 222
Extended-Precision Data .......................... 154

F
Far Control Transfer Instructions ............... 104
FEMMS Instruction ................................. 162
Fetch and Decode Pipeline Stages .......... 215
FFREEP Macro ......................................... 152
Floating-Point
Compare Instructions ............................. 153
Divides and Square Roots ..................... 38
Execution Unit ......................................... 211
Optimizations ......................................... 151
Pipeline Operations ............................... 224
Pipeline Stages ......................................... 220
Scheduler ............................................... 210
Signed Words to Floating-Point ............... 170
To Integer Conversions ......................... 40, 154
Variables and Expressions are Type Float .... 17
Floor() Function ........................................ 197
FRNDINT Instruction ............................... 154
FSINCOS Instruction ............................... 159
FXCH Instruction ....................................... 153

G
Group I—Essential Optimizations ............... 7–8
Group II—Secondary Optimizations .......... 7, 10

I
If Statement ............................................. 32
Immediates, 8-Bit Sign-Extended ............. 57
IMUL Instruction ........................................... 120
Inline Functions ........................................ 109–110, 125
Inline REP String with Low Counts .................... 124

Instruction
Cache .......................................................... 205
Control Unit ............................................... 208
Decoding ...................................................... 49
Short Encodings ......................................... 54
Short Forms ............................................... 201
Integer ....................................................... 115
Arithmetic, 64-Bit ........................................ 125
Division ...................................................... 44
Execution Unit ............................................ 209
Operand, Consider Sign ................................ 18
Pipeline Operations ................. 223
Pipeline Stages ........................................... 218
Scheduler .................................................. 209
Use 32-Bit Data Types for Integer Code .............. 17

L
L2 Cache Controller ...................................... 213
LEA Instruction ........................................... 54, 57
LEAVE Instruction ....................................... 56
Load/Store .................................................. 24, 212, 225
Load-Execute Instructions ......................... 10–11, 50
Floating-Point Instructions ......................... 11, 51
Integer Instructions ................................... 50
Local Functions ......................................... 32
Local Variables ......................................... 37, 44, 92
LOOP Instruction ....................................... 104
Loops
Deriving Loop Control For Partially Unrolled .... 108
Generic Loop Hoisting ................................ 30
Minimize Pointer Arithmetic ....................... 112
Partial Loop Unrolling ................................ 106
REP String with Low Variable Counts .......... 124
Unroll Small Loops ................................... 22
Unrolling Loops ....................................... 106

M
Memory
Dynamic Memory Allocation ......................... 33
Pushing Memory Data ................................... 114
Size and Alignment Issues ....................... 8, 63
Memory Type Range Register (MTRR) ............ 243
Capability Register Format ......................... 246
Default Type Register Format ..................... 247
Fixed-Range Register Format ...................... 254
MSR Format ............................................. 257
MTRRs and PAT ....................................... 250
Overlapping ............................................. 248
Variable-Range MTRR Register Format ....... 255
Memory Types ........................................ 246
MMX™ Instructions .................................... 161

N
Newton-Raphson Reciprocal ......................... 164
Newton-Raphson Reciprocal Square Root ......... 166

O
Operands .................................................. 222
Largest Possible Operand Size, Repeated String 124
Optimization Star ....................................... 8

P
Page Attribute Table (PAT) ......................... 243, 249–250
Parallelism ................................................ 33
PAVUSB for MPEG-2 Motion Compensation .... 195
PerfCtr MSR ........................................... 240
PerfEvtSel MSR ....................................... 236
Performance-Monitoring Counters ............... 235, 241
PF2ID Instruction ..................................... 40
PFCMP Instruction .................................. 173
PFMUL Instruction .................................. 172
PI2FW Instruction .................................. 170
Pipeline and Execution Unit Resources ....... 215
Pointers
 Dereferenced Arguments ........................... 44
Use Array-Style Code Instead ................. 20
Population Count Function ...................... 136, 184
Predecode .............................................. 206
Prefetch
Determining Distance .............................. 83
Multiple ................................................ 81
PREFETCH and PREFETCHW Instructions ... 9, 79–80,
Index

82, 85
Prototypes ........................................... 29
PSWAPD ............................................. 169
PSWAPD Instruction ................................. 169, 199

R
Read-Modify-Write Instructions ...................... 52
Recursive Functions .................................. 96
Register Operands ................................... 202
Register Reads and Writes, Partial ................. 55
REP Prefix ........................................... 124
RGBA Pixels .......................................... 192

S
Scalar Code Translated into 3DNow! Code ........ 100
Scheduling ............................................ 105
SHLD Instruction .................................. 57
SHR Instruction .................................... 57
Signed Words to Floating-Point Conversion ........ 170
Square Root
  Fast (15-Bit Precision) ............................ 165
  Fast (24-Bit Precision) ......................... 165
  Reciprocal Approximation (Newton-Raphson) . 166
Stack
  Alignment Considerations ....................... 90
  Allocation ........................................ 202
Store-to-Load Forwarding ......................... 23–24, 87–88, 90
Stream of Packed Unsigned Bytes ................. 198
String Instructions ............................... 123–124
Structure (Struct) .................................. 36–37, 91–92
Subexpressions, Explicitly Extract Common ...... 35
Superscalar Processor ............................. 204
Switch Statement ................................... 28, 32

T
TBYTE Variables .................................... 91
Trigonometric Instructions ......................... 157

U
Unit-Stride Access .................................. 79, 84

V
VectorPath Decoder .................................. 207

W
Write Combining .................................... 12, 85, 213, 229–231, 233

X
x86 Optimization Guidelines ....................... 201
XOR Instruction ..................................... 125