AMD64 Technology

AMD64 Architecture
Programmer’s Manual
Volume 2:
System Programming

Publication No. Revision Date
24593 3.34 April 2020
The information contained herein is for informational purposes only, and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD’s products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. Any unauthorized copying, alteration, distribution,

**Trademarks**

AMD, the AMD arrow logo, and combinations thereof, AMD Virtualization and 3DNow! are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

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

HyperTransport is a licensed trademark of the HyperTransport Technology Consortium.
# Contents

Contents ............................................................................................................. iii  
Figures.............................................................................................................. xix  
Tables ............................................................................................................... xxvii  
Revision History ............................................................................................... xxxi  
Preface .............................................................................................................. xli  
About This Book ............................................................................................... xlii  
Audience .......................................................................................................... xlii  
Organization .................................................................................................... xlii  
Conventions and Definitions ............................................................................ xliii  
Notational Conventions ................................................................................... xliii  
Definitions ....................................................................................................... xliiv  
Registers ......................................................................................................... 1  
Endian Order ..................................................................................................... liii  
Related Documents .......................................................................................... liii  

1 System-Programming Overview .................................................................... 1  
  1.1 Memory Model ......................................................................................... 1  
  Memory Addressing ....................................................................................... 2  
  Memory Organization .................................................................................... 3  
  Canonical Address Form .............................................................................. 4  
  1.2 Memory Management ............................................................................. 5  
  Segmentation ................................................................................................ 5  
  Paging ........................................................................................................... 7  
  Mixing Segmentation and Paging ................................................................. 8  
  Real Addressing ............................................................................................ 10  
  1.3 Operating Modes .................................................................................... 11  
  Long Mode ................................................................................................... 12  
  64-Bit Mode ................................................................................................ 13  
  Compatibility Mode ....................................................................................... 13  
  Legacy Modes ............................................................................................... 14  
  System Management Mode (SMM) .............................................................. 15  
  1.4 System Registers ................................................................................... 15  
  1.5 System-Data Structures ........................................................................ 17  
  1.6 Interrupts ................................................................................................ 19  
  1.7 Additional System-Programming Facilities ........................................... 20  
  Hardware Multitasking ............................................................................... 20  
  Machine Check ............................................................................................. 21  
  Software Debugging ..................................................................................... 21  
  Performance Monitoring .............................................................................. 22  

2 x86 and AMD64 Architecture Differences .................................................... 23  
  2.1 Operating Modes .................................................................................... 23  
  Long Mode .................................................................................................. 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2 Memory Model</td>
<td>24</td>
</tr>
<tr>
<td>2.3 Protection Checks</td>
<td>27</td>
</tr>
<tr>
<td>2.4 Registers</td>
<td>28</td>
</tr>
<tr>
<td>2.5 Instruction Set</td>
<td>29</td>
</tr>
<tr>
<td>2.6 Interrupts and Exceptions</td>
<td>31</td>
</tr>
<tr>
<td>2.7 Hardware Task Switching</td>
<td>37</td>
</tr>
<tr>
<td>2.8 Long-Mode vs. Legacy-Mode Differences</td>
<td>39</td>
</tr>
<tr>
<td>3 System Resources</td>
<td>41</td>
</tr>
<tr>
<td>3.1 System-Control Registers</td>
<td>41</td>
</tr>
<tr>
<td>CR0 Register</td>
<td>42</td>
</tr>
<tr>
<td>CR2 and CR3 Registers</td>
<td>45</td>
</tr>
<tr>
<td>CR4 Register</td>
<td>47</td>
</tr>
<tr>
<td>Additional Control Registers in 64-Bit-Mode</td>
<td>51</td>
</tr>
<tr>
<td>CR8 (Task Priority Register, TPR)</td>
<td>51</td>
</tr>
</tbody>
</table>
4 Segmented Virtual Memory .......................................................... 67
   4.1 Real Mode Segmentation ......................................................... 67
   4.2 Virtual-8086 Mode Segmentation ............................................. 68
   4.3 Protected Mode Segmented-Memory Models ................................ 68
       Multi-Segmented Model ......................................................... 68
       Flat-Memory Model ............................................................ 69
       Segmentation in 64-Bit Mode ................................................. 69
   4.4 Segmentation Data Structures and Registers ............................... 69
   4.5 Segment Selectors and Registers ............................................ 71
       Segment Selectors ............................................................... 71
       Segment Registers ............................................................. 72
       Segment Registers in 64-Bit Mode ......................................... 74
   4.6 Descriptor Tables ............................................................... 75
       Global Descriptor Table ....................................................... 75
       Global Descriptor-Table Register ......................................... 76
       Local Descriptor Table ....................................................... 77
       Local Descriptor-Table Register .......................................... 78
       Interrupt Descriptor Table ................................................. 80
       Interrupt Descriptor-Table Register .................................... 81
   4.7 Legacy Segment Descriptors ................................................... 82
       Descriptor Format ............................................................... 82
       Code-Segment Descriptors ................................................... 84
       Data-Segment Descriptors ................................................... 85
       System Descriptors ............................................................ 87
       Gate Descriptors ............................................................... 88
   4.8 Long-Mode Segment Descriptors ............................................. 90
       Code-Segment Descriptors ................................................... 90
       Data-Segment Descriptors ................................................... 91
       System Descriptors ............................................................ 92
       Gate Descriptors ............................................................... 94
       Long Mode Descriptor Summary .......................................... 96
   4.9 Segment-Protection Overview ................................................ 97
       Privilege-Level Concept ....................................................... 98
       Privilege-Level Types ......................................................... 98
   4.10 Data-Access Privilege Checks .............................................. 99
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessing Data Segments</td>
<td>99</td>
</tr>
<tr>
<td>Accessing Stack Segments</td>
<td>100</td>
</tr>
<tr>
<td>4.11 Control-Transfer Privilege Checks</td>
<td>102</td>
</tr>
<tr>
<td>Direct Control Transfers</td>
<td>102</td>
</tr>
<tr>
<td>Control Transfers Through Call Gates</td>
<td>106</td>
</tr>
<tr>
<td>Return Control Transfers</td>
<td>113</td>
</tr>
<tr>
<td>4.12 Limit Checks</td>
<td>114</td>
</tr>
<tr>
<td>Determining Limit Violations</td>
<td>114</td>
</tr>
<tr>
<td>Data Limit Checks in 64-bit Mode</td>
<td>116</td>
</tr>
<tr>
<td>4.13 Type Checks</td>
<td>116</td>
</tr>
<tr>
<td>Type Checks in Legacy and Compatibility Modes</td>
<td>116</td>
</tr>
<tr>
<td>Long Mode Type Check Differences</td>
<td>117</td>
</tr>
<tr>
<td>5 Page Translation and Protection</td>
<td>119</td>
</tr>
<tr>
<td>5.1 Page Translation Overview</td>
<td>120</td>
</tr>
<tr>
<td>Page-Translation Options</td>
<td>122</td>
</tr>
<tr>
<td>Page-Translation Enable (PG) Bit</td>
<td>122</td>
</tr>
<tr>
<td>Physical-Address Extensions (PAE) Bit</td>
<td>123</td>
</tr>
<tr>
<td>Page-Size Extensions (PSE) Bit</td>
<td>123</td>
</tr>
<tr>
<td>Page-Directory Page Size (PS) Bit</td>
<td>124</td>
</tr>
<tr>
<td>5.2 Legacy-Mode Page Translation</td>
<td>124</td>
</tr>
<tr>
<td>CR3 Register</td>
<td>125</td>
</tr>
<tr>
<td>Normal (Non-PAE) Paging</td>
<td>126</td>
</tr>
<tr>
<td>PAE Paging</td>
<td>128</td>
</tr>
<tr>
<td>5.3 Long-Mode Page Translation</td>
<td>132</td>
</tr>
<tr>
<td>Canonical Address Form</td>
<td>132</td>
</tr>
<tr>
<td>CR3</td>
<td>132</td>
</tr>
<tr>
<td>4-Kbyte Page Translation</td>
<td>133</td>
</tr>
<tr>
<td>2-Mbyte Page Translation</td>
<td>136</td>
</tr>
<tr>
<td>1-Gbyte Page Translation</td>
<td>138</td>
</tr>
<tr>
<td>5.4 Page-Translation-Table Entry Fields</td>
<td>140</td>
</tr>
<tr>
<td>Field Definitions</td>
<td>141</td>
</tr>
<tr>
<td>Notes on Accessed and Dirty Bits</td>
<td>144</td>
</tr>
<tr>
<td>5.5 Translation-Lookaside Buffer (TLB)</td>
<td>144</td>
</tr>
<tr>
<td>Process Context Identifier</td>
<td>145</td>
</tr>
<tr>
<td>Global Pages</td>
<td>145</td>
</tr>
<tr>
<td>TLB Management</td>
<td>146</td>
</tr>
<tr>
<td>5.6 Page-Protection Checks</td>
<td>148</td>
</tr>
<tr>
<td>User/Supervisor (U/S) Bit</td>
<td>149</td>
</tr>
<tr>
<td>Read/Write (R/W) Bit</td>
<td>149</td>
</tr>
<tr>
<td>No Execute (NX) Bit</td>
<td>149</td>
</tr>
<tr>
<td>Write Protect (CR0.WP) Bit</td>
<td>150</td>
</tr>
<tr>
<td>Supervisor-Mode Execution Prevention (CR4.SMEP) Bit</td>
<td>150</td>
</tr>
<tr>
<td>Memory Protection Keys (MPK) Bit</td>
<td>150</td>
</tr>
<tr>
<td>5.7 Protection Across Paging Hierarchy</td>
<td>151</td>
</tr>
<tr>
<td>Access to User Pages when CR0.WP=1</td>
<td>153</td>
</tr>
<tr>
<td>5.8 Effects of Segment Protection</td>
<td>153</td>
</tr>
</tbody>
</table>
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTRRs</td>
<td>195</td>
</tr>
<tr>
<td>Using MTRRs</td>
<td>201</td>
</tr>
<tr>
<td>MTRRs and Page Cache Controls</td>
<td>202</td>
</tr>
<tr>
<td>MTRRs in Multi-Processing Environments</td>
<td>204</td>
</tr>
<tr>
<td>7.8 Page-Attribute Table Mechananism</td>
<td>204</td>
</tr>
<tr>
<td>PAT Register</td>
<td>204</td>
</tr>
<tr>
<td>PAT Indexing</td>
<td>205</td>
</tr>
<tr>
<td>Identifying PAT Support</td>
<td>206</td>
</tr>
<tr>
<td>PAT Accesses</td>
<td>206</td>
</tr>
<tr>
<td>Combined Effect of MTRRs and PAT</td>
<td>207</td>
</tr>
<tr>
<td>PATs in Multi-Processing Environments</td>
<td>208</td>
</tr>
<tr>
<td>Changing Memory Type</td>
<td>208</td>
</tr>
<tr>
<td>7.9 Memory-Mapped I/O</td>
<td>208</td>
</tr>
<tr>
<td>Extended Fixed-Range MTRR Type-Field Encodings</td>
<td>209</td>
</tr>
<tr>
<td>IORRs</td>
<td>210</td>
</tr>
<tr>
<td>IORR Overlapping</td>
<td>212</td>
</tr>
<tr>
<td>Top of Memory</td>
<td>212</td>
</tr>
<tr>
<td>7.10 Secure Memory Encryption</td>
<td>214</td>
</tr>
<tr>
<td>Determining Support for Secure Memory Encryption</td>
<td>214</td>
</tr>
<tr>
<td>Enabling Memory Encryption Extensions</td>
<td>215</td>
</tr>
<tr>
<td>Supported Operating Modes</td>
<td>215</td>
</tr>
<tr>
<td>Page Table Support</td>
<td>215</td>
</tr>
<tr>
<td>I/O Accesses</td>
<td>216</td>
</tr>
<tr>
<td>Restrictions</td>
<td>216</td>
</tr>
<tr>
<td>SMM Interaction</td>
<td>217</td>
</tr>
<tr>
<td>Encrypt-in-Place</td>
<td>217</td>
</tr>
<tr>
<td>8 Exceptions and Interrupts</td>
<td>219</td>
</tr>
<tr>
<td>8.1 General Characteristics</td>
<td>219</td>
</tr>
<tr>
<td>Precision</td>
<td>219</td>
</tr>
<tr>
<td>Instruction Restart</td>
<td>220</td>
</tr>
<tr>
<td>Types of Exceptions</td>
<td>220</td>
</tr>
<tr>
<td>Masking External Interrupts</td>
<td>221</td>
</tr>
<tr>
<td>Masking Floating-Point and Media Instructions</td>
<td>221</td>
</tr>
<tr>
<td>Disabling Exceptions</td>
<td>222</td>
</tr>
<tr>
<td>8.2 Vectors</td>
<td>222</td>
</tr>
<tr>
<td>#DE—Divide-by-Zero-Error Exception (Vector 0)</td>
<td>225</td>
</tr>
<tr>
<td>#DB—Debug Exception (Vector 1)</td>
<td>225</td>
</tr>
<tr>
<td>NMI—Non-Maskable-Interrupt Exception (Vector 2)</td>
<td>226</td>
</tr>
<tr>
<td>#BP—Breakpoint Exception (Vector 3)</td>
<td>226</td>
</tr>
<tr>
<td>#OF—Overflow Exception (Vector 4)</td>
<td>227</td>
</tr>
<tr>
<td>#BR—Bound-Range Exception (Vector 5)</td>
<td>227</td>
</tr>
<tr>
<td>#UD—Invalid-Opcode Exception (Vector 6)</td>
<td>227</td>
</tr>
<tr>
<td>#NM—Device-Not-Available Exception (Vector 7)</td>
<td>228</td>
</tr>
<tr>
<td>#DF—Double-Fault Exception (Vector 8)</td>
<td>228</td>
</tr>
<tr>
<td>Coprocessor-Segment-Overrun Exception (Vector 9)</td>
<td>229</td>
</tr>
<tr>
<td>#TS—Invalid-TSS Exception (Vector 10)</td>
<td>230</td>
</tr>
<tr>
<td>#NP—Segment-Not-Present Exception (Vector 11)</td>
<td>231</td>
</tr>
</tbody>
</table>
#SS—Stack Exception (Vector 12) .................................................. 231
#GP—General-Protection Exception (Vector 13) ............................... 232
#PF—Page-Fault Exception (Vector 14) ........................................ 233
#MF—x87 Floating-Point Exception-Pending (Vector 16) ...................... 234
#AC—Alignment-Check Exception (Vector 17) ................................ 235
#MC—Machine-Check Exception (Vector 18) ................................... 236
#XF—SIMD Floating-Point Exception (Vector 19) ............................... 236
#HV—Hypervisor Injection Exception (Vector 28) ............................ 237
#VC—VMM Communication Exception (Vector 29) ............................. 237
#SX—Security Exception (Vector 30) ........................................... 237
User-Defined Interrupts (Vectors 32–255) ..................................... 237

8.3 Exceptions During a Task Switch ............................................. 238
8.4 Error Codes ................................................................. 238
Selector-Error Code ................................................................... 238
Page-Fault Error Code ............................................................. 239

8.5 Priorities .............................................................................. 240
Floating-Point Exception Priorities ................................................ 241
External Interrupt Priorities ......................................................... 243

8.6 Real-Mode Interrupt Control Transfers .................................... 244
8.7 Legacy Protected-Mode Interrupt Control Transfers ................. 245
Locating the Interrupt Handler ..................................................... 246
Interrupt To Same Privilege ......................................................... 247
Interrupt To Higher Privilege ....................................................... 248
Privilege Checks ........................................................................ 249
Returning From Interrupt Procedures ........................................... 252

8.8 Virtual-8086 Mode Interrupt Control Transfers ......................... 252
Protected-Mode Handler Control Transfer ...................................... 253
Virtual-8086 Handler Control Transfer .......................................... 255

8.9 Long-Mode Interrupt Control Transfers .................................... 255
Interrupt Gates and Trap Gates ..................................................... 255
Locating the Interrupt Handler ..................................................... 256
Interrupt Stack Frame ................................................................ 257
Interrupt-Stack Table ................................................................. 259
Returning From Interrupt Procedures ........................................... 261

8.10 Virtual Interrupts ................................................................. 262
Virtual-8086 Mode Extensions ...................................................... 262
Protected Mode Virtual Interrupts ................................................ 265
Effect of Instructions that Modify EFLAGS.IF ................................. 265

9 Machine Check Architecture ...................................................... 269
9.1 Introduction ......................................................................... 269
Reliability, Availability, and Serviceability ................................... 269
Error Detection, Logging, and Reporting ..................................... 270
Error Recovery ................................................................. 272

9.2 Determining Machine-Check Architecture Support .................... 273
9.3 Machine Check Architecture MSRs ......................................... 273
Global Status and Control Registers ............................................. 274
Error-Reporting Register Banks .................................................. 277
Contents

11 SSE, MMX, and x87 Programming ................................................. 309
  11.1 Overview of System-Software Considerations ............................ 309
  11.2 Determining Media and x87 Feature Support .............................. 309
  11.3 Enabling SSE Instructions .................................................. 311
      Enabling Legacy SSE Instruction Execution ............................... 311
      Enabling Extended SSE Instruction Execution ............................ 311
      SIMD Floating-Point Exception Handling ................................. 312
  11.4 Media and x87 Processor State ............................................. 312
      SSE Execution Unit State .................................................. 312
      MMX Execution Unit State .............................................. 313
      x87 Execution Unit State ............................................. 314
      Saving Media and x87 Execution Unit State .............................. 316
  11.5 XSAVE/XRSTOR Instructions .............................................. 323
      CPUID Enhancements .................................................... 323
      XFEATURE_ENABLED_MASK ............................................. 323
      Extended Save Area .................................................... 324
      Instruction Functions ................................................. 325
      YMM States and Supported Operating Modes .............................. 325
      Extended SSE Execution State Management .............................. 325
      Saving Processor State .............................................. 327
      Restoring Processor State ............................................. 327
      MXCSR State Management ............................................... 327

10 System-Management Mode ....................................................... 291
  10.1 SMM Differences .......................................................... 291
  10.2 SMM Resources ............................................................ 292
      SMRAM ................................................................. 292
      SMBASE Register ...................................................... 293
      SMRAM State-Save Area ............................................... 294
      SMM-Revision Identifier .............................................. 298
      SMRAM Protected Areas ............................................... 299
  10.3 Using SMM ................................................................. 301
      System-Management Interrupt (SMI) .................................... 301
      SMM Operating-Environment ........................................... 301
      Exceptions and Interrupts ............................................. 302
      Invalidating the Caches ............................................... 303
      Saving Additional Processor State .................................... 303
      Operating in Protected Mode and Long Mode ........................... 304
      Auto-Halt Restart ..................................................... 304
      I/O Instruction Restart ................................................ 305
  10.4 Leaving SMM ............................................................... 306
  10.5 Multiprocessor Considerations ............................................ 307

9 System-Management Mode ....................................................... 285
  9.4 Initializing the Machine-Check Mechanism ................................ 285
  9.5 Using MCA Features ........................................................ 286
      Determining the Scope of Detected Errors .............................. 287
      Handling Machine Check Exceptions ................................. 287
      Reporting Corrected Errors ........................................... 289

x

Contents
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Task Management</td>
<td>335</td>
</tr>
<tr>
<td>12.1</td>
<td>Hardware Multitasking Overview</td>
<td>335</td>
</tr>
<tr>
<td>12.2</td>
<td>Task-Management Resources</td>
<td>336</td>
</tr>
<tr>
<td></td>
<td>TSS Selector</td>
<td>338</td>
</tr>
<tr>
<td></td>
<td>TSS Descriptor</td>
<td>338</td>
</tr>
<tr>
<td></td>
<td>Task Register</td>
<td>339</td>
</tr>
<tr>
<td></td>
<td>Legacy Task-State Segment</td>
<td>341</td>
</tr>
<tr>
<td></td>
<td>64-Bit Task State Segment</td>
<td>345</td>
</tr>
<tr>
<td></td>
<td>Task Gate Descriptor (Legacy Mode Only)</td>
<td>348</td>
</tr>
<tr>
<td>12.3</td>
<td>Hardware Task-Management in Legacy Mode</td>
<td>348</td>
</tr>
<tr>
<td></td>
<td>Task Memory-Mapping</td>
<td>348</td>
</tr>
<tr>
<td></td>
<td>Switching Tasks</td>
<td>349</td>
</tr>
<tr>
<td></td>
<td>Task Switches Using Task Gates</td>
<td>351</td>
</tr>
<tr>
<td></td>
<td>Nesting Tasks</td>
<td>353</td>
</tr>
<tr>
<td>13</td>
<td>Software Debug and Performance Resources</td>
<td>355</td>
</tr>
<tr>
<td>13.1</td>
<td>Software-Debug Resources</td>
<td>356</td>
</tr>
<tr>
<td></td>
<td>Debug Registers</td>
<td>356</td>
</tr>
<tr>
<td></td>
<td>Setting Breakpoints</td>
<td>363</td>
</tr>
<tr>
<td></td>
<td>Using Breakpoints</td>
<td>365</td>
</tr>
<tr>
<td></td>
<td>Single Stepping</td>
<td>368</td>
</tr>
<tr>
<td></td>
<td>Breakpoint Instruction (INT3)</td>
<td>368</td>
</tr>
<tr>
<td></td>
<td>Control-Transfer Breakpoint Features</td>
<td>368</td>
</tr>
<tr>
<td></td>
<td>Debug Breakpoint Address Masking</td>
<td>370</td>
</tr>
<tr>
<td>13.2</td>
<td>Performance Monitoring Counters</td>
<td>370</td>
</tr>
<tr>
<td></td>
<td>Performance Counter MSRs</td>
<td>371</td>
</tr>
<tr>
<td></td>
<td>Detecting Hardware Support for Performance Counters</td>
<td>377</td>
</tr>
<tr>
<td></td>
<td>Using Performance Counters</td>
<td>377</td>
</tr>
<tr>
<td></td>
<td>Time-Stamp Counter</td>
<td>377</td>
</tr>
<tr>
<td>13.3</td>
<td>Instruction-Based Sampling</td>
<td>379</td>
</tr>
<tr>
<td></td>
<td>IBS Fetch Sampling</td>
<td>379</td>
</tr>
<tr>
<td></td>
<td>IBS Fetch Sampling Registers</td>
<td>380</td>
</tr>
<tr>
<td></td>
<td>IBS Execution Sampling</td>
<td>383</td>
</tr>
<tr>
<td></td>
<td>IBS Execution Sampling Registers</td>
<td>384</td>
</tr>
<tr>
<td>13.4</td>
<td>Lightweight Profiling</td>
<td>392</td>
</tr>
<tr>
<td></td>
<td>Overview</td>
<td>392</td>
</tr>
<tr>
<td></td>
<td>Events and Event Records</td>
<td>396</td>
</tr>
<tr>
<td></td>
<td>Detecting LWP</td>
<td>406</td>
</tr>
<tr>
<td></td>
<td>LWP Registers</td>
<td>410</td>
</tr>
<tr>
<td></td>
<td>LWP Instructions</td>
<td>412</td>
</tr>
<tr>
<td></td>
<td>LWP Control Block</td>
<td>416</td>
</tr>
<tr>
<td></td>
<td>XSAVE/XRSTOR</td>
<td>426</td>
</tr>
<tr>
<td></td>
<td>Implementation Notes</td>
<td>430</td>
</tr>
<tr>
<td>14</td>
<td>Processor Initialization and Long Mode Activation</td>
<td>435</td>
</tr>
<tr>
<td>14.1</td>
<td>Processor Initialization</td>
<td>435</td>
</tr>
</tbody>
</table>
15.11 MSR Intercepts .............................................. 472
15.12 Exception Intercepts ....................................... 474
  #DE (Divide By Zero) ........................................ 474
  #DB (Debug) .................................................. 474
  Vector 2 (Reserved) .......................................... 475
  #BP (Breakpoint) ............................................. 475
  #OF (Overflow) ................................................ 475
  #BR (Bound-Range) ........................................... 475
  #UD (Invalid Opcode) ....................................... 475
  #NM (Device-Not-Available) ............................... 475
  #DF (Double Fault) ......................................... 475
  Vector 9 (Reserved) .......................................... 475
  #TS (Invalid TSS) ........................................... 475
  #NP (Segment Not Present) ................................ 476
  #SS (Stack Fault) ............................................ 476
  #GP (General Protection) .................................. 476
  #PF (Page Fault) ............................................. 476
  #MF (X87 Floating Point) .................................. 476
  #AC (Alignment Check) ..................................... 476
  #MC (Machine Check) ........................................ 476
  #XF (SIMD Floating Point) .................................. 476
15.13 Interrupt Intercepts ....................................... 477
  INTR Intercept ................................................ 477
  NMI Intercept .................................................. 477
  SMI Intercept .................................................. 477
  INIT Intercept .................................................. 478
  Virtual Interrupt Intercept ................................ 478
15.14 Miscellaneous Intercepts .................................. 478
  Task Switch Intercept ....................................... 478
  Ferr_Freeze Intercept ........................................ 479
  Shutdown Intercept ........................................... 479
  Pause Intercept Filtering ................................... 479
15.15 VMCB State Caching ........................................ 480
  VMCB Clean Bits ............................................... 480
  Guidelines for Clearing VMCB Clean Bits ............... 480
  VMCB Clean Field ............................................. 481
15.16 TLB Control .................................................. 482
  TLB Flush ...................................................... 483
  Invalidate Page, Alternate ASID ......................... 484
15.17 Global Interrupt Flag, STGI and CLGI Instructions .... 484
15.18 VMMCALL Instruction ....................................... 485
15.19 Paged Real Mode ............................................ 485
15.20 Event Injection .............................................. 485
15.21 Interrupt and Local APIC Support ....................... 487
  Physical (INTR) Interrupt Masking in EFLAGS. ......... 487
15.22 SMM Support ................................................................. 490
15.23 Last Branch Record Virtualization ................................. 492
15.24 External Access Protection ......................................... 493
15.25 Nested Paging ............................................................. 500
15.26 Security ................................................................. 508
15.27 Secure Startup with SKINIT ........................................ 508

Virtualizing APIC.TPR ....................................................... 487
TPR Access in 32-Bit Mode ............................................... 487
Injecting Virtual (INTR) Interrupts .................................... 488
Interrupt Shadows ......................................................... 489
Virtual Interrupt Intercept .............................................. 489
Interrupt Masking in Local APIC ....................................... 489
INIT Support ............................................................... 489
NMI Support .............................................................. 490
Sources of SMI ............................................................. 490
Response to SMI ........................................................... 491
Containerizing Platform SMM .......................................... 491

Last Branch Record Virtualization .................................... 492
Hardware Acceleration for LBR Virtualization ..................... 493
LBR Virtualization CPUID Feature Detection ....................... 493

External Access Protection .............................................. 493
Device IDs and Protection Domains .................................. 493
Device Exclusion Vector (DEV) ....................................... 494
Access Checking .......................................................... 494
DEV Capability Block .................................................... 496
DEV Register Access Mechanism ..................................... 496
DEV Control and Status Registers ................................... 497
Unauthorized Access Logging ......................................... 499
Secure Initialization Support .......................................... 499

Nested Paging ............................................................. 500
Traditional Paging versus Nested Paging ......................... 500
Replicated State .......................................................... 501
Enabling Nested Paging ................................................ 502
Nested Paging and VMRUN/#VMEXIT .............................. 502
Nested Table Walk ........................................................ 503
Nested versus Guest Page Faults, Fault Ordering ................ 503
Combining Nested and Guest Attributes .......................... 504
Combining Memory Types, MTRRs .................................. 505
Page Splintering ........................................................... 506
Legacy PAE Mode ........................................................ 507
A20 Masking .............................................................. 507
Detecting Nested Paging Support ..................................... 507
Guest Mode Execute Trap Extension ................................. 507

Secure Startup with SKINIT ........................................... 508
Secure Loader ............................................................. 508
Secure Loader Image ..................................................... 509
Secure Loader Block ...................................................... 509
Trusted Platform Module ................................................ 510
System Interface, Memory Controller and I/O Hub Logic ...... 511
SKINIT Operation .......................................................... 511
SL Abort ................................................................. 512
Secure Multiprocessor Initialization .................................................. 512
15.28 Security Exception (#SX) ......................................................... 513
15.29 Advanced Virtual Interrupt Controller ....................................... 514
Introduction .................................................................................. 514
Architectural Definition .................................................................. 515
15.30 SVM Related MSRs .................................................................. 534
VM_CR MSR (C001_0114h) ............................................................. 534
IGNNE MSR (C001_0115h) ............................................................... 535
SMV_CTL MSR (C001_0116h) ......................................................... 535
VM_HSAVE_PA MSR (C001_0117h) .................................................. 536
TSC Ratio MSR (C000_0104h) ......................................................... 536
15.31 SVM-Lock ............................................................................... 537
SVM_KEY MSR (C001_0118h) ......................................................... 537
15.32 SMM-Lock ............................................................................... 538
SmmLock Bit — HWCR[0] ................................................................. 538
SMM_KEY MSR (C001_0119h) ......................................................... 538
15.33 Nested Virtualization ................................................................ 538
VMSAVE and VMLOAD Virtualization ............................................. 539
Virtual GIF (VGIF) .......................................................................... 539
15.34 Secure Encrypted Virtualization ................................................ 539
Determining Support for SEV ............................................................ 540
Key Management ........................................................................ 540
Enabling SEV ............................................................................... 541
Supported Operating Modes ............................................................. 541
SEV Encryption Behavior ................................................................. 541
Page Table Support ....................................................................... 542
Restrictions .................................................................................. 543
SEV Interaction with SME ............................................................... 543
Page Flush MSR ........................................................................ 545
SEV_STATUS MSR ....................................................................... 545
Virtual Transparent Encryption (VTE) ............................................. 546
15.35 Encrypted State (SEV-ES) .......................................................... 546
Determining Support for SEV-ES .................................................... 547
Enabling SEV-ES .......................................................................... 547
SEV-ES Overview ....................................................................... 547
Types of Exits ............................................................................... 548
#VC Exception ........................................................................... 549
VMGExit ................................................................................... 551
GHCB ........................................................................................... 551
VMRUN ...................................................................................... 551
Automatic Exits ........................................................................ 552
Control Register Write Traps ............................................................ 552
15.36 Secure Nested Paging (SEV-SNP) .............................................. 553
Determining Support for SEV-SNP .................................................. 553
Enabling SEV-SNP ....................................................................... 554
Reverse Map Table ....................................................................... 554
Initializing the RMP ..................................................................... 555

Contents  xv
16 Advanced Programmable Interrupt Controller (APIC) .................................................. 567
16.1 Sources of Interrupts to the Local APIC ................................................................. 568
16.2 Interrupt Control .............................................
16.3 Local APIC ............................................................... 569
       Local APIC Enable ........................................... 569
       APIC Registers .................................................. 570
       Local APIC ID ................................................... 571
       APIC Version Register ...................................... 572
       Extended APIC Feature Register ......................... 573
       Extended APIC Control Register ......................... 573
16.4 Local Interrupts .................................................. 574
       APIC Timer Interrupt ...................................... 576
       Local Interrupts LINT0 and LINT1 ..................... 578
16.5 Interprocessor Interrupts (IPI) ................................................................. 581
16.6 Local APIC Handling of Interrupts ...................................................... 585
       Receiving System and IPI Interrupts ................. 585
       Lowest Priority Messages and Arbitration ........... 586
       Accepting System and IPI Interrupts ................ 587
       Selecting and Handling Interrupts ..................... 590
16.7 SVM Support for Interrupts and the Local APIC .............................................. 592
       Specific End of Interrupt Register ..................... 593
       Interrupt Enable Register ................................. 593

17 Hardware Performance Monitoring and Control .................................................... 595
17.1 P-State Control .................................................. 595
17.2 Core Performance Boost ............................................. 597
17.3 Determining Processor Effective Frequency ............................................ 598
       Actual Performance Frequency Clock Count (APERF) 599
       Maximum Performance Frequency Clock Count (MPERF) 599
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPERF Read-only (MperfReadOnly)</td>
<td>600</td>
</tr>
<tr>
<td>17.4 Processor Feedback Interface</td>
<td>600</td>
</tr>
<tr>
<td>17.5 Processor Core Power Reporting</td>
<td>600</td>
</tr>
<tr>
<td>Processor Facilities</td>
<td>600</td>
</tr>
<tr>
<td>Software Algorithm</td>
<td>601</td>
</tr>
<tr>
<td>Appendix A MSR Cross-Reference</td>
<td>603</td>
</tr>
<tr>
<td>A.1 MSR Cross-Reference by MSR Address</td>
<td>603</td>
</tr>
<tr>
<td>A.2 System-Software MSRs</td>
<td>607</td>
</tr>
<tr>
<td>A.3 Memory-Typing MSRs</td>
<td>609</td>
</tr>
<tr>
<td>A.4 Machine-Check MSRs</td>
<td>611</td>
</tr>
<tr>
<td>A.5 Software-Debug MSRs</td>
<td>612</td>
</tr>
<tr>
<td>A.6 Performance-Monitoring MSRs</td>
<td>613</td>
</tr>
<tr>
<td>A.7 Secure Virtual Machine MSRs</td>
<td>614</td>
</tr>
<tr>
<td>A.8 System Management Mode MSRs</td>
<td>616</td>
</tr>
<tr>
<td>A.9 CPUID Name MSR Cross-Reference</td>
<td>616</td>
</tr>
<tr>
<td>Appendix B Layout of VMCB</td>
<td>617</td>
</tr>
<tr>
<td>Appendix C SVM Intercept Exit Codes</td>
<td>631</td>
</tr>
<tr>
<td>Appendix D SMM Containerization</td>
<td>635</td>
</tr>
<tr>
<td>D.1 SMM Containerization Pseudocode</td>
<td>635</td>
</tr>
<tr>
<td>Appendix E OS-Visible Workarounds</td>
<td>641</td>
</tr>
<tr>
<td>E.1 Erratum Process Overview</td>
<td>643</td>
</tr>
<tr>
<td>Index</td>
<td>645</td>
</tr>
</tbody>
</table>
Figures

Figure 1-1. Segmented-Memory Model ............................................................... 6
Figure 1-2. Flat Memory Model ................................................................. 7
Figure 1-3. Paged Memory Model ............................................................... 8
Figure 1-4. 64-Bit Flat, Paged-Memory Model .............................................. 9
Figure 1-5. Real-Address Memory Model .................................................... 10
Figure 1-6. Operating Modes of the AMD64 Architecture ............................. 12
Figure 1-7. System Registers ................................................................. 16
Figure 1-8. System-Data Structures ........................................................... 18
Figure 3-1. Control Register 0 (CR0) .......................................................... 43
Figure 3-2. Control Register 2 (CR2)—Legacy-Mode .................................. 46
Figure 3-3. Control Register 2 (CR2)—Long Mode ...................................... 46
Figure 3-4. Control Register 3 (CR3)—Legacy-Mode Non-PAE Paging .......... 46
Figure 3-5. Control Register 3 (CR3)—Legacy-Mode PAE Paging .................. 46
Figure 3-6. Control Register 3 (CR3)—Long Mode ....................................... 47
Figure 3-7. RFLAGS Register ................................................................. 52
Figure 3-8. Extended Feature Enable Register (EFER) ............................... 56
Figure 3-9. AMD64 Architecture Model-Specific Registers ......................... 59
Figure 3-10. System-Configuration Register (SYSCFG) ............................... 60
Figure 4-1. Segmentation Data Structures .................................................. 70
Figure 4-2. Segment and Descriptor-Table Registers ................................... 71
Figure 4-3. Segment Selector ................................................................. 71
Figure 4-4. Segment-Register Format ....................................................... 73
Figure 4-5. FS and GS Segment-Register Format—64-Bit Mode .................. 74
Figure 4-6. Global and Local Descriptor-Table Access .................................. 76
Figure 4-7. GDTR and IDTR Format—Legacy Modes ................................... 77
Figure 4-8. GDTR and IDTR Format—Long Mode ....................................... 77
Figure 4-9. Relationship between the LDT and GDT .................................... 78
Figure 4-10. LDTR Format—Legacy Mode .................................................. 79
Figure 4-11. LDTR Format—Long Mode ..................................................... 79
Figure 4-12. Indexing an IDT ................................................................. 81
Figure 4-13. Generic Segment Descriptor—Legacy Mode ............................ 82
Figure 4-14. Code-Segment Descriptor—Legacy Mode ................................................................. 84
Figure 4-15. Data-Segment Descriptor—Legacy Mode ............................................................... 85
Figure 4-16. LDT and TSS Descriptor—Legacy/Compatibility Modes ........................................ 88
Figure 4-17. Call-Gate Descriptor—Legacy Mode ................................................................. 89
Figure 4-18. Interrupt-Gate and Trap-Gate Descriptors—Legacy Mode ........................................ 89
Figure 4-19. Task-Gate Descriptor—Legacy Mode ............................................................... 89
Figure 4-20. Code-Segment Descriptor—Long Mode ............................................................... 90
Figure 4-21. Data-Segment Descriptor—Long Mode ............................................................... 91
Figure 4-22. System-Segment Descriptor—64-Bit Mode .......................................................... 93
Figure 4-23. Call-Gate Descriptor—Long Mode ............................................................... 94
Figure 4-24. Interrupt-Gate and Trap-Gate Descriptors—Long Mode ........................................ 95
Figure 4-25. Privilege-Level Relationships ................................................................................... 98
Figure 4-26. Data-Access Privilege-Check Examples ............................................................... 100
Figure 4-27. Stack-Access Privilege-Check Examples ............................................................. 101
Figure 4-28. Nonconforming Code-Segment Privilege-Check Examples ...................................... 104
Figure 4-29. Conforming Code-Segment Privilege-Check Examples ........................................ 105
Figure 4-30. Legacy-Mode Call-Gate Transfer Mechanism ....................................................... 106
Figure 4-31. Long-Mode Call-Gate Access Mechanism ........................................................... 107
Figure 4-32. Privilege-Check Examples for Call Gates ............................................................ 109
Figure 4-33. Legacy-Mode 32-Bit Stack Switch, with Parameters ................................................ 111
Figure 4-34. 32-Bit Stack Switch, No Parameters—Legacy Mode ................................................ 111
Figure 4-35. Stack Switch—Long Mode .................................................................................. 112
Figure 5-1. Virtual to Physical Address Translation—Long Mode ............................................ 121
Figure 5-2. Control Register 3 (CR3)—Non-PAE Paging Legacy-Mode ......................................... 125
Figure 5-3. Control Register 3 (CR3)—PAE Paging Legacy-Mode ................................................ 125
Figure 5-4. 4-Kbyte Non-PAE Page Translation—Legacy Mode ................................................ 126
Figure 5-5. 4-Kbyte PDE—Non-PAE Paging Legacy-Mode ....................................................... 127
Figure 5-6. 4-Kbyte PTE—Non-PAE Paging Legacy-Mode ........................................................... 127
Figure 5-7. 4-Mbyte Page Translation—Non-PAE Paging Legacy-Mode ........................................ 128
Figure 5-8. 4-Mbyte PDE—Non-PAE Paging Legacy-Mode ....................................................... 128
Figure 5-9. 4-Kbyte PAE Page Translation—Legacy Mode ........................................................ 129
Figure 5-10. 4-Kbyte PDPE—PAE Paging Legacy-Mode ............................................................ 130
Figure 5-11. 4-Kbyte PDE—PAE Paging Legacy-Mode .............................................................. 130
Figure 5-12. 4-Kbyte PTE—PAE Paging Legacy-Mode ......................................................... 130
Figure 5-13. 2-Mbyte PAE Page Translation—Legacy Mode ....................................................... 131
Figure 5-14. 2-Mbyte PDPE—PAE Paging Legacy-Mode ......................................................... 131
Figure 5-15. 2-Mbyte PDE—PAE Paging Legacy-Mode .......................................................... 132
Figure 5-16. Control Register 3 (CR3)—Long Mode ............................................................. 133
Figure 5-17. 4-Kbyte Page Translation—Long Mode .............................................................. 134
Figure 5-18. 4-Kbyte PML4E—Long Mode ............................................................................. 135
Figure 5-19. 4-Kbyte PDPE—Long Mode ................................................................................ 135
Figure 5-20. 4-Kbyte PDE—Long Mode .................................................................................. 135
Figure 5-21. 4-Kbyte PTE—Long Mode .................................................................................. 136
Figure 5-22. 2-Mbyte Page Translation—Long Mode ............................................................. 137
Figure 5-23. 2-Mbyte PML4E—Long Mode ............................................................................. 138
Figure 5-24. 2-Mbyte PDPE—Long Mode ............................................................................... 138
Figure 5-25. 2-Mbyte PDE—Long Mode ................................................................................ 138
Figure 5-26. 1-Gbyte Page Translation—Long Mode ............................................................. 139
Figure 5-27. 1-Gbyte PML4E—Long Mode ............................................................................. 140
Figure 5-28. 1-Gbyte PDPE—Long Mode ............................................................................... 140
Figure 5-29. PKRU Register ................................................................................................. 151
Figure 6-1. STAR, LSTAR, CSTAR, and MASK MSRs ............................................................. 159
Figure 6-2. SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP MSRs ............................................ 161
Figure 7-1. Processor and Memory System ............................................................................. 168
Figure 7-2. MOESI State Transitions ..................................................................................... 176
Figure 7-3. Cache Organization Example .............................................................................. 186
Figure 7-4. MTRR Mapping of Physical Memory ..................................................................... 196
Figure 7-5. Fixed-Range MTRR ............................................................................................. 197
Figure 7-6. MTRRphysBase[n] Register ................................................................................. 199
Figure 7-7. MTRRphysMask[n] Register ................................................................................. 199
Figure 7-8. MTRRdefType Register Format .......................................................................... 201
Figure 7-9. MTRR Capability Register Format ....................................................................... 202
Figure 7-10. PAT Register ...................................................................................................... 204
Figure 7-11. Extended MTRR Type-Field Format (Fixed-Range MTRRs) ................................. 209
Figure 7-12. IORRBase[n] Register ....................................................................................... 211
Figure 7-13. IORRMask[n] Register ....................................................................................... 212
Figure 7-14. Memory Organization Using Top-of-Memory Registers .............................. 213
Figure 7-15. Top-of-Memory Registers (TOP_MEM, TOP_MEM2) ................................. 214
Figure 7-16. Encrypted Memory Accesses ................................................................. 216
Figure 8-1. Control Register 2 (CR2) ........................................................................... 234
Figure 8-2. Selector Error Code ................................................................................... 239
Figure 8-3. Page-Fault Error Code .............................................................................. 239
Figure 8-4. Task Priority Register (CR8) ................................................................. 243
Figure 8-5. Real-Mode Interrupt Control Transfer ..................................................... 244
Figure 8-6. Stack After Interrupt in Real Mode ............................................................ 245
Figure 8-7. Protected-Mode Interrupt Control Transfer ............................................... 247
Figure 8-8. Stack After Interrupt to Same Privilege Level ............................................. 248
Figure 8-9. Stack After Interrupt to Higher Privilege .................................................... 249
Figure 8-10. Privilege-Check Examples for Interrupts .................................................... 251
Figure 8-11. Stack After Virtual-8086 Mode Interrupt to Protected Mode ................. 254
Figure 8-12. Long-Mode Interrupt Control Transfer .................................................... 256
Figure 8-13. Long-Mode Stack After Interrupt—Same Privilege ................................. 258
Figure 8-14. Long-Mode Stack After Interrupt—Higher Privilege ............................... 259
Figure 8-15. Long-Mode IST Mechanism ................................................................. 260
Figure 9-1. MCG_CAP Register ................................................................................. 274
Figure 9-2. MCG_STATUS Register .......................................................................... 275
Figure 9-3. MCG_CTL Register ................................................................................ 276
Figure 9-4. CPU Watchdog Timer Register Format ..................................................... 276
Figure 9-5. MCi_CTL Register .................................................................................. 279
Figure 9-6. MCi_STATUS Register ........................................................................... 280
Figure 9-7. MCi_MISC1 Addressing ......................................................................... 283
Figure 9-8. Miscellaneous Information Register (Thresholding Register Format) .......... 284
Figure 10-1. Default SMRAM Memory Map .............................................................. 293
Figure 10-2. SMBASE Register ............................................................................... 293
Figure 10-3. SMM-Revision Identifier ..................................................................... 299
Figure 10-4. SMM_ADDR Register Format .................................................................. 300
Figure 10-5. SMM_MASK Register Format ................................................................ 300
Figure 10-6. I/O Instruction Restart Dword ............................................................... 306
Figure 11-1. SSE Execution Unit State .................................................................... 313
Figure 11-2. MMX Execution Unit State ......................................................... 314
Figure 11-3. x87 Execution Unit State ......................................................... 316
Figure 11-4. FSAVE/FNSAVE Image (32-Bit, Protected Mode) ..................... 318
Figure 11-5. FSAVE/FNSAVE Image (32-Bit, Real/Virtual-8086 Modes) ...... 319
Figure 11-6. FSAVE/FNSAVE Image (16-Bit, Protected Mode) ................... 320
Figure 11-7. FSAVE/FNSAVE Image (16-Bit, Real/Virtual-8086 Modes) ...... 321
Figure 11-8. XFEATURE_ENABLED_MASK Register (XCR0) ....................... 324
Figure 11-9. FXSAVE and FXRSTOR Image (64-bit Mode) .......................... 329
Figure 11-10. FXSAVE and FXRSTOR Image (Non-64-bit Mode) ................. 330
Figure 12-1. Task-Management Resources ................................................. 337
Figure 12-2. Task-Segment Selector .......................................................... 338
Figure 12-3. TR Format, Legacy Mode ....................................................... 339
Figure 12-4. TR Format, Long Mode .......................................................... 340
Figure 12-5. Relationship between the TSS and GDT ................................. 340
Figure 12-6. Legacy 32-bit TSS ................................................................. 342
Figure 12-7. I/O-Permission Bitmap Example ............................................. 345
Figure 12-8. Long Mode TSS Format .......................................................... 347
Figure 12-9. Task-Gate Descriptor, Legacy Mode Only .................................. 348
Figure 12-10. Privilege-Check Examples for Task Gates ............................... 352
Figure 13-1. Address-Breakpoint Registers (DR0–DR3) ............................... 357
Figure 13-2. Debug-Status Register (DR6) .................................................. 358
Figure 13-3. Debug-Control Register (DR7) ............................................... 359
Figure 13-4. Debug-Control MSR (DebugCtl) .............................................. 362
Figure 13-5. Control-Transfer Recording MSRs ......................................... 363
Figure 13-6. Performance Counter Format .................................................. 372
Figure 13-7. Core Performance Event-Select Register (PerfEvtSeln). ............. 373
Figure 13-8. Northbridge Performance Event-Select Register (NB_PerfEvtSeln) . 375
Figure 13-9. L2 Cache Performance Event-Select Register (L2I_PerfEvtSeln) . 376
Figure 13-10. Time-Stamp Counter (TSC) .................................................... 378
Figure 13-11. IBS Fetch Control Register(IbsFetchCtl) .................................. 381
Figure 13-12. IBS Fetch Linear Address Register (IbsFetchLinAd) ................. 382
Figure 13-13. IBS Fetch Physical Address Register (IbsFetchPhysAd) ............ 383
Figure 13-14. IBS Execution Control Register (IbsOpCtl) ............................. 385
Figure 13-15. IBS Op Linear Address Register (IbsOpRip) .................................................... 386
Figure 13-16. IBS Op Data 1 Register (IbsOpData1) ................................................................. 387
Figure 13-17. IBS Op Data 3 Register (IbsOpData3) ................................................................. 389
Figure 13-18. IBS Data Cache Linear Address Register (IbsDeLinAd) ................................. 391
Figure 13-19. IBS Data Cache Physical Address Register (IbsDcPhysAd) ......................... 391
Figure 13-20. IBS Branch Target Address Register (IbsBrTarget) ......................................... 392
Figure 13-21. Generic Event Record ............................................................. 397
Figure 13-22. Programmed Value Sample Event Record ................................................. 398
Figure 13-23. Instructions Retired Event Record ................................................................. 399
Figure 13-24. Branch Retired Event Record ................................................................. 401
Figure 13-25. DCache Miss Event Record ................................................................. 403
Figure 13-26. CPU Clocks not Halted Event Record .......................................................... 404
Figure 13-27. CPU Reference Clocks not Halted Event Record ....................................... 405
Figure 13-28. Programmed Event Record ................................................................. 406
Figure 13-29. LWP_CFG—Lightweight Profiling Features MSR ....................................... 411
Figure 13-30. LWPCB—Lightweight Profiling Control Block ......................................... 418
Figure 13-31. LWPCB Flags ......................................................................................... 422
Figure 13-32. LWPCB Filters ......................................................................................... 423
Figure 13-33. XSAVE Area for LWP ........................................................................ 427
Figure 15-1. EXITINTINFO for All Intercepts ............................................................... 464
Figure 15-2. EXITINFO1 for IOIO Intercept .................................................................... 472
Figure 15-3. EXITINFO1 for SMI Intercept ..................................................................... 478
Figure 15-4. Layout of VMCB Clean Field ..................................................................... 481
Figure 15-5. EVENTINJ Field in the VMCB ................................................................ 486
Figure 15-6. Host Bridge DMA Checking ....................................................................... 495
Figure 15-7. Format of DEV_OP Register (in PCI Config Space) .................................... 496
Figure 15-8. Format of DEV_CAP Register (in PCI Config Space) .................................. 497
Figure 15-9. Format of DEV_BASE_HI[n] Registers ...................................................... 498
Figure 15-10. Format of DEV_BASE_LO[n] Registers ...................................................... 498
Figure 15-11. Format of DEV_MAP[n] Registers ............................................................. 499
Figure 15-12. Address Translation with Traditional Paging .............................................. 500
Figure 15-13. Address Translation with Nested Paging .................................................... 501
Figure 15-14. SLB Example Layout ............................................................................. 510
Figure 15. vAPIC Backing Page Access ................................................................. 516
Figure 15-16. Virtual APIC Task Priority Register Synchronization .......................... 519
Figure 15-17. Physical APIC ID Table Entry ......................................................... 524
Figure 15-18. Physical APIC Table in Memory....................................................... 525
Figure 15-19. Logical APIC ID Table Entry ......................................................... 526
Figure 15-20. Logical APIC ID Table Format, Flat Mode ....................................... 527
Figure 15-21. Logical APIC ID Table Format, Cluster Mode .................................... 528
Figure 15-22. Doorbell Register, MSR C001_011Bh ............................................. 531
Figure 15-23. EXITINFO1 ..................................................................................... 532
Figure 15-24. EXITINFO2 ..................................................................................... 532
Figure 15-25. Layout of VM_CR MSR (C001_0114h) .......................................... 534
Figure 15-26. Layout of SMM_CTL MSR (C001_0116h) ....................................... 535
Figure 15-27. TSC Ratio MSR (C000_0104h) ...................................................... 537
Figure 15-28. Guest Data Request ...................................................................... 543
Figure 15-29. EXAMPLE #VC FLOW ................................................................. 550

Figure 16-1. Block Diagram of a Typical APIC Implementation .............................. 567
Figure 16-2. APIC Base Address Register (MSR 0000_001Bh) ............................. 570
Figure 16-3. APIC ID Register (APIC Offset 20h) ................................................. 572
Figure 16-4. APIC Version Register (APIC Offset 30h) ......................................... 572
Figure 16-5. Extended APIC Feature Register (APIC Offset 400h) ....................... 573
Figure 16-6. Extended APIC Control Register (APIC Offset 410h) ....................... 574
Figure 16-7. General Local Vector Table Register Format .................................. 575
Figure 16-8. APIC Timer Local Vector Table Register (APIC Offset 320h) ......... 576
Figure 16-9. Timer Current Count Register (APIC Offset 390h) ......................... 577
Figure 16-10. Timer Initial Count Register (APIC Offset 380h) ............................ 577
Figure 16-11. Divide Configuration Register (APIC Offset 3E0h) ..................... 577
Figure 16-12. Local Interrupt 0/1 (LINT0/1) Local Vector Table Register (APIC Offset 350h/360h) 578
Figure 16-13. Performance Monitor Counter Local Vector Table Register (APIC Offset 340h) 579
Figure 16-14. Thermal Sensor Local Vector Table Register (APIC Offset 330h) ...... 579
Figure 16-15. APIC Error Local Vector Table Register (APIC Offset 370h) ......... 580
Figure 16-16. APIC Error Status Register (APIC Offset 280h) .............................. 580
Figure 16-17. Spurious Interrupt Register (APIC Offset F0h) .................................................. 581
Figure 16-18. Interrupt Command Register (APIC Offset 300h–3010h) ................................. 582
Figure 16-19. Remote Read Register (APIC Offset C0h) ...................................................... 584
Figure 16-20. Logical Destination Register (APIC Offset D0h) ............................................. 585
Figure 16-21. Destination Format Register (APIC Offset E0h) .............................................. 586
Figure 16-22. Arbitration Priority Register (APIC Offset 90h) ............................................. 587
Figure 16-23. Interrupt Request Register (APIC Offset 200h–270h) ...................................... 588
Figure 16-24. In Service Register (APIC Offset 100h–170h) .................................................. 589
Figure 16-25. Trigger Mode Register (APIC Offset 180h–1F0h) ........................................... 590
Figure 16-26. Task Priority Register (APIC Offset 80h) ......................................................... 591
Figure 16-27. Processor Priority Register (APIC Offset A0h) ................................................ 591
Figure 16-28. End of Interrupt (APIC Offset B0h) ................................................................. 592
Figure 16-29. Specific End of Interrupt (APIC Offset 420h) ................................................... 593
Figure 16-30. Interrupt Enable Register (APIC Offset 480h–4F0h) ..................................... 593
Figure 17-1. P-State Current Limit Register (MSR C001_0061h) ............................................. 596
Figure 17-2. P-State Control Register (MSR C001_0062h) ..................................................... 596
Figure 17-3. P-State Status Register (MSR C001_0063h) ....................................................... 597
Figure 17-4. Core Performance Boost (MSRC001_0015h) .................................................. 598
Figure 17-5. Actual Performance Frequency Count (MSR0000_00E8h) ............................... 599
Figure 17-6. Max Performance Frequency Count (MSR0000_00E7h) ............................... 599
Figure 17-7. MPERF Read Only (MSR C000_00E7h) ............................................................. 600
### Tables

| Table 1-1. | Operating Modes | 11 |
| Table 1-2. | Interrupts and Exceptions | 20 |
| Table 2-1. | Instructions That Reference RSP | 31 |
| Table 2-2. | 64-Bit Mode Near Branches, Default 64-Bit Operand Size | 32 |
| Table 2-3. | Invalid Instructions in 64-Bit Mode | 34 |
| Table 2-4. | Invalid Instructions in Long Mode | 35 |
| Table 2-5. | Opcodes Reassigned in 64-Bit Mode | 36 |
| Table 2-6. | Differences Between Long Mode and Legacy Mode | 39 |
| Table 4-1. | Segment Registers | 73 |
| Table 4-2. | Descriptor Types | 83 |
| Table 4-3. | Code-Segment Descriptor Types | 85 |
| Table 4-4. | Data-Segment Descriptor Types | 86 |
| Table 4-5. | System-Segment Descriptor Types (S=0)—Legacy Mode | 87 |
| Table 4-6. | System-Segment Descriptor Types—Long Mode | 92 |
| Table 4-7. | Descriptor-Entry Field Changes in Long Mode | 96 |
| Table 4-8. | Segment Limit Checks in 64-Bit Mode | 116 |
| Table 5-1. | Supported Paging Alternatives (CR0.PG=1) | 122 |
| Table 5-2. | Physical-Page Protection, CR0.WP=0 | 149 |
| Table 5-3. | Effect of CR0.WP=1 on Supervisor Page Access | 150 |
| Table 6-1. | System Management Instructions | 151 |
| Table 7-1. | Memory Access by Memory Type | 176 |
| Table 7-2. | Caching Policy by Memory Type | 176 |
| Table 7-3. | Memory Access Ordering Rules | 178 |
| Table 7-4. | AMD64 Architecture Cache-Operating Modes | 185 |
| Table 7-5. | MTRR Type Field Encodings | 191 |
| Table 7-6. | Fixed-Range MTRR Address Ranges | 193 |
| Table 7-7. | Combined MTRR and Page-Level Memory Type with Unmodified PAT MSR | 199 |
| Table 7-8. | PAT Type Encodings | 201 |
| Table 7-9. | PAT-Register PA-Field Indexing | 202 |
| Table 7-10. | Combined Effect of MTRR and PAT Memory Types | 203 |
| Table 7-11. | Serialization Requirements for Changing Memory Types | 204 |
| Table 7-12. | Extended Fixed-Range MTRR Type Encodings | 206 |
| Table 8-1. | Interrupt Vector Source and Cause | 219 |
| Table 8-2. | Interrupt Vector Classification | 220 |
Table 8-3. Double-Fault Exception Conditions ........................................ 225
Table 8-4. Invalid-TSS Exception Conditions ........................................ 226
Table 8-5. Stack Exception Error Codes ................................................ 228
Table 8-6. General-Protection Exception Conditions ............................... 228
Table 8-7. Data-Type Alignment ......................................................... 231
Table 8-8. Simultaneous Interrupt Priorities ......................................... 236
Table 8-9. Simultaneous Floating-Point Exception Priorities .................... 238
Table 8-10. Virtual-8086 Mode Interrupt Mechanisms ............................ 249
Table 8-11. Effect of Instructions that Modify the IF Bit ......................... 262
Table 9-1. CPU Watchdog Timer Time Base .......................................... 273
Table 9-2. CPU Watchdog Timer Count Select ..................................... 273
Table 9-3. Error Logging Priorities ..................................................... 274
Table 9-4. Error Scope ........................................................................ 283
Table 10-1. AMD64 Architecture SMM State-Save Area ....................... 290
Table 10-2. Legacy SMM State-Save Area (Not used by AMD64 Architecture) ........................................ 293
Table 10-3. SMM Register Initialization ............................................... 297
Table 11-1. SSE Subsets – CPUID Feature Identifiers ............................ 306
Table 11-2. Extended Save Area Format .............................................. 320
Table 11-3. XRSTOR Hardware-Specified Initial Values ....................... 323
Table 11-4. Deriving FSAVE Tag Field from FXSAVE Tag Field ............. 329
Table 12-1. Effects of Task Nesting ..................................................... 349
Table 13-1. Breakpoint-Setting Examples ............................................ 360
Table 13-2. Breakpoint Location by Condition ..................................... 361
Table 13-3. Host/Guest Only Bits ....................................................... 369
Table 13-4. Count Control Using CNT_MASK and INV ....................... 370
Table 13-5. Operating-System Mode and User Mode Bits ..................... 370
Table 13-6. EventId Values ............................................................ 393
Table 13-7. Lightweight Profiling CPUID Values ................................. 404
Table 13-8. LWPCB—Lightweight Profiling Control Block Fields .......... 415
Table 13-9. LWPCB Filters Fields ....................................................... 420
Table 13-10. XSAVE Area for LWP Fields ........................................... 424
Table 14-1. Initial Processor State ....................................................... 432
Table 14-2. Initial State of Segment-Register Attributes ....................... 434
Table 14-3. x87 Floating-Point State Initialization ................................. 436
Table 14-4. Processor Operating Modes .............................................. 441
Table 14-5. Long-Mode Consistency Checks ....................................... 442
Table 15-1. Guest Exception or Interrupt Types ........................................ 460
Table 15-2. EXITINFO1 for MOV CRx ................................................ 462
Table 15-3. EXITINFO1 for MOV DRx ............................................. 462
Table 15-4. EXITINFO1 for INTn .................................................. 463
Table 15-5. EXITINFO1 for INVLP .................}get ...463
Table 15-6. Guest Instruction Bytes ............................................. 463
Table 15-7. Instruction Intercepts ............................................... 464
Table 15-8. MSR Ranges Covered by MSRPM ................................ 468
Table 15-9. TLB Control Byte Encodings ...................................... 479
Table 15-10. Effect of the GIF on Interrupt Handling ................. 480
Table 15-11. Guest Exception or Interrupt Types ......................... 482
Table 15-12. INIT Handling in Different Operating Modes ............ 485
Table 15-13. NMI Handling in Different Operating Modes .......... 486
Table 15-14. SMI Handling in Different Operating Modes .......... 487
Table 15-15. DEV Capability Block, Overall Layout .................. 492
Table 15-16. DEV Capability Header (DEV_HDR) (in PCI Config Space) ....... 492
Table 15-17. Encoding of Function Field in DEV_OP Register ...... 493
Table 15-18. DEV_CR Control Register ........................................ 494
Table 15-19. Combining Guest and Host PAT Types .................... 502
Table 15-20. Combining PAT and MTRR Types .......................... 502
Table 15-21. GMET Page Configuration ...................................... 503
Table 15-22. Guest vAPIC Register Access Behavior ................... 513
Table 15-23. Virtual Interrupt Control (VMCB offset 60h) ............. 517
Table 15-24. New VMCB Fields Defined by AVIC ......................... 517
Table 15-25. Physical APIC ID Table Entry Fields ....................... 520
Table 15-26. Logical APIC ID Table Entry Fields ......................... 522
Table 15-27. EXTINFO1 Fields ............................................. 528
Table 15-28. EXTINFO2 Fields ............................................. 528
Table 15-29. ID Field—IPI Delivery Failure Cause ....................... 529
Table 15-30. EXTINFO1 Fields ............................................. 529
Table 15-31. EXTINFO2 Fields ............................................. 530
Table 15-32. Encryption Control ........................................... 540
Table 15-33. SEV/SME Interaction ........................................... 541
Table 15-34. SEV_STATUS MSR Fields .................................... 542
Table 15-35. AE Exitcodes ................................................ 544
Table 15-36. Fields of a RMP Entry ........................................ 550
Table 15-37. RMP Page Assignment Settings .......................................................... 552
Table 15-38. VMPL Permission Mask Definition ..................................................... 554
Table 15-39. RMP Memory Access Checks .............................................................. 556
Table 15-40. PVALIDATE/RMPADJUST Page Size Mismatch Combinations .......... 558
Table 15-41. VMRUN Page Checks ........................................................................ 559
Table 15-42. Non-Coherent Memory Type Conversion ............................................ 560
Table 16-1. Interrupt Sources for Local APIC .......................................................... 564
Table 16-2. APIC Registers .................................................................................... 567
Table 16-3. Divide Values ....................................................................................... 574
Table 16-4. Valid ICR Field Combinations .............................................................. 580
Table A-1. MSRs of the AMD64 Architecture ......................................................... 599
Table A-2. System-Software MSR Cross-Reference ............................................... 604
Table A-3. Memory-Typing MSR Cross-Reference .................................................. 605
Table A-4. Machine-Check MSR Cross-Reference ................................................... 607
Table A-5. Software-Debug MSR Cross-Reference ................................................... 608
Table A-6. Performance-Monitoring MSR Cross-Reference ...................................... 609
Table A-7. Secure Virtual Machine MSR Cross-Reference ....................................... 611
Table A-8. System Management Mode MSR Cross-Reference .................................. 612
Table A-9. CPUID Namestring MSR Cross Reference ............................................. 612
Table B-1. VMCB Layout, Control Area ................................................................. 613
Table B-2. VMCB Layout, State Save Area .............................................................. 618
Table B-3. Swap Types .......................................................................................... 621
Table B-4. VMSA Layout, State Save Area for SEV-ES .......................................... 621
Table C-1. SVM Intercept Codes ........................................................................... 627
# Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Revision</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>April 2020</td>
<td>3.34</td>
<td>Section 3.1.3: Updated register information. Added PCIDE and PKE registers. Updated (TCE) content. Section 5.3.2: Added Process Context Identifier register information and register figure. Section 5.3.3: Updated figure. Section 5.3.4: Updated figure. Section 5.3.5: Updated figure. Section 5.4.1: Added (MPK) register information. Section 5.5.1: Inserted Process Context Identifiers as Section 5.5.1. Section 5.5.3: Added bullets to Implicit Invalidations list. Section 5.6: Updated content. Section 8.2.15: Added bullet. Section 8.4.2: Updated register figure and added PK register information. Section 11.5.2: Updated register figure and table. Section 14.1.3: Updated table. Section 15.9: Updated table. Appendix B: Updated table. Appendix C: Updated table.</td>
</tr>
<tr>
<td>Date</td>
<td>Revision</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>April 2020</td>
<td>3.33</td>
<td>Section 1.1.2: Clarification on address size support.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Section 3.2.1: New feature enable bits in SYSCFG MSR.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Section 7.6.5: Updated terminology.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Section 7.10.6: Clarification to encrypted memory operation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Section 8.1.4: Clarification to IRET and NMI behavior.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tables 8-1 and 8-2: Added #HV exception.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inserted new 8.2.20 section for #HV exception.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Section 8.4.2: Changes for SEV-SNP extension.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Table 8-8: Added SEV-related exceptions.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Figure 10-6: Updated I/O Restart DWORD.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Section 15 and 15.1: General updates.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Section 15.5.2: Relocated VMLOAD/VMSAVE documentation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Section 15.2.4 and 15.2.6: Updated content.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Section 15.6: Added content.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Table 15-7: Added content.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Section 15.25.6: Clarification.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Section 15.25.13: General clarifications.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Section 15.33.1 and 15.33.2: General clarifications.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Section 15.34.3, 15.34.7, and 15.34.10: Clarifications, and additions for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SEV-SNP.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Table 15-35: Added content.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Section 15.35.8: Corrected terminology.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Section 15.36: Added SEV-SNP extension documentation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Table A-1 and Table A-7: Added SEV-SNP related MSRs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Appendix B: Updates for SEV-SNP extension.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Table C-1: Added exit code for SEV-SNP extension.</td>
</tr>
<tr>
<td>October 2019</td>
<td>3.32</td>
<td>Added UMIP, XSS, GMET, VTE, MCOMMIT, and RDPRU.</td>
</tr>
<tr>
<td>July 2019</td>
<td>3.31</td>
<td>Added CLWB and WBNOINVD details.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clarified FP error pointer save/restore behavior.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Corrected description of APIC Software Enable functionality.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clarified canonical address checking behavior.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clarified fault generation. April 2020 for instructions that cross page or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>segment boundaries.</td>
</tr>
<tr>
<td>Date</td>
<td>Revision</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>September 2018</td>
<td>3.30</td>
<td>Modified Section 7.4, Modified Section 7.6.4, Modified Section 8.5.2,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Modified Section 9.2, Corrected Figure 9-4, Corrected Table 9-1, Modified</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Section 9.3.2, Corrected Figure 9-6, Corrected Table 9-4, Modified Section</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14.2.3, Modified Section 14.4, Modified Section 15.6, Modified Section</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.7, Modified Section 15.34.9, Modified Section 15.34.10, Modified Section</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15.35.2, Corrected Table B-4 in Appendix B</td>
</tr>
<tr>
<td>December 2017</td>
<td>3.29</td>
<td>Modified Sections 7.10.1 and 7.10.4, Modified Sections 15.34.1, 15.34.7,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added new Section 15.34.10, Modified Section 15.35.10, Modified Appendix A,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Table A-7.</td>
</tr>
<tr>
<td>March 2017</td>
<td>3.28</td>
<td>Modified CR4 Register, Section 3.1.3, Removed UD2 in Table 6-1, Added new</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bullet in Section 7.1.1, Modified Note in Table 7-1, Added new Section 7.4.1,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clarified Self Modifying Code in Section 7.6.1, Added UD0 and UD1 instructions in Section 8.2.7, Added Instructions Retired Performance counter in Section 13.1.1, Modified Table in Section 15.34.9.</td>
</tr>
<tr>
<td>Date</td>
<td>Revision</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>December 2016</td>
<td>3.27</td>
<td>Added Resume Flag (RF) Bit in Section 3.1.6, &quot;RFLAGS Register,&quot; on page 51.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added Tom2ForceMemTypeWB in Section 3.2.1, &quot;System Configuration Register (SYSCFG),&quot; on page 59.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clarified SYSCALL and SYSRET in Section 6.1.1, &quot;SYSCALL and SYSRET,&quot; on page 159.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added Section 7.3.2, &quot;Access Atomicity,&quot; on page 178.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updated Note b in Table 7-11 on page 208.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Modified Table 8-1, &quot;Interrupt Vector Source and Cause&quot;, on page 223.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Modified Table 8-2, &quot;Interrupt Vector Classification&quot;, on page 224.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added Section 8.2.21, &quot;#VC—VMM Communication Exception (Vector 29),&quot; on page 238.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added Section 10.5, &quot;Multiprocessor Considerations,&quot; on page 307.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Updated CPUID 8000_001F[EAX] and added CPUID 8000_001F[EDX] in Section 15.34.1, &quot;Determining Support for SEV,&quot; on page 540.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added new Section 15.35, &quot;Encrypted State (SEV-ES),&quot; on page 546.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clarified TSC Ratio MSR in Section 15.30.5 &quot;TSC Ratio MSR (C000_0104h)&quot; on page 536.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Modified Appendix B, &quot;Layout of VMCB&quot; on page 617.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added Table B-3, &quot;Swap Types&quot;, on page 625.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added Codes 8Fh, 90h-9Fh, and 403h in Table C-1, &quot;SVM Intercept Codes&quot;, on page 631.</td>
</tr>
<tr>
<td>April 2016</td>
<td>3.26</td>
<td>Clarification on loading a null selector into FS or GS added in Section 4.5.3, &quot;Segment Registers in 64-Bit Mode,&quot; on page 74</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Translation table diagrams corrected for definition of bit 8 in Section 5.5, &quot;Translation-Lookaside Buffer (TLB),&quot; on page 145</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CR0.CD implementation-dependent behavior noted in Section 7.6.2, &quot;Cache Control Mechanisms,&quot; on page 188</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added clarification on IST usage in Section 8.9.4, &quot;Interrupt-Stack Table,&quot; on page 260.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added guideline for secure AP startup in Section 15.27.8, &quot;Secure Multiprocessor Initialization,&quot; on page 512</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added TLB maintenance requirement for multiprocessor VM's in Section 15.29.2.2, &quot;VMCB Changes in Support of AVIC,&quot; on page 520.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added new Section 15.34, &quot;Secure Encrypted Virtualization,&quot; on page 539.</td>
</tr>
<tr>
<td>Date</td>
<td>Revision</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>June 2015</td>
<td>3.25</td>
<td>Added new section 15.33 Nested Virtualization for coverage of VMSAVE and VMLOAD Virtualization and Virtual GIF. Various minor edits.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Indicated the deprecation of the Processor Feedback Interface. See Section 17.4, “Processor Feedback Interface,” on page 600.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added Section 17.5, “Processor Core Power Reporting,” on page 600.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added AVIC description. See Section 15.29, “Advanced Virtual Interrupt Controller,” on page 514.</td>
</tr>
<tr>
<td>September 2012</td>
<td>3.22</td>
<td>Clarified processor behavior on write of EFER[LMA] bit in Section 3.1.7 “Extended Feature Enable Register (EFER)” on page 55.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clarified difference between cold reset and warm reset in Section 9.3, “Machine Check Architecture MSRs,” on page 273.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added information on FFXSR feature bit to Table 11-1 on page 310.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clarified SMM code responsibility to manage VMCB clean bits. See Section 15.15.2, “Guidelines for Clearing VMCB Clean Bits,” on page 480.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added a note to Table 15-9 on page 483 to indicate that all encodings of TLB_CONTROL not defined are reserved.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Corrected information concerning the assignment of logical APIC IDs in Section 16.6.1, “Receiving System and IPI Interrupts,” on page 585.</td>
</tr>
<tr>
<td>March 2012</td>
<td>3.21</td>
<td>Added definition of processor feedback interface—frequency sensitivity monitor (See Section 17.4, “Processor Feedback Interface,” on page 600)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added Instruction-Based Sampling in a new section of Chapter 13 (See Section 13.3, “Instruction-Based Sampling,” on page 379.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Reworked Introduction and first section of Chapter 9, “Machine Check Architecture,” on page 269 and added deferred error handling.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added references to the RDFSBASE, RDGSBASE, WRFSBASE, and WRGBASE instructions in discussion of FS and GS segment descriptors.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(See &quot;FS and GS Registers in 64-Bit Mode&quot; on page 74)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Added Section 6.3.2, “Accessing Segment Register Hidden State,” on page 164.</td>
</tr>
<tr>
<td>Date</td>
<td>Revision</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>December 2011</td>
<td>3.20</td>
<td>Clarified description of the Cache Disable (CD) memory type in Section 7.4 “Memory Types” on page 178. Added caveat: an overflow of either APERF or MPERF can invalidate the effective frequency calculation. See Section 17.3, “Determining Processor Effective Frequency,” on page 598. Other minor editorial changes.</td>
</tr>
<tr>
<td>September 2011</td>
<td>3.19</td>
<td>Added XSAVEOPT to discussions on XSAVE. Corrections to discussion on multiprocessor memory access ordering in Chapter 7. Added discussion of extended core and northbridge performance counters and feature indicators to Chapter 13. Added Lightweight Profiling (LWP) to Chapter 13. Added Global Timestamp Counter, Continuous Mode to LWP description Clarification: Function of pin A20M# is only defined in real mode. Statement added to Section 1.2.4, “Real Addressing,” on page 10. Eliminated hardware P-state references</td>
</tr>
<tr>
<td>May 2011</td>
<td>3.18</td>
<td>Added information for OSXSAVE and XSAVE features. Added Cache Topology, Pause Filter Threshold, and XSETBV information. Updated TSC ratio information. Corrected description of FXSAVE/FXRSTOR exception behavior when CR0.EM=1</td>
</tr>
<tr>
<td>Date</td>
<td>Revision</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>November 2009</td>
<td>3.15</td>
<td>Added section 7.5, &quot;Buffering and Combining Memory Writes&quot; on page 183&lt;br&gt;Added MFENCE to list of &quot;Serializing Instructions&quot; on page 192.&lt;br&gt;Updated section 7.6.1, &quot;Cache Organization and Operation&quot; on page 185.&lt;br&gt;Updated Table 7-3, &quot;Memory Access Ordering Rules&quot;, on page 182 and notes.&lt;br&gt;Updated 7.4, &quot;Memory Types&quot; on page 178.&lt;br&gt;Clarified 5.5.3, &quot;TLB Management&quot; on page 146.&lt;br&gt;Added &quot;Invalidation of Table Entry Upgrades.&quot; on page 147.&lt;br&gt;Updated &quot;Speculative Caching of Address Translations&quot; on page 147.&lt;br&gt;Update &quot;Handling of D-Bit Updates&quot; on page 148.&lt;br&gt;Revised and updated section 7.2, &quot;Multiprocessor Memory Access Ordering&quot; on page 172 ff.&lt;br&gt;Added information on long mode segment-limit checks in &quot;Extended Feature Enable Register (EFER)&quot; on page 56 table on page 56 and &quot;Long Mode Segment Limit Enable (LMSLE) bit&quot; on page 57.&lt;br&gt;Added discussion of &quot;Data Limit Checks in 64-bit Mode&quot; on page 116 on page 116.&lt;br&gt;Updated Table 6-1, &quot;System Management Instructions&quot;, on page 155.&lt;br&gt;Updated &quot;Canonicalization and Consistency Checks&quot; on page 459 on page 459.&lt;br&gt;Added information about the next sequential instruction pointer (nRIP) in 15.7.1, &quot;State Saved on Exit&quot; on page 463.&lt;br&gt;Updated priority definition of PAUSE instruction intercept in Table 15-7, &quot;Instruction Intercepts&quot;, on page 468.&lt;br&gt;Added nRIP field to Table B-1, &quot;VMCB Layout, Control Area&quot;, on page 617.&lt;br&gt;Clarified information on ICEBP event injection, on page 486.&lt;br&gt;Deleted erroneous statement concerning the operation of the General Local Vector Table register Mask bit in section 16.4.&lt;br&gt;Clarified the description of the Interrupt Command Register Delivery Status bit in section &quot;Interprocessor Interrupts (IPI)&quot; on page 581 on page 581.</td>
</tr>
<tr>
<td>Date</td>
<td>Revision</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>September 2006</td>
<td>3.12</td>
<td>Added numerous minor clarifications.</td>
</tr>
<tr>
<td>February 2005</td>
<td>3.10</td>
<td>Corrected Table 8-6, &quot;General-Protection Exception Conditions&quot;, on page 233. Added SSE3 information. Clarified and corrected information on the CPUID instruction and feature identification. Added information on the RDTSCP instruction. Clarified information about MTRRs and PATs in multiprocessing systems.</td>
</tr>
<tr>
<td>Date</td>
<td>Revision</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>September 2003</td>
<td>3.09</td>
<td>Corrected numerous minor typographical errors.</td>
</tr>
<tr>
<td>April 2003</td>
<td>3.08</td>
<td>Clarified terms in section on FXSAVE/FXSTOR. Corrected several minor errors of omission. Documentation of CR0.NW bit has been corrected. Several register diagrams and figure labels have been corrected. Description of shared cache lines has been clarified in 7.3, &quot;Memory Coherency and Protocol&quot; on page 175.</td>
</tr>
<tr>
<td>September 2002</td>
<td>3.07</td>
<td>Made numerous small grammatical changes and factual clarifications. Added Revision History.</td>
</tr>
</tbody>
</table>
Preface

About This Book

This book is part of a multivolume work entitled the *AMD64 Architecture Programmer’s Manual*. This table lists each volume and its order number.

<table>
<thead>
<tr>
<th>Title</th>
<th>Order No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volume 1: Application Programming</td>
<td>24592</td>
</tr>
<tr>
<td>Volume 2: System Programming</td>
<td>24593</td>
</tr>
<tr>
<td>Volume 3: General-Purpose and System Instructions</td>
<td>24594</td>
</tr>
<tr>
<td>Volume 4: 128-Bit and 256-Bit Media Instructions</td>
<td>26568</td>
</tr>
<tr>
<td>Volume 5: 64-Bit Media and x87 Floating-Point Instructions</td>
<td>26569</td>
</tr>
</tbody>
</table>

Audience

This volume (Volume 2) is intended for programmers writing operating systems, loaders, linkers, device drivers, or system utilities. It assumes an understanding of AMD64 architecture application-level programming as described in Volume 1.

This volume describes the AMD64 architecture’s resources and functions that are managed by system software, including operating-mode control, memory management, interrupts and exceptions, task and state-change management, system-management mode (including power management), multi-processor support, debugging, and processor initialization.

Application-programming topics are described in Volume 1. Details about each instruction are described in Volumes 3, 4, and 5.

Organization

This volume begins with an overview of system programming and differences between the x86 and AMD64 architectures. This is followed by chapters that describe the following details of system programming:

- *System Resources*—The system registers and processor ID (CPUID) functions.
- *Segmented Virtual Memory*—The segmented-memory models supported by the architecture and their associated data structures and protection checks.
- *Page Translation and Protection*—The page-translation functions supported by the architecture and their associated data structures and protection checks.
• System Instructions—The instructions used to manage system functions.
• Memory System—The memory-system hierarchy and its resources and protocols, including memory-characterization, caching, and buffering functions.
• Exceptions and Interrupts—Details about the types and causes of exceptions and interrupts, and the methods of transferring control during these events.
• Machine-Check Mechanism—The resources and functions that support detection and handling of machine-check errors.
• System-Management Mode—The resources and functions that support system-management mode (SMM), including power-management functions.
• SSE, MMX, and x87 Programming—The resources and functions that support use (by application software) and state-saving (by the operation system) of the 256-bit media, 128-bit media, 64-bit media, and x87 floating-point instructions.
• Multiple-Processor Management—The features of the instruction set and the system resources and functions that support multiprocessing environments.
• Debug and Performance Resources—The system resources and functions that support software debugging and performance monitoring.
• Legacy Task Management—Support for the legacy hardware multitasking functions, including register resources and data structures.
• Processor Initialization and Long-Mode Activation—The methods by which system software initializes and changes operating modes.
• Mixing Code Across Operating Modes—Things to remember when running programs in different operating modes.
• Secure Virtual Machine—The system resources that support machine virtualization.
• Advanced Programmable Interrupt Controller (APIC) operation.

There are appendices describing details of model-specific registers (MSRs) and machine-check implementations. Definitions assumed throughout this volume are listed below. The index at the end of this volume cross-references topics within the volume. For other topics relating to the AMD64 architecture, see the tables of contents and indexes of the other volumes.

Conventions and Definitions

The section which follows, Notational Conventions, describes notational conventions used in this volume. The next section, Definitions, lists a number of terms used in this volume along with their technical definitions. Some of these definitions assume knowledge of the legacy x86 architecture. See “Related Documents” on page liii for further information about the legacy x86 architecture. Finally, the Registers section lists the registers which are a part of the system programming model.
Notational Conventions

#GP(0)
An instruction exception—in this example, a general-protection exception with error code of 0.

1011b
A binary value—in this example, a 4-bit value.

F0EA_0B02h
A hexadecimal value. Underscore characters may be inserted to improve readability.

128
Numbers without an alpha suffix are decimal unless the context indicates otherwise.

7:4
A bit range, from bit 7 to 4, inclusive. The high-order bit is shown first. Commas may be inserted to indicate gaps.

CPUID FnXXX_XXX_RRR[FieldName]
Support for optional features or the value of an implementation-specific parameter of a processor can be discovered by executing the CPUID instruction on that processor. To obtain this value, software must execute the CPUID instruction with the function code XXXX_XXXh in EAX and then examine the field FieldName returned in register RRR. If the “_RRR” notation is followed by “_xYYY”, register ECX must be set to the value YYYh before executing CPUID. When FieldName is not given, the entire contents of register RRR contains the desired value. When determining optional feature support, if the bit identified by FieldName is set to a one, the feature is supported on that processor.

CR0–CR4
A register range, from register CR0 through CR4, inclusive, with the low-order register first.

CR0[PE], CR0.PE
Notation for referring to a field within a register—in this case, the PE field of the CR0 register.

CR0[PE] = 1, CR0.PE = 1
The PE field of the CR0 register is set (contains the value 1).

EFER[LME] = 0, EFER.LME = 0
The LME field of the EFER register is cleared (contains a value of 0).

DS:SI
A far pointer or logical address. The real address or segment descriptor specified by the segment register (DS in this example) is combined with the offset contained in the second register (SI in this example) to form a real or virtual address.
RFLAGS[13:12]

A field within a register identified by its bit range. In this example, corresponding to the IOPL field.

Definitions

16-bit mode

Legacy mode or compatibility mode in which a 16-bit address size is active. See legacy mode and compatibility mode.

32-bit mode

Legacy mode or compatibility mode in which a 32-bit address size is active. See legacy mode and compatibility mode.

64-bit mode

A submode of long mode. In 64-bit mode, the default address size is 64 bits and new features, such as register extensions, are supported for system and application software.

absolute

Said of a displacement that references the base of a code segment rather than an instruction pointer. Contrast with relative.

ASID

Address space identifier.

byte

Eight bits.

clear

To write a bit value of 0. Compare set.

compatibility mode

A submode of long mode. In compatibility mode, the default address size is 32 bits, and legacy 16-bit and 32-bit applications run without modification.

commit

To irreversibly write, in program order, an instruction’s result to software-visible storage, such as a register (including flags), the data cache, an internal write buffer, or memory.

CPL

Current privilege level.

direct

Referencing a memory location whose address is included in the instruction’s syntax as an immediate operand. The address may be an absolute or relative address. Compare indirect.
dirty data
Data held in the processor’s caches or internal buffers that is more recent than the copy held in main memory.

displacement
A signed value that is added to the base of a segment (absolute addressing) or an instruction pointer (relative addressing). Same as offset.

doubleword
Two words, or four bytes, or 32 bits.

double quadword
Eight words, or 16 bytes, or 128 bits. Also called octword.
effective address size
The address size for the current instruction after accounting for the default address size and any address-size override prefix.
effective operand size
The operand size for the current instruction after accounting for the default operand size and any operand-size override prefix.
exception
An abnormal condition that occurs as the result of executing an instruction. The processor’s response to an exception depends on the type of the exception. For all exceptions except 128-bit media SIMD floating-point exceptions and x87 floating-point exceptions, control is transferred to the handler (or service routine) for that exception, as defined by the exception’s vector. For floating-point exceptions defined by the IEEE 754 standard, there are both masked and unmasked responses. When unmasked, the exception handler is called, and when masked, a default response is provided instead of calling the handler.
flush
An often ambiguous term meaning (1) writeback, if modified, and invalidate, as in “flush the cache line,” or (2) invalidate, as in “flush the pipeline,” or (3) change a value, as in “flush to zero.”

GDT
Global descriptor table.

GIF
Global interrupt flag.

GPA
Guest physical address. In a virtualized environment, the page tables maintained by the guest operating system provide the translation from the linear (virtual) address to the guest physical
address. Nested page tables define the translation of the GPA to the host physical address (HPA). See SPA and HPA.

HPA
Host physical address. The address space owned by the virtual machine monitor. In a virtualized environment, nested page translation tables controlled by the VMM provide the translation from the guest physical address to the host physical address. See GPA.

IDT
Interrupt descriptor table.

IGN
Ignored. Value written is ignored by hardware. Value returned on a read is indeterminate. See reserved.

indirect
Referencing a memory location whose address is in a register or other memory location. The address may be an absolute or relative address. Compare direct.

IRB
The virtual-8086 mode interrupt-redirection bitmap.

IST
The long-mode interrupt-stack table.

IVT
The real-address mode interrupt vector table.

LDT
Local descriptor table.

legacy x86
The legacy x86 architecture. See “Related Documents” on page liii for descriptions of the legacy x86 architecture.

legacy mode
An operating mode of the AMD64 architecture in which existing 16-bit and 32-bit applications and operating systems run without modification. A processor implementation of the AMD64 architecture can run in either long mode or legacy mode. Legacy mode has three submodes, real mode, protected mode, and virtual-8086 mode.

long mode
An operating mode unique to the AMD64 architecture. A processor implementation of the AMD64 architecture can run in either long mode or legacy mode. Long mode has two submodes, 64-bit mode and compatibility mode.
lsb
    Least-significant bit.

LSB
    Least-significant byte.

main memory
    Physical memory, such as RAM and ROM (but not cache memory) that is installed in a particular computer system.

mask
    (1) A control bit that prevents the occurrence of a floating-point exception from invoking an exception-handling routine. (2) A field of bits used for a control purpose.

MBZ
    Must be zero. If software attempts to set an MBZ bit to 1 in a system register, a general-protection exception (#GP) occurs; if in a translation table entry, a reserved-bit page fault exception (#PF) will occur if the hardware attempts to use the entry for address translation. See reserved.

memory
    Unless otherwise specified, main memory.

ModRM
    A byte following an instruction opcode that specifies address calculation based on mode (Mod), register (R), and memory (M) variables.

moffset
    A 16, 32, or 64-bit offset that specifies a memory operand directly, without using a ModRM or SIB byte.

msb
    Most-significant bit.

MSB
    Most-significant byte.

octword
    Same as double quadword.

offset
    Same as displacement.

overflow
    The condition in which a floating-point number is larger in magnitude than the largest, finite, positive or negative number that can be represented in the data-type format being used.
PAE
   Physical-address extensions.

physical memory
   Actual memory, consisting of main memory and cache.

probe
   A check for an address in a processor’s caches or internal buffers. External probes originate outside the processor, and internal probes originate within the processor.

protected mode
   A submode of legacy mode.

quadword
   Four words, or eight bytes, or 64 bits.

RAZ
   Value returned on a read is always zero (0) regardless of what was previously written. See reserved.

real-address mode
   See real mode.

real mode
   A short name for real-address mode, a submode of legacy mode.

relative
   Referencing with a displacement (also called offset) from an instruction pointer rather than the base of a code segment. Contrast with absolute.

reserved
   Fields marked as reserved may be used at some future time.
   To preserve compatibility with future processors, reserved fields require special handling when read or written by software. Software must not depend on the state of a reserved field (unless qualified as RAZ), nor upon the ability of such fields to return a previously written state.
   If a field is marked reserved without qualification, software must not change the state of that field; it must reload that field with the same value returned from a prior read.
   Reserved fields may be qualified as IGN, MBZ, RAZ, or SBZ (see definitions).

REX
   An instruction prefix that specifies a 64-bit operand size and provides access to additional registers.

RIP-relative addressing
   Addressing relative to the 64-bit RIP instruction pointer.
SBZ

Should be zero. An attempt by software to set an SBZ bit to 1 results in undefined behavior. See reserved.

set

To write a bit value of 1. Compare clear.

SIB

A byte following an instruction opcode that specifies address calculation based on scale (S), index (I), and base (B).

SPA

System physical address. The address directly used to address system memory. Under SVM, also known as the host physical address. See HPA.

sticky bit

A bit that is set or cleared by hardware and that remains in that state until explicitly changed by software.

SVM

Secure virtual machine. AMD’s virtualization architecture. SVM is defined in Chapter 15 on page 453.

System software

Privileged software that owns and manages the hardware resources of a system after initialization by system firmware and controls access to these resources. In a non-virtualized environment, system software is provided by the operating system. In a virtualized environment, system software is largely equivalent to the virtual machine monitor (VMM), also commonly known as the hypervisor.

TOP

The x87 top-of-stack pointer.

TSS

Task-state segment.

underflow

The condition in which a floating-point number is smaller in magnitude than the smallest nonzero, positive or negative number that can be represented in the data-type format being used.

vector

(1) A set of integer or floating-point values, called elements, that are packed into a single data object. Most of the SSE and 64-bit media instructions use vectors as operands.

(2) An index into an interrupt descriptor table (IDT), used to access exception handlers. Compare exception.
virtual-8086 mode
A submode of *legacy mode*.

VMCB
Virtual machine control block.

VMM
Virtual machine monitor.

word
Two bytes, or 16 bits.

x86
See *legacy x86*.

**Registers**

In the following list of registers, the names are used to refer either to a given register or to the contents of that register:

**AH–DH**
The high 8-bit AH, BH, CH, and DH registers. Compare *AL–DL*.

**AL–DL**
The low 8-bit AL, BL, CL, and DL registers. Compare *AH–DH*.

**AL–r15B**
The low 8-bit AL, BL, CL, DL, SIL, DIL, BPL, SPL, and R8B–R15B registers, available in 64-bit mode.

**BP**
Base pointer register.

**CRn**
Control register number *n*.

**CS**
Code segment register.

**eAX–eSP**
The 16-bit AX, BX, CX, DX, DI, SI, BP, and SP registers or the 32-bit EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP registers. Compare *rAX–rSP*.

**EFER**
Extended features enable register.
eFLAGS
   16-bit or 32-bit flags register. Compare rFLAGS.

EFLAGS
   32-bit (extended) flags register.

eIP
   16-bit or 32-bit instruction-pointer register. Compare rIP.

EIP
   32-bit (extended) instruction-pointer register.

FLAGS
   16-bit flags register.

GDTR
   Global descriptor table register.

GPRs
   General-purpose registers. For the 16-bit data size, these are AX, BX, CX, DX, DI, SI, BP, and SP. For the 32-bit data size, these are EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP. For the 64-bit data size, these include RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, and R8–R15.

IDTR
   Interrupt descriptor table register.

IP
   16-bit instruction-pointer register.

LDTR
   Local descriptor table register.

MSR
   Model-specific register.

r8–r15
   The 8-bit R8B–R15B registers, or the 16-bit R8W–R15W registers, or the 32-bit R8D–R15D registers, or the 64-bit R8–R15 registers.

rAX–rSP
   The 16-bit AX, BX, CX, DX, DI, SI, BP, and SP registers, or the 32-bit EAX, EBX, ECX, EDX, EDI, ESI, EBP, and ESP registers, or the 64-bit RAX, RBX, RCX, RDX, RDI, RSI, RBP, and RSP registers. Replace the placeholder r with nothing for 16-bit size, “E” for 32-bit size, or “R” for 64-bit size.
RAX
   64-bit version of the EAX register.

RBP
   64-bit version of the EBP register.

RBX
   64-bit version of the EBX register.

RCX
   64-bit version of the ECX register.

RDI
   64-bit version of the EDI register.

RDX
   64-bit version of the EDX register.

rFLAGS
   16-bit, 32-bit, or 64-bit flags register. Compare RFLAGS.

RFLAGS
   64-bit flags register. Compare rFLAGS.

rIP
   16-bit, 32-bit, or 64-bit instruction-pointer register. Compare RIP.

RIP
   64-bit instruction-pointer register.

RSI
   64-bit version of the ESI register.

RSP
   64-bit version of the ESP register.

SP
   Stack pointer register.

SS
   Stack segment register.

TPR
   Task priority register (CR8), a new register introduced in the AMD64 architecture to speed interrupt management.
TR

Task register.

YMM/XMM

Set of sixteen (eight accessible in legacy and compatibility modes) 256-bit wide registers that hold scalar and vector operands used by the SSE instructions.

Endian Order

The x86 and AMD64 architectures address memory using little-endian byte-ordering. Multibyte values are stored with their least-significant byte at the lowest byte address, and they are illustrated with their least significant byte at the right side. Strings are illustrated in reverse order, because the addresses of their bytes increase from right to left.

Related Documents

- AMD, *BIOS and Kernel Developer’s Guide* (BKDG) for particular hardware implementations of older families of the AMD64 architecture.
- AMD, *Processor Programming Reference (PPR)* for particular hardware implementations of newer families of the AMD64 architecture.
- AMD, *AMD I/O Virtualization Technology (IOMMU) Specification*, Revision 2.2 or later; order number 48882.
• Cyrix Corporation, *M1 Processor Data Book*, Cyrix Corporation, Richardson, TX, 1996.
• Cyrix Corporation, *MX Processor MMX Extension Opcode Table*, Cyrix Corporation, Richardson, TX, 1996.
• Cyrix Corporation, *MX Processor Data Book*, Cyrix Corporation, Richardson, TX, 1997.
1 System-Programming Overview

This entire volume is intended for system-software developers—programmers writing operating systems, loaders, linkers, device drivers, or utilities that require access to system resources. These system resources are generally available only to software running at the highest-privilege level (CPL=0), also referred to as privileged software. Privilege levels and their interactions are fully described in “Segment-Protection Overview” on page 97.

This chapter introduces the basic features and capabilities of the AMD64 architecture that are available to system-software developers. The concepts include:

- The supported address forms and how memory is organized.
- How memory-management hardware makes use of the various address forms to access memory.
- The processor operating modes, and how the memory-management hardware supports each of those modes.
- The system-control registers used to manage system resources.
- The interrupt and exception mechanism, and how it is used to interrupt program execution and to report errors.
- Additional, miscellaneous features available to system software, including support for hardware multitasking, reporting machine-check exceptions, debugging software problems, and optimizing software performance.

Many of the legacy features and capabilities are enhanced by the AMD64 architecture to support 64-bit operating systems and applications, while providing backward-compatibility with existing software.

1.1 Memory Model

The AMD64 architecture memory model is designed to allow system software to manage application software and associated data in a secure fashion. The memory model is backward-compatible with the legacy memory model. Hardware-translation mechanisms are provided to map addresses between virtual-memory space and physical-memory space. The translation mechanisms allow system software to relocate applications and data transparently, either anywhere in physical-memory space, or in areas on the system hard drive managed by the operating system.

In long mode, the AMD64 architecture implements a flat-memory model. In legacy mode, the architecture implements all legacy memory models.
1.1.1 Memory Addressing

The AMD64 architecture supports address relocation. To do this, several types of addresses are needed to completely describe memory organization. Specifically, four types of addresses are defined by the AMD64 architecture:

- Logical addresses
- Effective addresses, or segment offsets, which are a portion of the logical address.
- Linear (virtual) addresses
- Physical addresses

**Logical Addresses.** A *logical address* is a reference into a segmented-address space. It is comprised of the segment selector and the effective address. Notationally, a logical address is represented as

\[
\text{Logical Address} = \text{Segment Selector} : \text{Offset}
\]

The segment selector specifies an entry in either the global or local descriptor table. The specified descriptor-table entry describes the segment location in virtual-address space, its size, and other characteristics. The effective address is used as an offset into the segment specified by the selector.

Logical addresses are often referred to as *far pointers*. Far pointers are used in software addressing when the segment reference must be explicit (i.e., a reference to a segment outside the current segment).

**Effective Addresses.** The offset into a memory segment is referred to as an effective address (see “Segmentation” on page 5 for a description of segmented memory). Effective addresses are formed by adding together elements comprising a base value, a scaled-index value, and a displacement value. The effective-address computation is represented by the equation

\[
\text{Effective Address} = \text{Base} + (\text{Scale} \times \text{Index}) + \text{Displacement}
\]

The elements of an effective-address computation are defined as follows:

- *Base*—A value stored in any general-purpose register.
- *Scale*—A positive value of 1, 2, 4, or 8.
- *Index*—A two’s-complement value stored in any general-purpose register.
- *Displacement*—An 8-bit, 16-bit, or 32-bit two’s-complement value encoded as part of the instruction.

Effective addresses are often referred to as *near pointers*. A near pointer is used when the segment selector is known implicitly or when the flat-memory model is used.

Long mode defines a 64-bit effective-address length. If a processor implementation does not support the full 64-bit virtual-address space, the effective address must be in *canonical form* (see “Canonical Address Form” on page 4).
Linear (Virtual) Addresses. The segment-selector portion of a logical address specifies a segment-descriptor entry in either the global or local descriptor table. The specified segment-descriptor entry contains the segment-base address, which is the starting location of the segment in linear-address space. A linear address is formed by adding the segment-base address to the effective address (segment offset), which creates a reference to any byte location within the supported linear-address space. Linear addresses are often referred to as virtual addresses, and both terms are used interchangeably throughout this document.

Linear Address = Segment Base Address + Effective Address

When the flat-memory model is used—as in 64-bit mode—a segment-base address is treated as 0. In this case, the linear address is identical to the effective address. In long mode, linear addresses must be in canonical address form, as described in “Canonical Address Form” on page 4.

Physical Addresses. A physical address is a reference into the physical-address space, typically main memory. Physical addresses are translated from virtual addresses using page-translation mechanisms. See “Paging” on page 7 for information on how the paging mechanism is used for virtual-address to physical-address translation. When the paging mechanism is not enabled, the virtual (linear) address is used as the physical address.

1.1.2 Memory Organization

The AMD64 architecture organizes memory into virtual memory and physical memory. Virtual-memory and physical-memory spaces can be (and usually are) different in size. Generally, the virtual-address space is much larger than physical-address memory. System software relocates applications and data between physical memory and the system hard disk to make it appear that much more memory is available than really exists. System software then uses the hardware memory-management mechanisms to map the larger virtual-address space into the smaller physical-address space.

Virtual Memory. Software uses virtual addresses to access locations within the virtual-memory space. System software is responsible for managing the relocation of applications and data in virtual-memory space using segment-memory management. System software is also responsible for mapping virtual memory to physical memory through the use of page translation. The AMD64 architecture supports different virtual-memory sizes using the following address-translation modes:

- **Protected Mode**—This mode supports 4 gigabytes of virtual-address space using 32-bit virtual addresses.
- **Long Mode**—This mode supports 16 exabytes of virtual-address space using 64-bit virtual addresses. A given implementation may however support less than this, as reported by the CPUID feature identification facility.
Physical Memory. Physical addresses are used to directly access main memory. For a particular computer system, the size of the available physical-address space is equal to the amount of main memory installed in the system. The maximum amount of physical memory accessible depends on the processor implementation and on the address-translation mode. The AMD64 architecture supports varying physical-memory sizes using the following address-translation modes:

- **Real-Address Mode**—This mode, also called real mode, supports 1 megabyte of physical-address space using 20-bit physical addresses. This address-translation mode is described in “Real Addressing” on page 10. Real mode is available only from legacy mode (see “Legacy Modes” on page 14).

- **Legacy Protected Mode**—This mode supports several different address-space sizes, depending on the translation mechanism used and whether extensions to those mechanisms are enabled.

  Legacy protected mode supports 4 gigabytes of physical-address space using 32-bit physical addresses. Both segment translation (see “Segmentation” on page 5) and page translation (see “Paging” on page 7) can be used to access the physical address space, when the processor is running in legacy protected mode.

  When the physical-address size extensions are enabled (see “Physical-Address Extensions (PAE) Bit” on page 123), the page-translation mechanism can be extended to support 52-bit physical addresses. 52-bit physical addresses allow up to 4 petabytes of physical-address space to be supported. (Currently, the AMD64 architecture supports 40-bit addresses in this mode, allowing up to 1 terabyte of physical-address space to be supported.

- **Long Mode**—This mode is unique to the AMD64 architecture. This mode supports up to 4 petabytes of physical-address space using 52-bit physical addresses. Long mode requires the use of page-translation and the physical-address size extensions (PAE).

1.1.3 Canonical Address Form

Long mode defines 64 bits of virtual-address space, but processor implementations can support less. Although some processor implementations do not use all 64 bits of the virtual address, they all check bits 63 through the most-significant implemented bit to see if those bits are all zeros or all ones. An address that complies with this property is in canonical address form. In most cases, a virtual-memory reference that is not in canonical form (in either the linear or effective form of the address) causes a general-protection exception (#GP) to occur. However, implied stack references where the stack address is not in canonical form causes a stack exception (#SS) to occur. Implied stack references include all push and pop instructions, and any instruction using RSP or RBP as a base register.

By checking canonical-address form, the AMD64 architecture prevents software from exploiting unused high bits of pointers for other purposes. Software complying with canonical-address form on a specific processor implementation can run unchanged on long-mode implementations supporting larger virtual-address spaces.
1.2 Memory Management

Memory management consists of the methods by which addresses generated by software are translated by segmentation and/or paging into addresses in physical memory. Memory management is not visible to application software. It is handled by the system software and processor hardware.

1.2.1 Segmentation

Segmentation was originally created as a method by which system software could isolate software processes (tasks), and the data used by those processes, from one another in an effort to increase the reliability of systems running multiple processes simultaneously.

The AMD64 architecture is designed to support all forms of legacy segmentation. However, most modern system software does not use the segmentation features available in the legacy x86 architecture. Instead, system software typically handles program and data isolation using page-level protection. For this reason, the AMD64 architecture dispenses with multiple segments in 64-bit mode and, instead, uses a flat-memory model. The elimination of segmentation allows new 64-bit system software to be coded more simply, and it supports more efficient management of multi-processing than is possible in the legacy x86 architecture.

Segmentation is, however, used in compatibility mode and legacy mode. Here, segmentation is a form of base memory-addressing that allows software and data to be relocated in virtual-address space off of an arbitrary base address. Software and data can be relocated in virtual-address space using one or more variable-sized memory segments. The legacy x86 architecture provides several methods of restricting access to segments from other segments so that software and data can be protected from interfering with each other.

In compatibility and legacy modes, up to 16,383 unique segments can be defined. The base-address value, segment size (called a limit), protection, and other attributes for each segment are contained in a data structure called a segment descriptor. Collections of segment descriptors are held in descriptor tables. Specific segment descriptors are referenced or selected from the descriptor table using a segment selector register. Six segment-selector registers are available, providing access to as many as six segments at a time.

Figure 1-1 on page 6 shows an example of segmented memory. Segmentation is described in Chapter 4, “Segmented Virtual Memory.”
Figure 1-1. Segmented-Memory Model

Flat Segmentation. One special case of segmented memory is the flat-memory model. In the legacy flat-memory model, all segment-base addresses have a value of 0, and the segment limits are fixed at 4 Gbytes. Segmentation cannot be disabled but use of the flat-memory model effectively disables segment translation. The result is a virtual address that equals the effective address. Figure 1-2 on page 7 shows an example of the flat-memory model.

Software running in 64-bit mode automatically uses the flat-memory model. In 64-bit mode, the segment base is treated as if it were 0, and the segment limit is ignored. This allows an effective addresses to access the full virtual-address space supported by the processor.
1.2.2 Paging

Paging allows software and data to be relocated in physical-address space using fixed-size blocks called physical pages. The legacy x86 architecture supports three different physical-page sizes of 4 Kbytes, 2 Mbytes, and 4 Mbytes. As with segment translation, access to physical pages by lesser-privileged software can be restricted.

Page translation uses a hierarchical data structure called a page-translation table to translate virtual pages into physical-pages. The number of levels in the translation-table hierarchy can be as few as one or as many as four, depending on the physical-page size and processor operating mode. Translation tables are aligned on 4-Kbyte boundaries. Physical pages must be aligned on 4-Kbyte, 2-Mbyte, or 4-Mbyte boundaries, depending on the physical-page size.

Each table in the translation hierarchy is indexed by a portion of the virtual-address bits. The entry referenced by the table index contains a pointer to the base address of the next-lower-level table in the translation hierarchy. In the case of the lowest-level table, its entry points to the physical-page base address. The physical page is then indexed by the least-significant bits of the virtual address to yield the physical address.

Figure 1-3 on page 8 shows an example of paged memory with three levels in the translation-table hierarchy. Paging is described in Chapter 5, “Page Translation and Protection.”
Software running in long mode is required to have page translation enabled.

**1.2.3 Mixing Segmentation and Paging**

Memory-management software can combine the use of segmented memory and paged memory. Because segmentation cannot be disabled, paged-memory management requires some minimum initialization of the segmentation resources. Paging can be completely disabled, so segmented-memory management does not require initialization of the paging resources.

Segments can range in size from a single byte to 4 Gbytes in length. It is therefore possible to map multiple segments to a single physical page and to map multiple physical pages to a single segment. Alignment between segment and physical-page boundaries is not required, but memory-management software is simplified when segment and physical-page boundaries are aligned.
The simplest, most efficient method of memory management is the flat-memory model. In the flat-memory model, all segment base addresses have a value of 0 and the segment limits are fixed at 4 Gbytes. The segmentation mechanism is still used each time a memory reference is made, but because virtual addresses are identical to effective addresses in this model, the segmentation mechanism is effectively ignored. Translation of virtual (or effective) addresses to physical addresses takes place using the paging mechanism only.

Because 64-bit mode disables segmentation, it uses a flat, paged-memory model for memory management. The 4 Gbyte segment limit is ignored in 64-bit mode. Figure 1-4 shows an example of this model.

![Figure 1-4. 64-Bit Flat, Paged-Memory Model](513-204.eps)
1.2.4 Real Addressing

Real addressing is a legacy-mode form of address translation used in real mode. This simplified form of address translation is backward compatible with 8086-processor effective-to-physical address translation. In this mode, 16-bit effective addresses are mapped to 20-bit physical addresses, providing a 1-Mbyte physical-address space.

Segment selectors are used in real-address translation, but not as an index into a descriptor table. Instead, the 16-bit segment-selector value is shifted left by 4 bits to form a 20-bit segment-base address. The 16-bit effective address is added to this 20-bit segment base address to yield a 20-bit physical address. If the sum of the segment base and effective address carries over into bit 20, that bit can be optionally truncated to mimic the 20-bit address wrapping of the 8086 processor by using the A20M# input signal to mask the A20 address bit.

A20 address bit masking should only be used real mode (see next section for information on real mode). Use in other modes may result in address translation errors.

Real-address translation supports a 1-Mbyte physical-address space using up to 64K segments aligned on 16-byte boundaries. Each segment is exactly 64 Kbytes long. Figure 1-5 shows an example of real-address translation.

![Real-Address Memory Model](image)

Figure 1-5. Real-Address Memory Model
1.3 Operating Modes

The legacy x86 architecture provides four operating modes or environments that support varying forms of memory management, virtual-memory and physical-memory sizes, and protection:

- Real Mode.
- Protected Mode.
- Virtual-8086 Mode.
- System Management Mode.

The AMD64 architecture supports all these legacy modes, and it adds a new operating mode called long mode. Table 1-1 shows the differences between long mode and legacy mode. Software can move between all supported operating modes as shown in Figure 1-6 on page 12. Each operating mode is described in the following sections.

Table 1-1. Operating Modes

<table>
<thead>
<tr>
<th>Mode</th>
<th>System Software Required</th>
<th>Application Recompile Required</th>
<th>Defaults¹</th>
<th>Register Extensions²</th>
<th>Maximum GPR Width (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Long Mode³</td>
<td>New 64-bit OS</td>
<td>yes</td>
<td>64</td>
<td>32</td>
<td>yes 64</td>
</tr>
<tr>
<td>Compatibility Mode</td>
<td>no</td>
<td></td>
<td>32</td>
<td>16 16</td>
<td>no 32</td>
</tr>
<tr>
<td>Legacy Mode</td>
<td></td>
<td>no</td>
<td>32 32</td>
<td>16 16</td>
<td>32</td>
</tr>
<tr>
<td>Protected Mode</td>
<td>Legacy 32-bit OS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual-8086 Mode</td>
<td>Legacy 16-bit OS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real Mode</td>
<td>Legacy 16-bit OS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note:
1. Defaults can be overridden in most modes using an instruction prefix or system control bit.
2. Register extensions include access to the upper eight general-purpose and YMM/XMM registers, uniform access to lower 8 bits of all GPRs, and access to the upper 32 bits of the GPRs.
3. Long mode supports only x86 protected mode. It does not support x86 real mode or virtual-8086 mode.
1.3.1 Long Mode

Long mode consists of two submodes: 64-bit mode and compatibility mode. 64-bit mode supports several new features, including the ability to address 64-bit virtual-address space. Compatibility mode provides binary compatibility with existing 16-bit and 32-bit applications when running on 64-bit system software.

Throughout this document, references to long mode refer collectively to both 64-bit mode and compatibility mode. If a function is specific to either 64-bit mode or compatibility mode, then those specific names are used instead of the name long mode.

Before enabling and activating long mode, system software must first enable protected mode. The process of enabling and activating long mode is described in Chapter 14, “Processor Initialization and
Long Mode Activation.” Long mode features are described throughout this document, where applicable.

1.3.2 64-Bit Mode

64-bit mode, a submode of long mode, provides support for 64-bit system software and applications by adding the following features:

- 64-bit virtual addresses (processor implementations can have fewer).
- Access to General Purpose Register bits 63:32
- Access to additional registers through the REX, VEX, and XOP instruction prefixes:
  - eight additional GPRs (R8–R15)
  - eight additional Streaming SIMD Extension (SSE) registers (YMM/XMM8–15)
- 64-bit instruction pointer (RIP).
- New RIP-relative data-addressing mode.
- Flat-segment address space with single code, data, and stack space.

The mode is enabled by the system software on an individual code-segment basis. Although code segments are used to enable and disable 64-bit mode, the legacy segmentation mechanism is largely disabled. Page translation is required for memory management purposes. Because 64-bit mode supports a 64-bit virtual-address space, it requires 64-bit system software and development tools.

In 64-bit mode, the default address size is 64 bits, and the default operand size is 32 bits. The defaults can be overridden on an instruction-by-instruction basis using instruction prefixes. A new REX prefix is introduced for specifying a 64-bit operand size and the new registers.

1.3.3 Compatibility Mode

Compatibility mode, a submode of long mode, allows system software to implement binary compatibility with existing 16-bit and 32-bit x86 applications. It allows these applications to run, without recompilation, under 64-bit system software in long mode, as shown in Table 1-1 on page 11.

In compatibility mode, applications can only access the first 4 Gbytes of virtual-address space. Standard x86 instruction prefixes toggle between 16-bit and 32-bit address and operand sizes.

Compatibility mode, like 64-bit mode, is enabled by system software on an individual code-segment basis. Unlike 64-bit mode, however, segmentation functions the same as in the legacy-x86 architecture, using 16-bit or 32-bit protected-mode semantics. From an application viewpoint, compatibility mode looks like a legacy protected-mode environment. From a system-software viewpoint, the long-mode mechanisms are used for address translation, interrupt and exception handling, and system data-structures.
1.3.4 Legacy Modes

Legacy mode consists of three submodes: real mode, protected mode, and virtual-8086 mode. Protected mode can be either paged or unpaged. Legacy mode preserves binary compatibility not only with existing x86 16-bit and 32-bit applications but also with existing x86 16-bit and 32-bit system software.

Real Mode. In this mode, also called real-address mode, the processor supports a physical-memory space of 1 Mbyte and operand sizes of 16 bits (default) or 32 bits (with instruction prefixes). Interrupt handling and address generation are nearly identical to the 80286 processor's real mode. Paging is not supported. All software runs at privilege level 0.

Real mode is entered after reset or processor power-up. The mode is not supported when the processor is operating in long mode because long mode requires that paged protected mode be enabled.

Protected Mode. In this mode, the processor supports virtual-memory and physical-memory spaces of 4 Gbytes and operand sizes of 16 or 32 bits. All segment translation, segment protection, and hardware multitasking functions are available. System software can use segmentation to relocate effective addresses in virtual-address space. If paging is not enabled, virtual addresses are equal to physical addresses. Paging can be optionally enabled to allow translation of virtual addresses to physical addresses and to use the page-based memory-protection mechanisms.

In protected mode, software runs at privilege levels 0, 1, 2, or 3. Typically, application software runs at privilege level 3, the system software runs at privilege levels 0 and 1, and privilege level 2 is available to system software for other uses. The 16-bit version of this mode was first introduced in the 80286 processor.

Virtual-8086 Mode. Virtual-8086 mode allows system software to run 16-bit real-mode software on a virtualized-8086 processor. In this mode, software written for the 8086, 8088, 80186, or 80188 processor can run as a privilege-level-3 task under protected mode. The processor supports a virtual-memory space of 1 Mbytes and operand sizes of 16 bits (default) or 32 bits (with instruction prefixes), and it uses real-mode address translation.

Virtual-8086 mode is enabled by setting the virtual-machine bit in the EFLAGS register (EFLAGS.VM). EFLAGS.VM can only be set or cleared when the EFLAGS register is loaded from the TSS as a result of a task switch, or by executing an IRET instruction from privileged software. The POPF instruction cannot be used to set or clear the EFLAGS.VM bit.

Virtual-8086 mode is not supported when the processor is operating in long mode. When long mode is enabled, any attempt to enable virtual-8086 mode is silently ignored.
1.3.5 System Management Mode (SMM)

System management mode (SMM) is an operating mode designed for system-control activities that are typically transparent to conventional system software. Power management is one popular use for system management mode. SMM is primarily targeted for use by platform firmware and specialized low-level device drivers. The code and data for SMM are stored in the SMM memory area, which is isolated from main memory by the SMM output signal.

SMM is entered by way of a system management interrupt (SMI). Upon recognizing an SMI, the processor enters SMM and switches to a separate address space where the SMM handler is located and executes. In SMM, the processor supports real-mode addressing with 4 Gbyte segment limits and default operand, address, and stack sizes of 16 bits (prefixes can be used to override these defaults).

1.4 System Registers

Figure 1-7 on page 16 shows the system registers defined for the AMD64 architecture. System software uses these registers to, among other things, manage the processor operating environment, define system resource characteristics, and to monitor software execution. With the exception of the RFLAGS register, system registers can be read and written only from privileged software.

Except for the descriptor-table registers and task register, the AMD64 architecture defines all system registers to be 64 bits wide. The descriptor table and task registers are defined by the AMD64 architecture to include 64-bit base-address fields, in addition to their other fields.

As shown in Figure 1-7 on page 16, the system registers include:

- **Control Registers**—These registers are used to control system operation and some system features. See “System-Control Registers” on page 41 for details.
- **System-Flags Register**—The RFLAGS register contains system-status flags and masks. It is also used to enable virtual-8086 mode and to control application access to I/O devices and interrupts. See “RFLAGS Register” on page 51 for details.
- **Descriptor-Table Registers**—These registers contain the location and size of descriptor tables stored in memory. Descriptor tables hold segmentation data structures used in protected mode. See “Descriptor Tables” on page 75 for details.
- **Task Register**—The task register contains the location and size in memory of the task-state segment. The hardware-multitasking mechanism uses the task-state segment to hold state information for a given task. The TSS also holds other data, such as the inner-level stack pointers used when changing to a higher privilege level. See “Task Register” on page 339 for details.
- **Debug Registers**—Debug registers are used to control the software-debug mechanism, and to report information back to a debug utility or application. See “Debug Registers” on page 356 for details.
Also defined as system registers are a number of model-specific registers included in the AMD64 architectural definition, and shown in Figure 1-7:

- **Extended-Feature-Enable Register**—The EFER register is used to enable and report status on special features not controlled by the CR\textsubscript{n} control registers. In particular, EFER is used to control activation of long mode. See “Extended Feature Enable Register (EFER)” on page 55 for more information.
• **System-Configuration Register**—The SYSCFG register is used to enable and configure system-bus features. See “System Configuration Register (SYSCFG)” on page 59 for more information.

• **System-Linkage Registers**—These registers are used by system-linkage instructions to specify operating-system entry points, stack locations, and pointers into system-data structures. See “Fast System Call and Return” on page 158 for details.

• **Memory-Typing Registers**—Memory-typing registers can be used to characterize (type) system memory. Typing memory gives system software control over how instructions and data are cached, and how memory reads and writes are ordered. See “MTRRs” on page 195 for details.

• **Debug-Extension Registers**—These registers control additional software-debug reporting features. See “Debug Registers” on page 356 for details.

• **Performance-Monitoring Registers**—Performance-monitoring registers are used to count processor and system events, or the duration of events. See “Performance Monitoring Counters” on page 370 for more information.

• **Machine-Check Registers**—The machine-check registers control the response of the processor to non-recoverable failures. They are also used to report information on such failures back to system utilities designed to respond to such failures. See “Machine Check Architecture MSRs” on page 273 for more information.

### 1.5 System-Data Structures

Figure 1-8 on page 18 shows the system-data structures defined for the AMD64 architecture. System-data structures are created and maintained by system software for use by the processor when running in protected mode. A processor running in protected mode uses these data structures to manage memory and protection, and to store program-state information when an interrupt or task switch occurs.
As shown in Figure 1-8, the system-data structures include:

- **Descriptors**—A descriptor provides information about a segment to the processor, such as its location, size and privilege level. A special type of descriptor, called a gate, is used to provide a code selector and entry point for a software routine. Any number of descriptors can be defined, but system software must at a minimum create a descriptor for the currently executing code segment and stack segment. See “Legacy Segment Descriptors” on page 82, and “Long-Mode Segment Descriptors” on page 90 for complete information on descriptors.

- **Descriptor Tables**—As the name implies, descriptor tables hold descriptors. The global-descriptor table holds descriptors available to all programs, while a local-descriptor table holds descriptors used by a single program. The interrupt-descriptor table holds only gate descriptors used by
interrupt handlers. System software must initialize the global-descriptor and interrupt-descriptor tables, while use of the local-descriptor table is optional. See “Descriptor Tables” on page 75 for more information.

- **Task-State Segment**—The task-state segment is a special segment for holding processor-state information for a specific program, or task. It also contains the stack pointers used when switching to more-privileged programs. The hardware multitasking mechanism uses the state information in the segment when suspending and resuming a task. Calls and interrupts that switch stacks cause the stack pointers to be read from the task-state segment. System software must create at least one task-state segment, even if hardware multitasking is not used. See “Legacy Task-State Segment” on page 341, and “64-Bit Task State Segment” on page 345 for details.

- **Page-Translation Tables**—Use of page translation is optional in protected mode, but it is required in long mode. A four-level page-translation data structure is provided to allow long-mode operating systems to translate a 64-bit virtual-address space into a 52-bit physical-address space. Legacy protected mode can use two- or three-level page-translation data structures. See “Page Translation Overview” on page 120 for more information on page translation.

### 1.6 Interrupts

The AMD64 architecture provides a mechanism for the processor to automatically suspend (interrupt) software execution and transfer control to an interrupt handler when an interrupt or exception occurs. An interrupt handler is privileged software designed to identify and respond to the cause of an interrupt or exception, and return control back to the interrupted software. *Interrupts* can be caused when system hardware signals an interrupt condition using one of the external-interrupt signals on the processor. Interrupts can also be caused by software that executes an interrupt instruction. *Exceptions* occur when the processor detects an abnormal condition as a result of executing an instruction. The term “interrupts” as used throughout this volume includes both interrupts and exceptions when the distinction is unnecessary.

System software not only sets up the interrupt handlers, but it must also create and initialize the data structures the processor uses to execute an interrupt handler when an interrupt occurs. The data structures include the code-segment descriptors for the interrupt-handler software and any data-segment descriptors for data and stack accesses. Interrupt-gate descriptors must also be supplied. Interrupt gates point to interrupt-handler code-segment descriptors, and the entry point in an interrupt handler. Interrupt gates are stored in the interrupt-descriptor table. The code-segment and data-segment descriptors are stored in the global-descriptor table and, optionally, the local-descriptor table.

When an interrupt occurs, the processor uses the interrupt vector to find the appropriate interrupt gate in the interrupt-descriptor table. The gate points to the interrupt-handler code segment and entry point, and the processor transfers control to that location. Before invoking the interrupt handler, the processor saves information required to return to the interrupted program. For details on how the processor transfers control to interrupt handlers, see “Legacy Protected-Mode Interrupt Control Transfers” on page 245, and “Long-Mode Interrupt Control Transfers” on page 255.
Table 1-2 shows the supported interrupts and exceptions, ordered by their vector number. Refer to “Vectors” on page 222 for a complete description of each interrupt, and a description of the interrupt mechanism.

<table>
<thead>
<tr>
<th>Vector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Integer Divide-by-Zero Exception</td>
</tr>
<tr>
<td>1</td>
<td>Debug Exception</td>
</tr>
<tr>
<td>2</td>
<td>Non-Maskable-Interrupt</td>
</tr>
<tr>
<td>3</td>
<td>Breakpoint Exception (INT 3)</td>
</tr>
<tr>
<td>4</td>
<td>Overflow Exception (INTO instruction)</td>
</tr>
<tr>
<td>5</td>
<td>Bound-Range Exception (BOUND instruction)</td>
</tr>
<tr>
<td>6</td>
<td>Invalid-Opcode Exception</td>
</tr>
<tr>
<td>7</td>
<td>Device-Not-Available Exception</td>
</tr>
<tr>
<td>8</td>
<td>Double-Fault Exception</td>
</tr>
<tr>
<td>9</td>
<td>Coprocessor-Segment-Overrun Exception (reserved in AMD64)</td>
</tr>
<tr>
<td>10</td>
<td>Invalid-TSS Exception</td>
</tr>
<tr>
<td>11</td>
<td>Segment-Not-Present Exception</td>
</tr>
<tr>
<td>12</td>
<td>Stack Exception</td>
</tr>
<tr>
<td>13</td>
<td>General-Protection Exception</td>
</tr>
<tr>
<td>14</td>
<td>Page-Fault Exception</td>
</tr>
<tr>
<td>15</td>
<td>(Reserved)</td>
</tr>
<tr>
<td>16</td>
<td>x87 Floating-Point Exception</td>
</tr>
<tr>
<td>17</td>
<td>Alignment-Check Exception</td>
</tr>
<tr>
<td>18</td>
<td>Machine-Check Exception</td>
</tr>
<tr>
<td>19</td>
<td>SIMD Floating-Point Exception</td>
</tr>
<tr>
<td>0–255</td>
<td>Interrupt Instructions</td>
</tr>
<tr>
<td>0–255</td>
<td>Hardware Maskable Interrupts</td>
</tr>
</tbody>
</table>

1.7 Additional System-Programming Facilities

1.7.1 Hardware Multitasking

A task is any program that the processor can execute, suspend, and later resume executing at the point of suspension. During the time a task is suspended, other tasks are allowed to execute. Each task has its own execution space, consisting of a code segment, data segments, and a stack segment for each privilege level. Tasks can also have their own virtual-memory environment managed by the page-translation mechanism. The state information defining this execution space is stored in the task-state segment (TSS) maintained for each task.
Support for hardware multitasking is provided by implementations of the AMD64 architecture when software is running in legacy mode. Hardware multitasking provides automated mechanisms for switching tasks, saving the execution state of the suspended task, and restoring the execution state of the resumed task. When hardware multitasking is used to switch tasks, the processor takes the following actions:

- The processor automatically suspends execution of the task, allowing any executing instructions to complete and save their results.
- The execution state of a task is saved in the task TSS.
- The execution state of a new task is loaded into the processor from its TSS.
- The processor begins executing the new task at the location specified in the new task TSS.

Use of hardware-multitasking features is optional in legacy mode. Generally, modern operating systems do not use the hardware-multitasking features, and instead perform task management entirely in software. Long mode does not support hardware multitasking at all.

Whether hardware multitasking is used or not, system software must create and initialize at least one task-state segment data-structure. This requirement holds for both long-mode and legacy-mode software. The single task-state segment holds critical pieces of the task execution environment and is referenced during certain control transfers.

Detailed information on hardware multitasking is available in Chapter 12, “Task Management,” along with a full description of the requirements that must be met in initializing a task-state segment when hardware multitasking is not used.

1.7.2 Machine Check

Implementations of the AMD64 architecture support the machine-check exception. This exception is useful in system applications with stringent requirements for reliability, availability, and serviceability. The exception allows specialized system-software utilities to report hardware errors that are generally severe and non-recoverable. Providing the capability to report such errors can allow complex system problems to be pinpointed rapidly.

The machine-check exception is described in Chapter 9, “Machine Check Architecture.” Much of the error-reporting capabilities is implementation dependent. For more information, developers of machine-check error-reporting software should refer to the BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual or applicable to your product.

1.7.3 Software Debugging

A software-debugging mechanism is provided in hardware to help software developers quickly isolate programming errors. This capability can be used to debug system software and application software alike. Only privileged software can access the debugging facilities. Generally, software-debug support is provided by a privileged application program rather than by the operating system itself.

The facilities supported by the AMD64 architecture allow debugging software to perform the following:
• Set breakpoints on specific instructions within a program.
• Set breakpoints on an instruction-address match.
• Set breakpoints on a data-address match.
• Set breakpoints on specific I/O-port addresses.
• Set breakpoints to occur on task switches when hardware multitasking is used.
• Single step an application instruction-by-instruction.
• Single step only branches and interrupts.
• Record a history of branches and interrupts taken by a program.

The debugging facilities are fully described in “Software-Debug Resources” on page 356. Some processors provide additional, implementation-specific debug support. For more information, refer to the BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual applicable to your product.

1.7.4 Performance Monitoring

For many software developers, the ability to identify and eliminate performance bottlenecks from a program is nearly as important as quickly isolating programming errors. Implementations of the AMD64 architecture provide hardware performance-monitoring resources that can be used by special software applications to identify such bottlenecks. Non-privileged software can access the performance monitoring facilities, but only if privileged software grants that access.

The performance-monitoring facilities allow the counting of events, or the duration of events. Performance-analysis software can use the data to calculate the frequency of certain events, or the time spent performing specific activities. That information can be used to suggest areas for improvement and the types of optimizations that are helpful.

The performance-monitoring facilities are fully described in “Performance Monitoring Counters” on page 370. The specific events that can be monitored are generally implementation specific. For more information, refer to the BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual applicable to your product.
2 x86 and AMD64 Architecture Differences

The AMD64 architecture is designed to provide full binary compatibility with all previous AMD implementations of the x86 architecture. This chapter summarizes the new features and architectural enhancements introduced by the AMD64 architecture, and compares those features and enhancements with previous AMD x86 processors. Most of the new capabilities introduced by the AMD64 architecture are available only in long mode (64-bit mode, compatibility mode, or both). However, some of the new capabilities are also available in legacy mode, and are mentioned where appropriate.

The material throughout this chapter assumes the reader has a solid understanding of the x86 architecture. For those who are unfamiliar with the x86 architecture, please read the remainder of this volume before reading this chapter.

2.1 Operating Modes

See “Operating Modes” on page 11 for a complete description of the operating modes supported by the AMD64 architecture.

2.1.1 Long Mode

The AMD64 architecture introduces long mode and its two sub-modes: 64-bit mode and compatibility mode.

64-Bit Mode. 64-bit mode provides full support for 64-bit system software and applications. The new features introduced in support of 64-bit mode are summarized throughout this chapter. To use 64-bit mode, a 64-bit operating system and tool chain are required.

Compatibility Mode. Compatibility mode allows 64-bit operating systems to implement binary compatibility with existing 16-bit and 32-bit x86 applications. It allows these applications to run, without recompilation, under control of a 64-bit operating system in long mode. The architectural enhancements introduced by the AMD64 architecture that support compatibility mode are summarized throughout this chapter.

Unsupported Modes. Long mode does not support the following two operating modes:

- Virtual-8086 Mode—The virtual-8086 mode bit (EFLAGS.VM) is ignored when the processor is running in long mode. When long mode is enabled, any attempt to enable virtual-8086 mode is silently ignored. System software must leave long mode in order to use virtual-8086 mode.
- Real Mode—Real mode is not supported when the processor is operating in long mode because long mode requires that protected mode be enabled.

2.1.2 Legacy Mode

The AMD64 architecture supports a pure x86 legacy mode, which preserves binary compatibility not only with existing 16-bit and 32-bit applications but also with existing 16-bit and 32-bit operating
systems. Legacy mode supports real mode, protected mode, and virtual-8086 mode. A reset always places the processor in legacy mode (real mode), and the processor continues to run in legacy mode until system software activates long mode. New features added by the AMD64 architecture that are supported in legacy mode are summarized in this chapter.

### 2.1.3 System-Management Mode

The AMD64 architecture supports system-management mode (SMM). SMM can be entered from both long mode and legacy mode, and SMM can return directly to either mode. The following differences exist between the support of SMM in the AMD64 architecture and the SMM support found in previous processor generations:

- The SMRAM state-save area format is changed to hold the 64-bit processor state. This state-save area format is used regardless of whether SMM is entered from long mode or legacy mode.
- The auto-halt restart and I/O-instruction restart entries in the SMRAM state-save area are one byte instead of two bytes.
- The initial processor state upon entering SMM is expanded to reflect the 64-bit nature of the processor.
- New conditions exist that can cause a processor shutdown while exiting SMM.
- SMRAM caching considerations are modified because the legacy FLUSH# external signal (writeback, if modified, and invalidate) is not supported on implementations of the AMD64 architecture.


### 2.2 Memory Model

The AMD64 architecture provides enhancements to the legacy memory model to support very large physical-memory and virtual-memory spaces while in long mode. Some of this expanded support for physical memory is available in legacy mode.

#### 2.2.1 Memory Addressing

**Virtual-Memory Addressing.** Virtual-memory support is expanded to 64 address bits in long mode. This allows up to 16 exabytes of virtual-address space to be accessed. The virtual-address space supported in legacy mode is unchanged.

**Physical-Memory Addressing.** Physical-memory support is expanded to 52 address bits in long mode and legacy mode. This allows up to 4 petabytes of physical memory to be accessed. The expanded physical-memory support is achieved by using paging and the page-size extensions.

Note that given processor may implement less than the architecturally-defined physical address size of 52 bits.
Effective Addressing. The effective-address length is expanded to 64 bits in long mode. An effective-address calculation uses 64-bit base and index registers, and sign-extends 8-bit and 32-bit displacements to 64 bits. In legacy mode, effective addresses remain 32 bits long.

2.2.2 Page Translation

The AMD64 architecture defines an expanded page-translation mechanism supporting translation of a 64-bit virtual address to a 52-bit physical address. See “Long-Mode Page Translation” on page 132 for detailed information on the enhancements to page translation in the AMD64 architecture. The enhancements are summarized below.

Physical-Address Extensions (PAE). The AMD64 architecture requires physical-address extensions to be enabled (CR4.PAE=1) before long mode is entered. When PAE is enabled, all paging data-structures are 64 bits, allowing references into the full 52-bit physical-address space supported by the architecture.

Page-Size Extensions (PSE). Page-size extensions (CR4.PSE) are ignored in long mode. Long mode does not support the 4-Mbyte page size enabled by page-size extensions. Long mode does, however, support 4-Kbyte and 2-Mbyte page sizes.

Paging Data Structures. The AMD64 architecture extends the page-translation data structures in support of long mode. The extensions are:

- **Page-map level-4 (PML4)**—Long mode defines a new page-translation data structure, the PML4 table. The PML4 table sits at the top of the page-translation hierarchy and references PDP tables.
- **Page-directory pointer (PDP)**—The PDP tables in long mode are expanded from 4 entries to 512 entries each.
- **Page-directory pointer entry (PDPE)**—Previously undefined fields within the legacy-mode PDPE are defined by the AMD64 architecture.

CR3 Register. The CR3 register is expanded to 64 bits for use in long-mode page translation. When long mode is active, the CR3 register references the base address of the PML4 table. In legacy mode, the upper 32 bits of CR3 are masked by the processor to support legacy page translation. CR3 references the PDP base-address when physical-address extensions are enabled, or the page-directory table base-address when physical-address extensions are disabled.

Legacy-Mode Enhancements. Legacy-mode software can take advantage of the enhancements made to the physical-address extension (PAE) support and page-size extension (PSE) support. The four-level page translation mechanism introduced by long mode is not available to legacy-mode software.

- **PAE**—When physical-address extensions are enabled (CR4.PAE=1), the AMD64 architecture allows legacy-mode software to load up to 52-bit (maximum size) physical addresses into the PDE and PTE. Note that addresses are expanded to the maximum physical address size supported by the implementation.
• **PSE**—The use of page-size extensions allows legacy mode software to define 4-Mbyte pages using the 32-bit page-translation tables. When page-size extensions are enabled (CR4.PSE=1), the AMD64 architecture enhances the 4-Mbyte PDE to support 40 physical-address bits.

See “Legacy-Mode Page Translation” on page 124 for more information on these enhancements.

### 2.2.3 Segmentation

In long mode, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode:

- In compatibility mode, segmentation functions just as it does in legacy mode, using legacy 16-bit or 32-bit protected mode semantics.
- 64-bit mode requires a flat-memory model for creating a flat 64-bit virtual-address space. Much of the segmentation capability present in legacy mode and compatibility mode is disabled when the processor is running in 64-bit mode.

The differences in the segmentation model as defined by the AMD64 architecture are summarized in the following sections. See Chapter 4, “Segmented Virtual Memory,” for a thorough description of these differences.

**Descriptor-Table Registers.** In long mode, the base-address portion of the descriptor-table registers (GDTR, IDTR, LDTR, and TR) are expanded to 64 bits. The full 64-bit base address can only be loaded by software when the processor is running in 64-bit mode (using the LGDT, LIDT, LLDT, andLTR instructions, respectively). However, the full 64-bit base address is **used** by a processor running in compatibility mode (in addition to 64-bit mode) when making a reference into a descriptor table.

A processor running in legacy mode can only load the low 32 bits of the base address, and the high 32 bits are ignored when references are made to the descriptor tables.

**Code-Segment Descriptors.** The AMD64 architecture defines a new code-segment descriptor attribute, L (long). In compatibility mode, the processor treats code-segment descriptors as it does in legacy mode, with the exception that the processor recognizes the L attribute. If a code descriptor with L=1 is loaded in compatibility mode, the processor leaves compatibility mode and enters 64-bit mode. In legacy mode, the L attribute is reserved.

The following differences exist for code-segment descriptors in 64-bit mode only:

- The CS base-address field is ignored by the processor.
- The CS limit field is ignored by the processor.
- Only the L (long), D (default size), and DPL (descriptor-privilege level) fields are used by the processor in 64-bit mode. All remaining attributes are ignored.

**Data-Segment Descriptors.** The following differences exist for data-segment descriptors in 64-bit mode only:

- The DS, ES, and SS descriptor base-address fields are ignored by the processor.
• The FS and GS descriptor base-address fields are expanded to 64 bits and used in effective-address calculations. The 64 bits of base address are mapped to model-specific registers (MSRs), and can only be loaded using the WRMSR instruction.

• The limit fields and attribute fields of all data-segment descriptors (DS, ES, FS, GS, and SS) are ignored by the processor.

In compatibility mode, the processor treats data-segment descriptors as it does in legacy mode. Compatibility mode ignores the high 32 bits of base address in the FS and GS segment descriptors when calculating an effective address.

**System-Segment Descriptors.** In 64-bit mode only, The LDT and TSS system-segment descriptor formats are expanded by 64 bits, allowing them to hold 64-bit base addresses. LLDT and LTR instructions can be used to load these descriptors into the LDTR and TR registers, respectively, from 64-bit mode.

In compatibility mode and legacy mode, the **formats** of the LDT and TSS system-segment descriptors are unchanged. Also, unlike code-segment and data-segment descriptors, system-segment descriptor limits are **checked** by the processor in long mode.

Some legacy mode LDT and TSS type-field encodings are illegal in long mode (both compatibility mode and 64-bit mode), and others are redefined to new types. See “System Descriptors” on page 92 for additional information.

**Gate Descriptors.** The following differences exist between gate descriptors in long mode (both compatibility mode and 64-bit mode) and in legacy mode:

• In long mode, all 32-bit gate descriptors are redefined as 64-bit gate descriptors, and are expanded to hold 64-bit offsets. The length of a gate descriptor in long mode is therefore 128 bits (16 bytes), versus the 64 bits (8 bytes) in legacy mode.

• Some type-field encodings are illegal in long mode, and others are redefined to new types. See “Gate Descriptors” on page 94 for additional information.

• The interrupt-gate and trap-gate descriptors define a new field, called the interrupt-stack table (IST) field.

**2.3 Protection Checks**

The AMD64 architecture makes the following changes to the protection mechanism in long mode:

• The page-protection-check mechanism is expanded in long mode to include the U/S and R/W protection bits stored in the PML4 entries and PDP entries.

• Several system-segment types and gate-descriptor types that are legal in legacy mode are illegal in long mode (compatibility mode and 64-bit mode) and fail type checks when used in long mode.

• Segment-limit checks are disabled in 64-bit mode for the CS, DS, ES, FS, GS, and SS segments. Segment-limit checks remain enabled for the LDT, GDT, IDT and TSS system segments.

All segment-limit checks are performed in compatibility mode.
• Code and data segments used in 64-bit mode are treated as both readable and writable.

See “Page-Protection Checks” on page 149 and “Segment-Protection Overview” on page 97 for detailed information on the protection-check changes.

2.4 Registers

The AMD64 architecture adds additional registers to the architecture, and in many cases expands the size of existing registers to 64 bits. The 80-bit floating-point stack registers and their overlaid 64-bit MMX™ registers are not modified by the AMD64 architecture.

2.4.1 General-Purpose Registers

In 64-bit mode, the general-purpose registers (GPRs) are 64 bits wide, and eight additional GPRs are available. The GPRs are: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, and the new R8–R15 registers. To access the full 64-bit operand size, or the new R8–R15 registers, an instruction must include a new REX instruction-prefix byte (see “REX Prefixes” on page 29 for a summary of this prefix).

In compatibility and legacy modes, the GPRs consist only of the eight legacy 32-bit registers. All legacy rules apply for determining operand size.

2.4.2 YMM/XMM Registers

In 64-bit mode, eight additional YMM/XMM registers are available, YMM/XMM8–15. A REX instruction prefix is used to access these registers. In compatibility and legacy modes, only registers YMM/XMM0–7 are accessible.

2.4.3 Flags Register

The flags register is expanded to 64 bits, and is called RFLAGS. All 64 bits can be accessed in 64-bit mode, but the upper 32 bits are reserved and always read back as zeros. Compatibility mode and legacy mode can read and write only the lower-32 bits of RFLAGS (the legacy EFLAGS).

2.4.4 Instruction Pointer

In long mode, the instruction pointer is extended to 64 bits, to support 64-bit code offsets. This 64-bit instruction pointer is called RIP.

2.4.5 Stack Pointer

In 64-bit mode, the size of the stack pointer, RSP, is always 64 bits. The stack size is not controlled by a bit in the SS descriptor, as it is in compatibility or legacy mode, nor can it be overridden by an instruction prefix. Address-size overrides are ignored for implicit stack references.
2.4.6 Control Registers

The AMD64 architecture defines several enhancements to the control registers (CRn). In long mode, all control registers are expanded to 64 bits, although the entire 64 bits can be read and written only from 64-bit mode. A new control register, the task-priority register (CR8 or TPR) is added, and can be read and written from 64-bit mode. Last, the function of the page-enable bit (CR0.PG) is expanded. When long mode is enabled, the PG bit is used to activate and deactivate long mode.

2.4.7 Debug Registers

In long mode, all debug registers are expanded to 64 bits, although the entire 64 bits can be read and written only from 64-bit mode. Expanded register encodings for the decode registers allow up to eight new registers to be defined (DR8–DR15), although presently those registers are not supported by the AMD64 architecture.

2.4.8 Extended Feature Register (EFER)

The EFER is expanded by the AMD64 architecture to include a long-mode-enable bit (LME), and a long-mode-active bit (LMA). These new bits can be accessed from legacy mode and long mode.

2.4.9 Memory Type Range Registers (MTRRs)

The legacy MTRRs are architecturally defined as 64 bits, and can accommodate the maximum 52-bit physical address allowed by the AMD64 architecture. From both long mode and legacy mode, implementations of the AMD64 architecture reference the entire 52-bit physical-address value stored in the MTRRs. Long mode and legacy mode system software can update all 64 bits of the MTRRs to manage the expanded physical-address space.

2.4.10 Other Model-Specific Registers (MSRs)

Several other MSRs have fields holding physical addresses. Examples include the APIC-base register and top-of-memory register. Generally, any model-specific register that contains a physical address is defined architecturally to be 64 bits wide, and can accommodate the maximum physical-address size defined by the AMD64 architecture. When physical addresses are read from MSRs by the processor, the entire value is read regardless of the operating mode. In legacy implementations, the high-order MSR bits are reserved, and software must write those values with zeros. In legacy mode on AMD64 architecture implementations, software can read and write all supported high-order MSR bits.

2.5 Instruction Set

2.5.1 REX Prefixes

REX prefixes are used in 64-bit mode to:

- Specify the new GPRs and YMM/XMM registers.
- Specify a 64-bit operand size.
Specify additional control registers. One additional control register, CR8, is defined in 64-bit mode.

Specify additional debug registers (although none are currently defined).

Not all instructions require a REX prefix. The prefix is necessary only if an instruction references one of the extended registers or uses a 64-bit operand. If a REX prefix is used when it has no meaning, it is ignored.

**Default 64-Bit Operand Size.** In 64-bit mode, two groups of instructions have a default operand size of 64 bits and thus do not need a REX prefix for this operand size:

- Near branches.
- All instructions, except far branches, that implicitly reference the RSP. See “Instructions that Reference RSP” on page 31 for additional information.

### 2.5.2 Segment-Override Prefixes in 64-Bit Mode

In 64-bit mode, the DS, ES, SS, and CS segment-override prefixes have no effect. These four prefixes are no longer treated as segment-override prefixes in the context of multiple-prefix rules. Instead, they are treated as null prefixes.

The FS and GS segment-override prefixes are treated as segment-override prefixes in 64-bit mode. Use of the FS and GS prefixes cause their respective segment bases to be added to the effective address calculation. See “FS and GS Registers in 64-Bit Mode” on page 74 for additional information on using these segment registers.

### 2.5.3 Operands and Results

The AMD64 architecture provides support for using 64-bit operands and generating 64-bit results when operating in 64-bit mode.

**Operand-Size Overrides.** In 64-bit mode, the default operand size is 32 bits. A REX prefix can be used to specify a 64-bit operand size. Software uses a legacy operand-size (66h) prefix to toggle to 16-bit operand size. The REX prefix takes precedence over the legacy operand-size prefix.

**Zero Extension of Results.** In 64-bit mode, when performing 32-bit operations with a GPR destination, the processor zero-extends the 32-bit result into the full 64-bit destination. Both 8-bit and 16-bit operations on GPRs preserve all unwritten upper bits of the destination GPR. This is consistent with legacy 16-bit and 32-bit semantics for partial-width results.

### 2.5.4 Address Calculations

The AMD64 architecture modifies aspects of effective-address calculation to support 64-bit mode. These changes are summarized in the following sections. See “Memory Addressing” in Volume 1 for details.
Address-Size Overrides. In 64-bit mode, the default-address size is 64 bits. The address size can be overridden to 32 bits by using the address-size prefix (67h). 16-bit addresses are not supported in 64-bit mode. In compatibility mode and legacy mode, address-size overrides function the same as in x86 legacy architecture.

Displacements and Immediates. Generally, displacement and immediate values in 64-bit mode are not extended to 64 bits. They are still limited to 32 bits and are sign extended during effective-address calculations. In 64-bit mode, however, support is provided for some 64-bit displacement and immediate forms of the MOV instruction.

Zero Extending 16-Bit and 32-Bit Addresses. All 16-bit and 32-bit address calculations are zero-extended in long mode to form 64-bit addresses. Address calculations are first truncated to the effective-address size of the current mode (64-bit mode or compatibility mode), as overridden by any address-size prefix. The result is then zero-extended to the full 64-bit address width.

RIP-Relative Addressing. A new addressing form, RIP-relative (instruction-pointer relative) addressing, is implemented in 64-bit mode. The effective address is formed by adding the displacement to the 64-bit RIP of the next instruction.

2.5.5 Instructions that Reference RSP

With the exception of far branches, all instructions that implicitly reference the 64-bit stack pointer, RSP, default to a 64-bit operand size in 64-bit mode (see Table 2-1 for a listing). Pushes and pops of 32-bit stack values are not possible in 64-bit mode with these instructions, but they can be overridden to 16 bits.

Table 2-1. Instructions That Reference RSP

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Opcode (hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENTER</td>
<td>C8</td>
<td>Create Procedure Stack Frame</td>
</tr>
<tr>
<td>LEAVE</td>
<td>C9</td>
<td>Delete Procedure Stack Frame</td>
</tr>
<tr>
<td>POP reg/mem</td>
<td>8F/0</td>
<td>Pop Stack (register or memory)</td>
</tr>
<tr>
<td>POP reg</td>
<td>58-5F</td>
<td>Pop Stack (register)</td>
</tr>
<tr>
<td>POP FS</td>
<td>0F A1</td>
<td>Pop Stack into FS Segment Register</td>
</tr>
<tr>
<td>POP GS</td>
<td>0F A9</td>
<td>Pop Stack into GS Segment Register</td>
</tr>
<tr>
<td>POPF, POPFD, POPFQ</td>
<td>9D</td>
<td>Pop to rFLAGS Word, Doubleword, or Quadword</td>
</tr>
<tr>
<td>PUSH imm32</td>
<td>68</td>
<td>Push onto Stack (sign-extended doubleword)</td>
</tr>
<tr>
<td>PUSH imm8</td>
<td>6A</td>
<td>Push onto Stack (sign-extended byte)</td>
</tr>
<tr>
<td>PUSH reg/mem</td>
<td>FF/6</td>
<td>Push onto Stack (register or memory)</td>
</tr>
<tr>
<td>PUSH reg</td>
<td>50-57</td>
<td>Push onto Stack (register)</td>
</tr>
<tr>
<td>PUSH FS</td>
<td>0F A0</td>
<td>Push FS Segment Register onto Stack</td>
</tr>
<tr>
<td>PUSH GS</td>
<td>0F A8</td>
<td>Push GS Segment Register onto Stack</td>
</tr>
<tr>
<td>PUSHF, PUSHFD, PUSHFQ</td>
<td>9C</td>
<td>Push rFLAGS Word, Doubleword, or Quadword onto Stack</td>
</tr>
</tbody>
</table>
2.5.6 Branches

The AMD64 architecture expands two branching mechanisms to accommodate branches in the full 64-bit virtual-address space:

- In 64-bit mode, near-branch semantics are redefined.
- In both 64-bit and compatibility modes, a 64-bit call-gate descriptor is defined for far calls.

In addition, enhancements are made to the legacy SYSCALL and SYSRET instructions.

Near Branches. In 64-bit mode, the operand size for all near branches defaults to 64 bits (see Table 2-2 for a listing). Therefore, these instructions update the full 64-bit RIP without the need for a REX operand-size prefix. The following aspects of near branches default to 64 bits:

- Truncation of the instruction pointer.
- Size of a stack pop or stack push, resulting from a CALL or RET.
- Size of a stack-pointer increment or decrement, resulting from a CALL or RET.
- Size of operand fetched by indirect-branch operand size.

The operand size for near branches can be overridden to 16 bits in 64-bit mode.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Opcode (hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>E8, FF/2</td>
<td>Call Procedure Near</td>
</tr>
<tr>
<td>Jcc</td>
<td>many</td>
<td>Jump Conditional Near</td>
</tr>
<tr>
<td>JMP</td>
<td>E9, EB, FF/4</td>
<td>Jump Near</td>
</tr>
<tr>
<td>LOOP</td>
<td>E2</td>
<td>Loop</td>
</tr>
<tr>
<td>LOOPcc</td>
<td>E0, E1</td>
<td>Loop Conditional</td>
</tr>
<tr>
<td>RET</td>
<td>C3, C2</td>
<td>Return From Call (near)</td>
</tr>
</tbody>
</table>

The address size of near branches is not forced in 64-bit mode. Such addresses are 64 bits by default, but they can be overridden to 32 bits by a prefix.

The size of the displacement field for relative branches is still limited to 32 bits.

Far Branches Through Long-Mode Call Gates. Long mode redefines the 32-bit call-gate descriptor type as a 64-bit call-gate descriptor and expands the call-gate descriptor size to hold a 64-bit offset. The long-mode call-gate descriptor allows far branches to reference any location in the supported virtual-address space. In long mode, the call-gate mechanism is changed as follows:

- In long mode, CALL and JMP instructions that reference call-gates must reference 64-bit call gates.
- A 64-bit call-gate descriptor must reference a 64-bit code-segment.
• When a control transfer is made through a 64-bit call gate, the 64-bit target address is read from the 64-bit call-gate descriptor. The base address in the target code-segment descriptor is ignored.

**Stack Switching.** Automatic stack switching is also modified when a control transfer occurs through a call gate in long mode:

• The target-stack pointer read from the TSS is a 64-bit RSP value.
• The SS register is loaded with a null selector. Setting the new SS selector to null allows nested control transfers in 64-bit mode to be handled properly. The SS.RPL value is updated to remain consistent with the newly loaded CPL value.
• The size of pushes onto the new stack is modified to accommodate the 64-bit RIP and RSP values.
• Automatic parameter copying is not supported in long mode.

**Far Returns.** In long mode, far returns can load a null SS selector from the stack under the following conditions:

• The target operating mode is 64-bit mode.
• The target CPL<3.

Allowing RET to load SS with a null selector under these conditions makes it possible for the processor to unnest far CALLs (and interrupts) in long mode.

**Task Gates.** Control transfers through task gates are not supported in long mode.

**Branches to 64-Bit Offsets.** Because immediate values are generally limited to 32 bits, the only way a full 64-bit absolute RIP can be specified in 64-bit mode is with an indirect branch. For this reason, direct forms of far branches are eliminated from the instruction set in 64-bit mode.

**SYSCALL and SYSRET Instructions.** The AMD64 architecture expands the function of the legacy SYSCALL and SYSRET instructions in long mode. In addition, two new STAR registers, LSTAR and CSTAR, are provided to hold the 64-bit target RIP for the instructions when they are executed in long mode. The legacy STAR register is not expanded in long mode. See “SYSCALL and SYSRET” on page 159 for additional information.

**SWAPGS Instruction.** The AMD64 architecture provides the SWAPGS instruction as a fast method for system software to load a pointer to system data-structures. SWAPGS is valid only in 64-bit mode. An undefined-opcode exception (#UD) occurs if software attempts to execute SWAPGS in legacy mode or compatibility mode. See “SWAPGS Instruction” on page 161 for additional information.

**SYSENTER and SYSEXIT Instructions.** The SYSENTER and SYSEXIT instructions are invalid in long mode, and result in an invalid opcode exception (#UD) if software attempts to use them. Software should use the SYSCALL and SYSRET instructions when running in long mode. See “SYSENTER and SYSEXIT (Legacy Mode Only)” on page 160 for additional information.
2.5.7 NOP Instruction

The legacy x86 architecture commonly uses opcode 90h as a one-byte NOP. In 64-bit mode, the processor treats opcode 90h specially in order to preserve this NOP definition. This is necessary because opcode 90h is actually the XCHG EAX, EAX instruction in the legacy architecture. Without special handling in 64-bit mode, the instruction would not be a true no-operation. Therefore, in 64-bit mode the processor treats opcode 90h (the legacy XCHG EAX, EAX instruction) as a true NOP, regardless of a REX operand-size prefix.

This special handling does not apply to the two-byte ModRM form of the XCHG instruction. Unless a 64-bit operand size is specified using a REX prefix byte, using the two-byte form of XCHG to exchange a register with itself does not result in a no-operation, because the default operation size is 32 bits in 64-bit mode.

2.5.8 Single-Byte INC and DEC Instructions

In 64-bit mode, the legacy encodings for the 16 single-byte INC and DEC instructions (one for each of the eight GPRs) are used to encode the REX prefix values. The functionality of these INC and DEC instructions is still available, however, using the ModRM forms of those instructions (opcodes FF/0 and FF/1). See “Single-Byte INC and DEC Instructions in 64-Bit Mode” in Volume 3 for additional information.

2.5.9 MOVSXD Instruction

MOVSXD is a new instruction in 64-bit mode (the legacy ARPL instruction opcode, 63h, is reassigned as the MOVSXD opcode). It reads a fixed-size 32-bit source operand from a register or memory and (if a REX prefix is used with the instruction) sign-extends the value to 64 bits. MOVSXD is analogous to the MOVSX instruction, which sign-extends a byte to a word or a word to a doubleword, depending on the effective operand size. See the instruction reference page for the MOVSXD instruction in Volume 3 for additional information.

2.5.10 Invalid Instructions

Table 2-3 lists instructions that are illegal in 64-bit mode. Table 2-4 on page 35 lists instructions that are invalid in long mode (both compatibility mode and 64-bit mode). Attempted use of these instructions causes an invalid-opcode exception (#UD) to occur.

Table 2-3. Invalid Instructions in 64-Bit Mode

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Opcode (hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAA</td>
<td>37</td>
<td>ASCII Adjust After Addition</td>
</tr>
<tr>
<td>AAD</td>
<td>D5</td>
<td>ASCII Adjust Before Division</td>
</tr>
<tr>
<td>AAM</td>
<td>D4</td>
<td>ASCII Adjust After Multiply</td>
</tr>
<tr>
<td>AAS</td>
<td>3F</td>
<td>ASCII Adjust After Subtraction</td>
</tr>
<tr>
<td>BOUND</td>
<td>62</td>
<td>Check Array Bounds</td>
</tr>
</tbody>
</table>
### Table 2-3. Invalid Instructions in 64-Bit Mode (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Opcode (hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL (far)</td>
<td>9A</td>
<td>Procedure Call Far (absolute)</td>
</tr>
<tr>
<td>DAA</td>
<td>27</td>
<td>Decimal Adjust after Addition</td>
</tr>
<tr>
<td>DAS</td>
<td>2F</td>
<td>Decimal Adjust after Subtraction</td>
</tr>
<tr>
<td>INTO</td>
<td>CE</td>
<td>Interrupt to Overflow Vector</td>
</tr>
<tr>
<td>JMP (far)</td>
<td>EA</td>
<td>Jump Far (absolute)</td>
</tr>
<tr>
<td>LDS</td>
<td>C5</td>
<td>Load DS Segment Register</td>
</tr>
<tr>
<td>LES</td>
<td>C4</td>
<td>Load ES Segment Register</td>
</tr>
<tr>
<td>POP DS</td>
<td>1F</td>
<td>Pop Stack into DS Segment</td>
</tr>
<tr>
<td>POP ES</td>
<td>07</td>
<td>Pop Stack into ES Segment</td>
</tr>
<tr>
<td>POP SS</td>
<td>17</td>
<td>Pop Stack into SS Segment</td>
</tr>
<tr>
<td>POPA, POPAD</td>
<td>61</td>
<td>Pop All to GPR Words or Doublewords</td>
</tr>
<tr>
<td>PUSH CS</td>
<td>0E</td>
<td>Push CS Segment Selector onto Stack</td>
</tr>
<tr>
<td>PUSH DS</td>
<td>1E</td>
<td>Push DS Segment Selector onto Stack</td>
</tr>
<tr>
<td>PUSH ES</td>
<td>06</td>
<td>Push ES Segment Selector onto Stack</td>
</tr>
<tr>
<td>PUSH SS</td>
<td>16</td>
<td>Push SS Segment Selector onto Stack</td>
</tr>
<tr>
<td>PUSHA, PUSHAD</td>
<td>60</td>
<td>Push All GPR Words or Doublewords onto Stack</td>
</tr>
<tr>
<td>Redundant Grp1 (undocumented)</td>
<td>82</td>
<td>Redundant encoding of group1 Eb,lb opcodes</td>
</tr>
<tr>
<td>SALC (undocumented)</td>
<td>D6</td>
<td>Set AL According to CF</td>
</tr>
</tbody>
</table>

### Table 2-4. Invalid Instructions in Long Mode

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Opcode (hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYSENTER</td>
<td>0F 34</td>
<td>System Call</td>
</tr>
<tr>
<td>SYSEXIT</td>
<td>0F 35</td>
<td>System Return</td>
</tr>
</tbody>
</table>
2.5.11 Reassigned Opcodes

Table 2-5 below lists opcodes that are assigned functions in 64-bit mode that differ from their legacy functions.

<table>
<thead>
<tr>
<th>Opcode (hex)</th>
<th>Compatibility and Legacy Modes</th>
<th>64-Bit Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>ARPL—Adjust Requestor Privilege Level</td>
<td>MOVSD—Move Doubleword with Sign Extension</td>
</tr>
<tr>
<td>40–4F</td>
<td>DEC—Decrement by 1 INC—Increment by 1 REX Prefix</td>
<td></td>
</tr>
</tbody>
</table>

Note: Two-byte versions of DEC and INC are still available in 64-bit mode.

2.5.12 FXSAVE and FXRSTOR Instructions

The FXSAVE and FXRSTOR instructions are used to save and restore the entire 128-bit media (XMM), 64-bit media, and x87 instruction-set environment during a context switch. The AMD64 architecture modifies the memory format used by these instructions in order to save and restore the full 64-bit instruction and data pointers, as well as the XMM8–15 registers. Selection of the 32-bit legacy format or the expanded 64-bit format is accomplished by using the corresponding operand size with the FXSAVE and FXRSTOR instructions. When 64-bit software executes an FXSAVE and FXRSTOR with a 32-bit operand size (no operand-size override) the 32-bit legacy format is used. When 64-bit software executes an FXSAVE and FXRSTOR with a 64-bit operand size, the 64-bit format is used.

For more information on the save area formats, see Section 11.4.4, “Saving Media and x87 Execution Unit State,” on page 316.

If the fast-FXSAVE/FXRSTOR (FFXSR) feature is enabled in EFER, FXSAVE and FXRSTOR do not save or restore the XMM0–15 registers when executed in 64-bit mode at CPL 0. The x87 environment and MXCSR are saved whether fast-FXSAVE/FXRSTOR is enabled or not. The fast-FXSAVE/FXRSTOR feature has no effect on FXSAVE/FXRSTOR in non 64-bit mode or when CPL > 0.

Software can use the CPUID instruction to determine whether the fast-FXSAVE/FXRSTOR feature is available (CPUID F8000_0001h_EDX[FFXSR]). For information on using the CPUID instruction to obtain processor feature information, see Section 3.3, “Processor Feature Identification,” on page 64.

2.6 Interrupts and Exceptions

When a processor is running in long mode, an interrupt or exception causes the processor to enter 64-bit mode. All long-mode interrupt handlers must be implemented as 64-bit code. The AMD64 architecture expands the legacy interrupt-processing and exception-processing mechanism to support
handling of interrupts by 64-bit operating systems and applications. The changes are summarized in
the following sections. See “Long-Mode Interrupt Control Transfers” on page 255 for detailed
information on these changes.

2.6.1 Interrupt Descriptor Table

The long-mode interrupt-descriptor table (IDT) must contain 64-bit mode interrupt-gate or trap-gate
descriptors for all interrupts or exceptions that can occur while the processor is running in long mode.
Task gates cannot be used in the long-mode IDT, because control transfers through task gates are not
supported in long mode. In long mode, the IDT index is formed by scaling the interrupt vector by 16.
In legacy protected mode, the IDT is indexed by scaling the interrupt vector by eight.

2.6.2 Stack Frame Pushes

In legacy mode, the size of an IDT entry (16 bits or 32 bits) determines the size of interrupt-stack-
frame pushes, and SS:eSP is pushed only on a CPL change. In long mode, the size of interrupt stack-
frame pushes is fixed at eight bytes, because interrupts are handled in 64-bit mode. Long mode
interrupts also cause SS:RSP to be pushed unconditionally, rather than pushing only on a CPL change.

2.6.3 Stack Switching

Legacy mode provides a mechanism to automatically switch stack frames in response to an interrupt.
In long mode, a slightly modified version of the legacy stack-switching mechanism is implemented,
and an alternative stack-switching mechanism—called the interrupt stack table (IST)—is supported.

Long-Mode Stack Switches. When stacks are switched as part of a long-mode privilege-level
change resulting from an interrupt, the following occurs:
• The target-stack pointer read from the TSS is a 64-bit RSP value.
• The SS register is loaded with a null selector. Setting the new SS selector to null allows nested
control transfers in 64-bit mode to be handled properly. The SS.RPL value is cleared to 0.
• The old SS and RSP are saved on the new stack.

Interrupt Stack Table. In long mode, a new interrupt stack table (IST) mechanism is available as an
alternative to the modified legacy stack-switching mechanism. The IST mechanism unconditionally
switches stacks when it is enabled. It can be enabled for individual interrupt vectors using a field in the
IDT entry. This allows mixing interrupt vectors that use the modified legacy mechanism with vectors
that use the IST mechanism. The IST pointers are stored in the long-mode TSS. The IST mechanism is
only available when long mode is enabled.

2.6.4 IRET Instruction

In compatibility mode, IRET pops SS:eSP off the stack only if there is a CPL change. This allows
legacy applications to run properly in compatibility mode when using the IRET instruction.

In 64-bit mode, IRET unconditionally pops SS:eSP off of the interrupt stack frame, even if the CPL
does not change. This is done because the original interrupt always pushes SS:RSP. Because interrupt
stack-frame pushes are always eight bytes in long mode, an IRET from a long-mode interrupt handler (64-bit code) must pop eight-byte items off the stack. This is accomplished by preceding the IRET with a 64-bit REX operand-size prefix.

In long mode, an IRET can load a null SS selector from the stack under the following conditions:

- The target operating mode is 64-bit mode.
- The target CPL<3.

Allowing IRET to load SS with a null selector under these conditions makes it possible for the processor to unnest interrupts (and far CALLs) in long mode.

2.6.5 Task-Priority Register (CR8)

The AMD64 architecture allows software to define up to 15 external interrupt-priority classes. Priority classes are numbered from 1 to 15, with priority-class 1 being the lowest and priority-class 15 the highest.

A new control register (CR8) is introduced by the AMD64 architecture for managing priority classes. This register, also called the task-priority register (TPR), uses the four low-order bits for specifying a task priority. How external interrupts are organized into these priority classes is implementation dependent. See “External Interrupt Priorities” on page 243 for information on this feature.

2.6.6 New Exception Conditions

The AMD64 architecture defines a number of new conditions that can cause an exception to occur when the processor is running in long mode. Many of the conditions occur when software attempts to use an address that is not in canonical form. See “Vectors” on page 222 for information on the new exception conditions that can occur in long mode.

2.7 Hardware Task Switching

The legacy hardware task-switch mechanism is disabled when the processor is running in long mode. However, long mode requires system software to create data structures for a single task—the long-mode task.

- **TSS Descriptors**—A new TSS-descriptor type, the 64-bit TSS type, is defined for use in long mode. It is the only valid TSS type that can be used in long mode, and it must be loaded into the TR by executing the LTR instruction in 64-bit mode. See “TSS Descriptor” on page 338 for additional information.

- **Task Gates**—Because the legacy task-switch mechanism is not supported in long mode, software cannot use task gates in long mode. Any attempt to transfer control to another task through a task gate causes a general-protection exception (#GP) to occur.

- **Task-State Segment**—A 64-bit task state segment (TSS) is defined for use in long mode. This new TSS format contains 64-bit stack pointers (RSP) for privilege levels 0–2, interrupt-stack-table
(IST) pointers, and the I/O-map base address. See “64-Bit Task State Segment” on page 345 for additional information.

## 2.8 Long-Mode vs. Legacy-Mode Differences

Table 2-6 on page 39 summarizes several major system-programming differences between 64-bit mode and legacy protected mode. The third column indicates whether the difference also applies to compatibility mode. “Differences Between Long Mode and Legacy Mode” in Volume 3 summarizes the application-programming model differences.

### Table 2-6. Differences Between Long Mode and Legacy Mode

<table>
<thead>
<tr>
<th>Subject</th>
<th>64-Bit Mode Difference</th>
<th>Applies To Compatibility Mode?</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86 Modes</td>
<td>Real and virtual-8086 modes not supported</td>
<td>Yes</td>
</tr>
<tr>
<td>Task Switching</td>
<td>Task switching not supported</td>
<td>Yes</td>
</tr>
<tr>
<td>Addressing</td>
<td>64-bit virtual addresses</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>4-level paging structures</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PAE must always be enabled</td>
<td>Yes</td>
</tr>
<tr>
<td>Loaded Segment (Usage during memory reference)</td>
<td>CS, DS, ES, SS segment bases are ignored</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>CS, DS, ES, FS, GS, SS segment limits are ignored</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DS, ES, FS, GS attribute are ignored</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CS, DS, ES, SS Segment prefixes are ignored</td>
<td></td>
</tr>
<tr>
<td>Exception and Interrupt Handling</td>
<td>All pushes are 8 bytes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>IDT entries are expanded to 16 bytes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SS is not changed for stack switch</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SS:RSP is pushed unconditionally</td>
<td></td>
</tr>
<tr>
<td>Call Gates</td>
<td>All pushes are 8 bytes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>16-bit call gates are illegal</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32-bit call gate type is redefined as 64-bit call gate and is expanded to 16 bytes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SS is not changed for stack switch</td>
<td></td>
</tr>
<tr>
<td>System-Descriptor Registers</td>
<td>GDT, IDT, LDT, TR base registers expanded to 64 bits</td>
<td>Yes</td>
</tr>
<tr>
<td>System-Descriptor Table Entries and Pseudo-Descriptors</td>
<td>LGDT and LIDT use expanded 10-byte pseudo-descriptors</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>LLDT and LTR use expanded 16-byte table entries</td>
<td></td>
</tr>
</tbody>
</table>
3 System Resources

The operating system manages the software-execution environment and general system operation through the use of system resources. These resources consist of system registers (control registers and model-specific registers) and system-data structures (memory-management and protection tables). The system-control registers are described in detail in this chapter; many of the features they control are described elsewhere in this volume. The model-specific registers supported by the AMD64 architecture are introduced in this chapter.

Because of their complexity, system-data structures are described in separate chapters. Refer to the following chapters for detailed information on these data structures:

- Descriptors and descriptor tables are described in Section 4.4 “Segmentation Data Structures and Registers,” on page 69.
- Page-translation tables are described in Section 5.2 “Legacy-Mode Page Translation,” on page 124 and Section 5.3 “Long-Mode Page Translation,” on page 132.
- The task-state segment is described in Section 12.2.4 “Legacy Task-State Segment,” on page 341 and Section 12.2.5 “64-Bit Task State Segment,” on page 345.

Not all processor implementations are required to support all possible features. The last section in this chapter addresses processor-feature identification. System software uses the capabilities described in that section to determine which features are supported so that the appropriate service routines are loaded.

3.1 System-Control Registers

The registers that control the AMD64 architecture operating environment include:

- **CR0**—Provides operating-mode controls and some processor-feature controls.
- **CR2**—This register is used by the page-translation mechanism. It is loaded by the processor with the page-fault virtual address when a page-fault exception occurs.
- **CR3**—This register is also used by the page-translation mechanism. It contains the base address of the highest-level page-translation table, and also contains cache controls for the specified table.
- **CR4**—This register contains additional controls for various operating-mode features.
- **CR8**—This new register, accessible in 64-bit mode using the REX prefix, is introduced by the AMD64 architecture. CR8 is used to prioritize external interrupts and is referred to as the task-priority register (TPR).
- **RFLAGS**—This register contains processor-status and processor-control fields. The status and control fields are used primarily in the management of virtual-8086 mode, hardware multitasking, and interrupts.
• **EFER**—This model-specific register contains status and controls for additional features not managed by the CR0 and CR4 registers. Included in this register are the long-mode enable and activation controls introduced by the AMD64 architecture.

Control registers CR1, CR5–CR7, and CR9–CR15 are reserved.

In legacy mode, all control registers and RFLAGS are 32 bits. The EFER register is 64 bits in all modes. The AMD64 architecture expands all 32-bit system-control registers to 64 bits. In 64-bit mode, the MOV CRn instructions read or write all 64 bits of these registers (operand-size prefixes are ignored). In compatibility and legacy modes, control-register writes fill the low 32 bits with data and the high 32 bits with zeros, and control-register reads return only the low 32 bits.

In 64-bit mode, the high 32 bits of CR0 and CR4 are reserved and must be written with zeros. Writing a 1 to any of the high 32 bits results in a general-protection exception, #GP(0). All 64 bits of CR2 are writable. However, the MOV CRn instructions do not check that addresses written to CR2 are within the virtual-address limitations of the processor implementation.

All CR3 bits are writable, except for unimplemented physical address bits, which must be cleared to 0.

The upper 32 bits of RFLAGS are always read as zero by the processor. Attempts to load the upper 32 bits of RFLAGS with anything other than zero are ignored by the processor.

### 3.1.1 CR0 Register

The CR0 register is shown in Figure 3-1 on page 43. The legacy CR0 register is identical to the low 32 bits of this register (CR0 bits 31:0).
Figure 3-1. Control Register 0 (CR0)

The functions of the CR0 control bits are (unless otherwise noted, all bits are read/write):

**Protected-Mode Enable (PE) Bit.** Bit 0. Software enables protected mode by setting PE to 1, and disables protected mode by clearing PE to 0. When the processor is running in protected mode, segment-protection mechanisms are enabled.

See Section 4.9 “Segment-Protection Overview,” on page 97 for information on the segment-protection mechanisms.

**Monitor Coprocessor (MP) Bit.** Bit 1. Software uses the MP bit with the task-switched control bit (CR0.TS) to control whether execution of the WAIT/FWAIT instruction causes a device-not-available exception (#NM) to occur, as follows:

- If both the monitor-coprocessor and task-switched bits are set (CR0.MP=1 and CR0.TS=1), then executing the WAIT/FWAIT instruction causes a device-not-available exception (#NM).
- If either the monitor-coprocessor or task-switched bits are clear (CR0.MP=0 or CR0.TS=0), then executing the WAIT/FWAIT instruction proceeds normally.
Software typically should set MP to 1 if the processor implementation supports x87 instructions. This allows the CR0.TS bit to completely control when the x87-instruction context is saved as a result of a task switch.

**Emulate Coprocessor (EM) Bit.** Bit 2. Software forces all x87 instructions to cause a device-not-available exception (#NM) by setting EM to 1. Likewise, setting EM to 1 forces an invalid-opcode exception (#UD) when an attempt is made to execute any of the 64-bit or 128-bit media instructions except the FXSAVE and FXRSTOR instructions. Attempting to execute these instructions when EM is set results in an #NM exception instead. The exception handlers can emulate these instruction types if desired. Setting the EM bit to 1 does not cause an #NM exception when the WAIT/FWAIT instruction is executed.

**Task Switched (TS) Bit.** Bit 3. When an attempt is made to execute an x87 or media instruction while TS=1, a device-not-available exception (#NM) occurs. Software can use this mechanism—sometimes referred to as “lazy context-switching”—to save the unit contexts before executing the next instruction of those types. As a result, the x87 and media instruction-unit contexts are saved only when necessary as a result of a task switch.

When a hardware task switch occurs, TS is automatically set to 1. System software that implements software task-switching rather than using the hardware task-switch mechanism can still use the TS bit to control x87 and media instruction-unit context saves. In this case, the task-management software uses a MOV CR0 instruction to explicitly set the TS bit to 1 during a task switch. Software can clear the TS bit by either executing the CLTS instruction or by writing to the CR0 register directly. Long-mode system software can use this approach even though the hardware task-switch mechanism is not supported in long mode.

The CR0.MP bit controls whether the WAIT/FWAIT instruction causes an #NM exception when TS=1.

**Extension Type (ET) Bit.** Bit 4, read-only. In some early x86 processors, software set ET to 1 to indicate support of the 387DX math-coprocessor instruction set. This bit is now reserved and forced to 1 by the processor. Software cannot clear this bit to 0.

**Numeric Error (NE) Bit.** Bit 5. Clearing the NE bit to 0 disables internal control of x87 floating-point exceptions and enables external control. When NE is cleared to 0, the IGNNE# input signal controls whether x87 floating-point exceptions are ignored:

- When IGNNE# is 1, x87 floating-point exceptions are ignored.
- When IGNNE# is 0, x87 floating-point exceptions are reported by setting the FERR# input signal to 1. External logic can use the FERR# signal as an external interrupt.

When NE is set to 1, internal control over x87 floating-point exception reporting is enabled and the external reporting mechanism is disabled. It is recommended that software set NE to 1. This enables optimal performance in handling x87 floating-point exceptions.

**Write Protect (WP) Bit.** Bit 16. Read-only pages are protected from supervisor-level writes when the WP bit is set to 1. When WP is cleared to 0, supervisor software can write into read-only pages.
See Section 5.6 “Page-Protection Checks,” on page 149 for information on the page-protection mechanism.

Alignment Mask (AM) Bit. Bit 18. Software enables automatic alignment checking by setting the AM bit to 1 when RFLAGS.AC=1. Alignment checking can be disabled by clearing either AM or RFLAGS.AC to 0. When automatic alignment checking is enabled and CPL=3, a memory reference to an unaligned operand causes an alignment-check exception (#AC).

Not Writethrough (NW) Bit. Bit 29. Ignored. This bit can be set to 1 or cleared to 0, but its value is ignored. The NW bit exists only for legacy purposes.

Cache Disable (CD) Bit. Bit 30. When CD is cleared to 0, the internal caches are enabled. When CD is set to 1, no new data or instructions are brought into the internal caches. However, the processor still accesses the internal caches when CD = 1 under the following situations:

- Reads that hit in an internal cache cause the data to be read from the internal cache that reported the hit.
- Writes that hit in an internal cache cause the cache line that reported the hit to be written back to memory and invalidated in the cache.

Cache misses do not affect the internal caches when CD = 1. Software can prevent cache access by setting CD to 1 and invalidating the caches.

Setting CD to 1 also causes the processor to ignore the page-level cache-control bits (PWT and PCD) when paging is enabled. These bits are located in the page-translation tables and CR3 register. See Section “Page-Level Writethrough (PWT) Bit,” on page 142 and Section “Page-Level Cache Disable (PCD) Bit,” on page 142 for information on page-level cache control.

See Section 7.6 “Memory Caches,” on page 185 for information on the internal caches.

Paging Enable (PG) Bit. Bit 31. Software enables page translation by setting PG to 1, and disables page translation by clearing PG to 0. Page translation cannot be enabled unless the processor is in protected mode (CR0.PE=1). If software attempts to set PG to 1 when PE is cleared to 0, the processor causes a general-protection exception (#GP).

See Section 5.1 “Page Translation Overview,” on page 120 for information on the page-translation mechanism.

Reserved Bits. Bits 28:19, 17, 15:6, and 63:32. When writing the CR0 register, software should set the values of reserved bits to the values found during the previous CR0 read. No attempt should be made to change reserved bits, and software should never rely on the values of reserved bits. In long mode, bits 63:32 are reserved and must be written with zero, otherwise a #GP occurs.

3.1.2 CR2 and CR3 Registers

The CR2 (page-fault linear address) register, shown in Figure 3-2 on page 46 and Figure 3-3 on page 46, and the CR3 (page-translation-table base address) register, shown in Figure 3-4 and Figure 3-5 on page 46, and Figure 3-6 on page 47, are used only by the page-translation mechanism.
Figure 3-2. Control Register 2 (CR2)—Legacy-Mode

Figure 3-3. Control Register 2 (CR2)—Long Mode

See Section “CR2 Register,” on page 233 for a description of the CR2 register.

The CR3 register is used to point to the base address of the highest-level page-translation table.

Figure 3-4. Control Register 3 (CR3)—Legacy-Mode Non-PAE Paging

Figure 3-5. Control Register 3 (CR3)—Legacy-Mode PAE Paging
Figure 3-6. Control Register 3 (CR3)—Long Mode

The legacy CR3 register is described in Section 5.2.1 “CR3 Register,” on page 125, and the long-mode CR3 register is described in Section 5.3.2 “CR3,” on page 132.

3.1.3 CR4 Register

The CR4 register is shown in Figure 3-7. In legacy mode, the CR4 register is identical to the low 32 bits of the register (CR4 bits 31:0). The features controlled by the bits in the CR4 register are model-specific extensions. Except for the performance-counter extensions (PCE) feature, software can use the CPUID instruction to verify that each feature is supported before using that feature. See Section 3.3 “Processor Feature Identification,” on page 64 for information on using the CPUID instruction.
The function of the CR4 control bits are (all bits are read/write):

**Virtual-8086 Mode Extensions (VME).** Bit 0. Setting VME to 1 enables hardware-supported performance enhancements for software running in virtual-8086 mode. Clearing VME to 0 disables this support. The enhancements enabled when VME=1 include:

- Virtualized, maskable, external-interrupt control and notification using the VIF and VIP bits in the RFLAGS register. Virtualizing affects the operation of several instructions that manipulate the RFLAGS.IF bit.
- Selective intercept of software interrupts (INTn instructions) using the interrupt-redirection bitmap in the TSS.

**Protected-Mode Virtual Interrupts (PVI).** Bit 1. Setting PVI to 1 enables support for protected-mode virtual interrupts. Clearing PVI to 0 disables this support. When PVI=1, hardware support of two bits in the RFLAGS register, VIF and VIP, is enabled.

Only the STI and CLI instructions are affected by enabling PVI. Unlike the case when CR0.VME=1, the interrupt-redirection bitmap in the TSS cannot be used for selective INTn interception.

PVI enhancements are also supported in long mode. See Section 8.10 “Virtual Interrupts,” on page 262 for more information on using PVI.

**Time-Stamp Disable (TSD).** Bit 2. The TSD bit allows software to control the privilege level at which the time-stamp counter can be read. When TSD is cleared to 0, software running at any privilege
level can read the time-stamp counter using the RDTSC or RDTSCP instructions. When TSD is set to 1, only software running at privilege-level 0 can execute the RDTSC or RDTSCP instructions.

**Debugging Extensions (DE).** Bit 3. Setting the DE bit to 1 enables the I/O breakpoint capability and enforces treatment of the DR4 and DR5 registers as reserved. Software that accesses DR4 or DR5 when DE=1 causes a invalid opcode exception (#UD).

When the DE bit is cleared to 0, I/O breakpoint capabilities are disabled. Software references to the DR4 and DR5 registers are aliased to the DR6 and DR7 registers, respectively.

**Page-Size Extensions (PSE).** Bit 4. Setting PSE to 1 enables the use of 4-Mbyte physical pages. With PSE=1, the physical-page size is selected between 4 Kbytes and 4 Mbytes using the page-directory entry page-size field (PS). Clearing PSE to 0 disables the use of 4-Mbyte physical pages and restricts all physical pages to 4 Kbytes.

The PSE bit has no effect when physical-address extensions are enabled (CR4.PAE=1). Because long mode requires CR4.PAE=1, the PSE bit is ignored when the processor is running in long mode.

See Section “4-Mbyte Page Translation,” on page 127 for more information on 4-Mbyte page translation.

**Physical-Address Extension (PAE).** Bit 5. Setting PAE to 1 enables the use of physical-address extensions and 2-Mbyte physical pages. Clearing PAE to 0 disables these features.

With PAE=1, the page-translation data structures are expanded from 32 bits to 64 bits, allowing the translation of up to 52-bit physical addresses. Also, the physical-page size is selectable between 4 Kbytes and 2 Mbytes using the page-directory-entry page-size field (PS). Long mode requires PAE to be enabled in order to use the 64-bit page-translation data structures to translate 64-bit virtual addresses to 52-bit physical addresses.

See Section 5.2.3 “PAE Paging,” on page 128 for more information on physical-address extensions.

**Machine-Check Enable (MCE).** Bit 6. Setting MCE to 1 enables the machine-check exception mechanism. Clearing this bit to 0 disables the mechanism. When enabled, a machine-check exception (#MC) occurs when an uncorrectable machine-check error is encountered.

Regardless of whether machine-check exceptions are enabled, the processor records enabled-errors when they occur. Error-reporting is performed by the machine-check error-reporting register banks. Each bank includes a control register for enabling error reporting and a status register for capturing errors. Correctable machine-check errors are also reported, but they do not cause a machine-check exception.

See Chapter 9, “Machine Check Architecture,” for a description of the machine-check mechanism, the registers used, and the types of errors captured by the mechanism.

**Page-Global Enable (PGE).** Bit 7. When page translation is enabled, system-software performance can often be improved by making some page translations global to all tasks and procedures. Setting PGE to 1 enables the global-page mechanism. Clearing this bit to 0 disables the mechanism.
When PGE is enabled, system software can set the global-page (G) bit in the lowest level of the page-translation hierarchy to 1, indicating that the page translation is global. Page translations marked as global are not invalidated in the TLB when the page-translation-table base address (CR3) is updated. When the G bit is cleared, the page translation is not global. All supported physical-page sizes also support the global-page mechanism. See Section 5.5.2 “Global Pages,” on page 146 for information on using the global-page mechanism.

**Performance-Monitoring Counter Enable (PCE).** Bit 8. Setting PCE to 1 allows software running at any privilege level to use the RDPMC instruction. Software uses the RDPMC instruction to read the performance-monitoring counter MSRs, *PerfCtrn. Clearing PCE to 0 allows only the most-privileged software (CPL=0) to use the RDPMC instruction.

**FXSAVE/FXRSTOR Support (OSFXSR).** Bit 9. System software must set the OSFXSR bit to 1 to enable use of the legacy SSE instructions. When this bit is set to 1, it also indicates that system software uses the FXSAVE and FXRSTOR instructions to save and restore the processor state for the x87, 64-bit media, and 128-bit media instructions.

Clearing the OSFXSR bit to 0 indicates that legacy SSE instructions cannot be used. Attempts to use those instructions while this bit is clear result in an invalid-opcode exception (#UD). Software can continue to use the FXSAVE/FXRSTOR instructions for saving and restoring the processor state for the x87 and 64-bit media instructions.

**Unmasked Exception Support (OSXMMEXCPT).** Bit 10. System software must set the OSXMMEXCPT bit to 1 when it supports the SIMD floating-point exception (#XF) for handling of unmasked 256-bit and 128-bit media floating-point errors. Clearing the OSXMMEXCPT bit to 0 indicates the #XF handler is not supported. When OSXMMEXCPT=0, unmasked 128-bit media floating-point exceptions cause an invalid-opcode exception (#UD). See “SIMD Floating-Point Exception Causes” in Volume 1 for more information on unmasked SSE floating-point exceptions.

**FSGSBASE.** Bit 16. System software must set this bit to 1 to enable the execution of the RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE instructions when supported. When enabled, these instructions allow software running in 64-bit mode at any privilege level to read and write the FS.base and GS.base hidden segment register state. See the discussion of segment registers in 64-bit mode in Section 4.5.3 “Segment Registers in 64-Bit Mode,” on page 74. Also see descriptions of the RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE instructions in Volume 3.

**Processor Context Identifier Enable (PCIDE).** Bit 17. Enable support for Process Context Identifiers (PCIDs). System software must set this bit to 1 to enable execution of the INVPCID instruction when supported. Can only be set in long mode (EFER.LMA = 1). See 145 for more information on Process Context Identifiers.

**XSAVE and Extended States (OSXSAVE).** Bit 18. After verifying hardware support for the extended processor state management instructions, operating system software sets this bit to indicate support for the XGETBV, XSAVE and XRSTOR instructions.

Setting this bit also:
• allows the execution of the XGETBV and XSETBV instructions, and
• enables the XSAVE and XRSTOR instructions to save and restore the x87 FPU state (including MMX registers), along with other processor extended states enabled in XCR0.

After initializing the XSAVE/XRSTOR save area, XSAVEOPT (if supported) may be used to save x87 FPU and other enabled extended processor state. For more information on XSAVEOPT, see individual instruction listing in Chapter 2 of Volume 4.

Note that legacy SSE instruction execution must be enabled prior to enabling extended processor state management.

**Supervisor Mode Execution Prevention (SMEP).** Bit 20. Setting this bit enables the supervisor mode execution prevention feature, if supported. This feature prevents the execution of instructions that reside in pages accessible by user-mode software when the processor is in supervisor-mode. See Section 5.6 “Page-Protection Checks,” on page 149 for more information.

**Protection Key Enable (PKE).** Bit 22. Enable support for memory Protection Keys. Also enables support for the RDPKRU and WRPKRU instructions. A MOV to CR4 that changes CR4.PKE from 0 to 1 causes all cached entries in the TLB for the logical processor to be invalidated. (See Section 5.6.6 “Memory Protection Keys (MPK) Bit,” on page 151 for more information on memory protection keys.)

**CR1 and CR5–CR7 Registers.** Control registers CR1, CR5–CR7, and CR9–CR15 are reserved. Attempts by software to use these registers result in an undefined-opcode exception (#UD).

### 3.1.4 Additional Control Registers in 64-Bit-Mode

In 64-bit mode, additional encodings are available to address up to eight additional control registers. The REX.R bit, in a REX prefix, is used to modify the ModRM reg field when that field encodes a control register, as shown in “REX Prefixes” in Volume 3. These additional encodings enable the processor to address CR8–CR15.

One additional control register, CR8, is defined in 64-bit mode for all hardware implementations, as described in “CR8 (Task Priority Register, TPR),” below. Access to the CR9–CR15 registers is implementation-dependent. Any attempt to access an unimplemented register results in an invalid-opcode exception (#UD).

### 3.1.5 CR8 (Task Priority Register, TPR)

The AMD64 architecture introduces a new control register, CR8, defined as the task priority register (TPR). The register is accessible in 64-bit mode using the REX prefix. See Section 8.5.2 “External Interrupt Priorities,” on page 243 for a description of the TPR and how system software can use the TPR for controlling external interrupts.

### 3.1.6 RFLAGS Register

The RFLAGS register contains two different types of information:
• **Control bits** provide system-software controls and directional information for string operations. Some of these bits can have privilege-level restrictions.

• **Status bits** provide information resulting from logical and arithmetic operations. These are written by the processor and can be read by software running at any privilege level.

Figure 3-7 on page 52 shows the format of the RFLAGS register. The legacy EFLAGS register is identical to the low 32 bits of the register shown in Figure 3-7 (RFLAGS bits 31:0). The term *rFLAGS* is used to refer to the 16-bit, 32-bit, or 64-bit flags register, depending on context.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:22</td>
<td>Reserved</td>
<td>Reserved, Read as Zero</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>ID</td>
<td>ID Flag</td>
<td>R/W</td>
</tr>
<tr>
<td>20</td>
<td>VIP</td>
<td>Virtual Interrupt Pending</td>
<td>R/W</td>
</tr>
<tr>
<td>19</td>
<td>VIF</td>
<td>Virtual Interrupt Flag</td>
<td>R/W</td>
</tr>
<tr>
<td>18</td>
<td>AC</td>
<td>Alignment Check</td>
<td>R/W</td>
</tr>
<tr>
<td>17</td>
<td>VM</td>
<td>Virtual-8086 Mode</td>
<td>R/W</td>
</tr>
<tr>
<td>16</td>
<td>RF</td>
<td>Resume Flag</td>
<td>R/W</td>
</tr>
<tr>
<td>15</td>
<td>Reserved</td>
<td>Reserved, Read as Zero</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>NT</td>
<td>Nested Task</td>
<td>R/W</td>
</tr>
<tr>
<td>13:12</td>
<td>IOPL</td>
<td>I/O Privilege Level</td>
<td>R/W</td>
</tr>
<tr>
<td>11</td>
<td>OF</td>
<td>Overflow Flag</td>
<td>R/W</td>
</tr>
<tr>
<td>10</td>
<td>DF</td>
<td>Direction Flag</td>
<td>R/W</td>
</tr>
<tr>
<td>9</td>
<td>IF</td>
<td>Interrupt Flag</td>
<td>R/W</td>
</tr>
<tr>
<td>8</td>
<td>TF</td>
<td>Trap Flag</td>
<td>R/W</td>
</tr>
<tr>
<td>7</td>
<td>SF</td>
<td>Sign Flag</td>
<td>R/W</td>
</tr>
<tr>
<td>6</td>
<td>ZF</td>
<td>Zero Flag</td>
<td>R/W</td>
</tr>
<tr>
<td>5</td>
<td>Reserved</td>
<td>Reserved, Read as Zero</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>AF</td>
<td>Auxiliary Flag</td>
<td>R/W</td>
</tr>
<tr>
<td>3</td>
<td>Reserved</td>
<td>Reserved, Read as Zero</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>PF</td>
<td>Parity Flag</td>
<td>R/W</td>
</tr>
<tr>
<td>1</td>
<td>Reserved</td>
<td>Reserved, Read as One</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>CF</td>
<td>Carry Flag</td>
<td>R/W</td>
</tr>
</tbody>
</table>

Figure 3-7. **RFLAGS Register**
The functions of the RFLAGS control and status bits used by application software are described in “Flags Register” in Volume 1. The functions of RFLAGS system bits are (unless otherwise noted, all bits are read/write):

**Trap Flag (TF) Bit.** Bit 8. Software sets the TF bit to 1 to enable single-step mode during software debug. Clearing this bit to 0 disables single-step mode.

When single-step mode is enabled at the start of an instruction's execution, a debug exception (#DB) occurs immediately after the instruction completes execution. Single stepping is automatically disabled (TF is set to 0) when the #DB exception occurs or when any exception or interrupt occurs.

See Section 13.1.4 “Single Stepping,” on page 368 for information on using the single-step mode during debugging.

**Interrupt Flag (IF) Bit.** Bit 9. Software sets the IF bit to 1 to enable maskable interrupts. Clearing this bit to 0 causes the processor to ignore maskable interrupts. The state of the IF bit does not affect the response of a processor to non-maskable interrupts, software-interrupt instructions, or exceptions.

The ability to modify the IF bit depends on several factors:
- The current privilege-level (CPL)
- The I/O privilege level (RFLAGS.IOPL)
- Whether or not virtual-8086 mode extensions are enabled (CR4.VME=1)
- Whether or not protected-mode virtual interrupts are enabled (CR4.PVI=1)

See Section 8.1.4 “Masking External Interrupts,” on page 221 for information on interrupt masking. See Section 6.2.3 “Accessing the RFLAGS Register,” on page 162 for information on the specific instructions used to modify the IF bit.

**I/O Privilege Level Field (IOPL) Field.** Bits 13:12. The IOPL field specifies the privilege level required to execute I/O address-space instructions (i.e., instructions that address the I/O space rather than memory-mapped I/O, such as IN, OUT, INS, OUTS, etc.). For software to execute these instructions, the current privilege-level (CPL) must be equal to or higher than (lower numerical value than) the privilege specified by IOPL (CPL <= IOPL). If the CPL is lower than (higher numerical value than) that specified by the IOPL (CPL > IOPL), the processor causes a general-protection exception (#GP) when software attempts to execute an I/O instruction. See “Protected-Mode I/O” in Volume 1 for information on how IOPL controls access to address-space I/O.

Virtual-8086 mode uses IOPL to control virtual interrupts and the IF bit when virtual-8086 mode extensions are enabled (CR4.VME=1). The protected-mode virtual-interrupt mechanism (PVI) also uses IOPL to control virtual interrupts and the IF bit when PVI is enabled (CR4.PVI=1). See Section 8.10 “Virtual Interrupts,” on page 262 for information on how IOPL is used by the virtual interrupt mechanism.

**Nested Task (NT) Bit.** Bit 14, IRET reads the NT bit to determine whether the current task is nested within another task. When NT is set to 1, the current task is nested within another task. When NT is cleared to 0, the current task is at the top level (not nested).
The processor sets the NT bit during a task switch resulting from a CALL, interrupt, or exception through a task gate. When an IRET is executed from legacy mode while the NT bit is set, a task switch occurs. See Section 12.3.3 “Task Switches Using Task Gates,” on page 351 for information on switching tasks using task gates, and Section 12.3.4 “Nesting Tasks,” on page 353 for information on task nesting.

**Resume Flag (RF) Bit.** Bit 16. The RF bit, when set to 1, temporarily disables instruction breakpoint reporting to prevent repeated debug exceptions (#DB) from occurring. This allows an instruction which had been inhibited by an instruction-breakpoint debug exception to be restarted by the debug exception handler.

The processor clears the RF bit after every instruction is successfully executed, except when the instruction is:

- An IRET that sets the RF bit.
- JMP, CALL, or INT\(n\) through a task gate.

In both of the above cases, RF is not cleared to 0 until the next instruction successfully executes.

When an exception occurs (or when a string instruction is interrupted), the processor normally sets RF=1 in the RFLAGS image saved on the interrupt stack. However, when a #DB exception occurs as a result of an instruction breakpoint, the processor clears the RF bit to 0 in the interrupt-stack RFLAGS image.

For instruction restart to work properly following an instruction breakpoint, the #DB exception handler must set RF to 1 in the interrupt-stack RFLAGS image. When an IRET is later executed to return to the instruction that caused the instruction-breakpoint #DB exception, the set RF bit (RF=1) is loaded from the interrupt-stack RFLAGS image. RF is not cleared by the processor until the instruction causing the #DB exception successfully executes.

**Virtual-8086 Mode (VM) Bit.** Bit 17. Software sets the VM bit to 1 to enable virtual-8086 mode. Software clears the VM bit to 0 to disable virtual-8086 mode. System software can only change this bit using a task switch or an IRET. It cannot modify the bit using the POPFD instruction.

**Alignment Check (AC) Bit.** Bit 18. Software enables automatic alignment checking by setting the AC bit to 1 when CR0.AM=1. Alignment checking can be disabled by clearing either AC or CR0.AM to 0. When automatic alignment checking is enabled and the current privilege-level (CPL) is 3 (least privileged), a memory reference to an unaligned operand causes an alignment-check exception (#AC).

**Virtual Interrupt (VIF) Bit.** Bit 19. The VIF bit is a virtual image of the RFLAGS.IF bit. It is enabled when either virtual-8086 mode extensions are enabled (CR4.VME=1) or protected-mode virtual interrupts are enabled (CR4.PVI=1), and the RFLAGS.IOPL field is less than 3. When enabled, instructions that ordinarily would modify the IF bit actually modify the VIF bit with no effect on the RFLAGS.IF bit.
System software that supports virtual-8086 mode should enable the VIF bit using CR4.VME. This allows 8086 software to execute instructions that can set and clear the RFLAGS.IF bit without causing an exception. With VIF enabled in virtual-8086 mode, those instructions set and clear the VIF bit instead, giving the appearance to the 8086 software that it is modifying the RFLAGS.IF bit. System software reads the VIF bit to determine whether or not to take the action desired by the 8086 software (enabling or disabling interrupts by setting or clearing the RFLAGS.IF bit).

In long mode, the use of the VIF bit is supported when CR4.PVI=1. See Section 8.10 “Virtual Interrupts,” on page 262 for more information on virtual interrupts.

**Virtual Interrupt Pending (VIP) Bit.** Bit 20. The VIP bit is provided as an extension to both virtual-8086 mode and protected mode. It is used by system software to indicate that an external, maskable interrupt is pending (awaiting) execution by either a virtual-8086 mode or protected-mode interrupt-service routine. Software must enable virtual-8086 mode extensions (CR4.VME=1) or protected-mode virtual interrupts (CR4.PVI=1) before using VIP.

VIP is normally set to 1 by a protected-mode interrupt-service routine that was entered from virtual-8086 mode as a result of an external, maskable interrupt. Before returning to the virtual-8086 mode application, the service routine sets VIP to 1 if EFLAGS.VIF=1. When the virtual-8086 mode application attempts to enable interrupts by clearing EFLAGS.VIF to 0 while VIP=1, a general-protection exception (#GP) occurs. The #GP service routine can then decide whether to allow the virtual-8086 mode service routine to handle the pending external, maskable interrupt. (EFLAGS is specifically referred to in this case because virtual-8086 mode is supported only from legacy mode.)

In long mode, the use of the VIP bit is supported when CR4.PVI=1. See Section 8.10 “Virtual Interrupts,” on page 262 for more information on virtual-8086 mode interrupts and the VIP bit.

**Processor Feature Identification (ID) Bit.** Bit 21. The ability of software to modify this bit indicates that the processor implementation supports the CPUID instruction. See Section 3.3 “Processor Feature Identification,” on page 64 for more information on the CPUID instruction.

### 3.1.7 Extended Feature Enable Register (EFER)

The extended-feature-enable register (EFER) contains control bits that enable additional processor features not controlled by the legacy control registers. The EFER is a model-specific register (MSR) with an address of C000_0080h (see Section 3.2 “Model-Specific Registers (MSRs),” on page 58 for more information on MSRs). It can be read and written only by privileged software. Figure 3-8 on page 56 shows the format of the EFER register.
The defined EFER bits shown in Figure 3-8 above are described below:

**System-Call Extension (SCE) Bit.** Bit 0, read/write. Setting this bit to 1 enables the SYSCALL and SYSRET instructions. Application software can use these instructions for low-latency system calls and returns in a non-segmented (flat) address space. See Section 6.1 “Fast System Call and Return,” on page 158 for additional information.

**Long Mode Enable (LME) Bit.** Bit 8, read/write. Setting this bit to 1 enables the processor to activate long mode. Long mode is not activated until software enables paging some time later. When paging is enabled after LME is set to 1, the processor sets the EFER.LMA bit to 1, indicating that long mode is not only enabled but also active. See Chapter 14, “Processor Initialization and Long Mode Activation,” for more information on activating long mode.

**Long Mode Active (LMA) Bit.** Bit 10, read/write. This bit indicates that long mode is active. The processor sets LMA to 1 when both long mode and paging have been enabled by system software. See Chapter 14, “Processor Initialization and Long Mode Activation,” for more information on activating long mode.

When LMA=1, the processor is running either in compatibility mode or 64-bit mode, depending on the value of the L bit in a code-segment descriptor, as shown in Figure 1-6 on page 12.

---

### Figure 3-8. Extended Feature Enable Register (EFER)

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>63:16</td>
<td>Reserved, MBZ</td>
<td>Reserved, Must be Zero</td>
<td>R/W</td>
</tr>
<tr>
<td>15</td>
<td>TCE</td>
<td>Translation Cache Extension</td>
<td>R/W</td>
</tr>
<tr>
<td>14</td>
<td>FFXSR</td>
<td>Fast FXSAVE/FXRSTOR</td>
<td>R/W</td>
</tr>
<tr>
<td>13</td>
<td>LMSLE</td>
<td>Long Mode Segment Limit Enable</td>
<td>R/W</td>
</tr>
<tr>
<td>12</td>
<td>SVME</td>
<td>Secure Virtual Machine Enable</td>
<td>R/W</td>
</tr>
<tr>
<td>11</td>
<td>NXE</td>
<td>No-Execute Enable</td>
<td>R/W</td>
</tr>
<tr>
<td>10</td>
<td>LMA</td>
<td>Long Mode Active</td>
<td>R/W</td>
</tr>
<tr>
<td>9</td>
<td>Reserved, MBZ</td>
<td>Reserved, Must be Zero</td>
<td>R/W</td>
</tr>
<tr>
<td>8</td>
<td>LME</td>
<td>Long Mode Enable</td>
<td>R/W</td>
</tr>
<tr>
<td>7:1</td>
<td>Reserved, RAZ</td>
<td>Reserved, Read as Zero</td>
<td>R/W</td>
</tr>
<tr>
<td>0</td>
<td>SCE</td>
<td>System Call Extensions</td>
<td>R/W</td>
</tr>
</tbody>
</table>
When LMA=0, the processor is running in legacy mode. In this mode, the processor behaves like a standard 32-bit x86 processor, with none of the new 64-bit features enabled. When writing the EFER register the value of this bit must be preserved. Software must read the EFER register to determine the value of LMA, change any other bits as required and then write the EFER register. An attempt to write a value that differs from the state determined by hardware results in a #GP fault.

**No-Execute Enable (NXE) Bit.** Bit 11, read/write. Setting this bit to 1 enables the no-execute page-protection feature. The feature is disabled when this bit is cleared to 0. See Section “No Execute (NX) Bit,” on page 143 for more information.

Before setting NXE, system software should verify the processor supports the feature by examining the feature flag CPUID Fn8000_0001_EDX[NX]. See Section 3.3 “Processor Feature Identification,” on page 64 for information on using the CPUID instruction.

**Secure Virtual Machine Enable (SVME) Bit.** Bit 12, read/write. Enables the SVM extensions. When this bit is zero, the SVM instructions cause #UD exceptions. EFER.SVME defaults to a reset value of zero. The effect of turning off EFER.SVME while a guest is running is undefined; therefore, the VMM should always prevent guests from writing EFER. SVM extensions can be disabled by setting VM_CR.SVME_DISABLE. For more information, see descriptions of LOCK and SMVE_DISABLE bits in Section 15.30.1 “VM_CR MSR (C001_0114h),” on page 534.

**Long Mode Segment Limit Enable (LMSLE) bit.** Bit 13, read/write. Setting this bit to 1 enables certain limit checks in 64-bit mode. See Section 4.12.2 “Data Limit Checks in 64-bit Mode,” on page 116, for more information on these limit checks.

**Fast FXSAVE/FXRSTOR (FFXSR) Bit.** Bit 14, read/write. Setting this bit to 1 enables the FXSAVE and FXRSTOR instructions to execute faster in 64-bit mode at CPL 0. This is accomplished by not saving or restoring the XMM registers (XMM0-XMM15). The FFXSR bit has no effect when the FXSAVE/FXRSTOR instructions are executed in non 64-bit mode, or when CPL > 0. The FFXSR bit does not affect the save/restore of the legacy x87 floating-point state, or the save/restore of MXCSR.

Before setting FFXSR, system software should verify whether this feature is supported by examining the feature flag CPUID Fn8000_0001_EDX[FFXSR]. See Section 3.3 “Processor Feature Identification,” on page 64 for information on using the CPUID instruction.

**Translation Cache Extension (TCE) Bit.** Bit 15, read/write. Setting this bit to 1 changes how the INVLPG, INVLPGB, and INVPCID instructions operate on TLB entries. When this bit is 0, these instructions remove the target PTE from the TLB as well as all upper-level table entries that are cached in the TLB, whether or not they are associated with the target PTE. When this bit is set, these instructions will remove the target PTE and only those upper-level entries that lead to the target PTE in the page table hierarchy, leaving unrelated upper-level entries intact. This may provide a performance benefit.

Page table management software must be written in a way that takes this behavior into account. Software that was written for a processor that does not cache upper-level table entries may result in...
stale entries being incorrectly used for translations when TCE is enabled. Software that is compatible with TCE mode will operate in either mode.

For software using INVLPGB to broadcast TLB invalidations, the invalidations are controlled by the EFER.TCE value on the processor executing the INVLPGB instruction.

Before setting TCE, system software should verify that this feature is supported by examining the feature flag CPUID Fn8000_0001_ECX[TCE]. See Section 3.3 “Processor Feature Identification,” on page 64 for information on using the CPUID instruction.

### 3.1.8 Extended Control Registers (XCR\textsubscript{n})

Extended control registers (XCR\textsubscript{n}) form a new register space that is available for managing processor architectural features and capabilities. Currently only XCR0 is defined. All other XCR registers are reserved. For more details on the Extended Control Registers, see “Extended Control Registers” in Volume 4, Chapter 1.

### 3.2 Model-Specific Registers (MSRs)

Processor implementations provide model-specific registers (MSRs) for software control over the unique features supported by that implementation. Software reads and writes MSRs using the privileged RDMSR and WRMSR instructions. Implementations of the AMD64 architecture can contain a mixture of two basic MSR types:

- **Legacy MSRs.** The AMD family of processors often share model-specific features with other x86 processor implementations. Where possible, AMD implementations use the same MSRs for the same functions. For example, the memory-typing and debug-extension MSRs are implemented on many AMD and non-AMD processors.

- **AMD model-specific MSRs.** There are many MSRs common to the AMD family of processors but not to legacy x86 processors. Where possible, AMD implementations use the same AMD-specific MSRs for the same functions.

Every model-specific register, as the name implies, is not necessarily implemented by all members of the AMD family of processors. Appendix A, “MSR Cross-Reference,” lists MSR-address ranges currently used by various AMD and other x86 processors.

The AMD64 architecture includes a number of features that are controlled using MSRs. Those MSRs are shown in Figure 3-9. The EFER register—described in Section 3.1.7 “Extended Feature Enable Register (EFER),” on page 55—is also an MSR.
The following sections briefly describe the MSRs in the AMD64 architecture.

3.2.1 System Configuration Register (SYSCFG)

The system-configuration register (SYSCFG) contains control bits for enabling and configuring system bus features. SYSCFG is a model-specific register (MSR) with an address of C001_0010h. Figure 3-10 on page 60 shows the format of the SYSCFG register. Some features are implementation specific, and are described in the *BIOS and Kernel Developer’s Guide (BKDG)* or *Processor Programming Reference Manual* applicable to your product. Implementation-specific features are not shown in Figure 3-10.
The function of the SYSCFG bits are (all bits are read/write unless otherwise noted):

**MtrrFixDramEn Bit.** Bit 18. Setting this bit to 1 enables use of the RdMem and WrMem attributes in the fixed-range MTRR registers. When cleared, these attributes are disabled. The RdMem and WrMem attributes allow system software to define fixed-range IORRs using the fixed-range MTRRs. See Section 7.9.1 “Extended Fixed-Range MTRR Type-Field Encodings,” on page 209 for information on using this feature.

**MtrrFixDramModEn Bit.** Bit 19. Setting this bit to 1 allows software to read and write the RdMem and WrMem bits. When cleared, writes do not modify the RdMem and WrMem bits, and reads return 0. See Section 7.9.1 “Extended Fixed-Range MTRR Type-Field Encodings,” on page 209 for information on using this feature.

**MtrrVarDramEn Bit.** Bit 20. Setting this bit to 1 enables the TOP_MEM register and the variable-range IORRs. These registers are disabled when the bit is cleared to 0. See Section 7.9.2 “IORRs,” on page 210 and Section 7.9.4 “Top of Memory,” on page 212 for information on using these features.

**MtrrTom2En Bit.** Bit 21. Setting this bit to 1 enables the TOP_MEM2 register. The register is disabled when this bit is cleared to 0. See Section 7.9.4 “Top of Memory,” on page 212 for information on using this feature.

**Tom2ForceMemTypeWB.** Bit 22. Setting this bit to 1 forces the default memory type for memory between 4GB and the address specified by TOP_MEM2 to be write back instead of the memory type
defined by MTRRdefType[Type]. For this bit to have any effect, MTRRdefType[E] must be 1. MTRR variable-range settings and PAT can be used to override this memory type.

**MemEncryptionModeEn.** Bit 23. Setting this bit to 1 enables the SME (Section 7.10 “Secure Memory Encryption,” on page 214) and SEV (Section 15.34 “Secure Encrypted Virtualization,” on page 539) memory encryption features. When cleared, these features are disabled. If MSRC001_0015[SmmLock] is set, the MemEncryptionModeEn bit is sticky and cannot be changed from a 1 to a 0.

**SecureNestedPagingEn.** Bit 24. Setting this bit to 1 enables SEV-SNP (Section 15.36 “Secure Nested Paging (SEV-SNP),” on page 553). When cleared, this feature is disabled. Once this bit is set to 1, it cannot be changed. This bit can only be set if MemEncryptionModeEn has been previously or is simultaneously also set to 1.

**VMPLEn.** Bit 25. Setting this bit to 1 enables the VMPL feature (Section 15.36.7 “Virtual Machine Privilege Levels,” on page 557). Software should set this bit to 1 when SecureNestedPagingEn is being set to 1. Once SecureNestedPagingEn is set to 1, VMPLEn cannot be changed.

### 3.2.2 System-Linkage Registers

System-linkage MSRs are used by system software to allow fast control transfers between applications and the operating system. The functions of these registers are:

**STAR, LSTAR, CSTAR, and SFMASK Registers.** These registers are used to provide mode-dependent linkage information for the SYSCALL and SYSRET instructions. STAR is used in legacy modes, LSTAR in 64-bit mode, and CSTAR in compatibility mode. SFMASK is used by the SYSCALL instruction for RFLAGS in long mode.

**FS.base and GS.base Registers.** These registers allow 64-bit base-address values to be specified for the FS and GS segments, for use in 64-bit mode. See Section “FS and GS Registers in 64-Bit Mode,” on page 74 for a description of the special treatment the FS and GS segments receive.

**KernelGSbase Register.** This register is used by the SWAPGS instruction. This instruction exchanges the value located in KernelGSbase with the value located in GS.base.

**SYSENTERx Registers.** The SYSENTER_CS, SYSENTER_ESP, and SYSENTER_EIP registers are used to provide linkage information for the SYSENTER and SYSEXIT instructions. These instructions are only used in legacy mode.

The system-linkage instructions and their use of MSRs are described in Section 6.1 “Fast System Call and Return,” on page 158.

### 3.2.3 Memory-Typing Registers

Memory-typing MSRs are used to characterize, or type, memory. Memory typing allows software to control the cacheability of memory, and determine how accesses to memory are ordered. The memory-typing registers perform the following functions:
MTRRcap Register. This register contains information describing the level of MTRR support provided by the processor.

MTRRdefType Register. This register establishes the default memory type to be used for physical memory that is not specifically characterized using the fixed-range and variable-range MTRRs.

MTRRphysBasen and MTRRphysMaskn Registers. These registers form a register pair that can be used to characterize any address range within the physical-memory space, including all of physical memory. Up to eight address ranges of varying sizes can be characterized using these registers.

MTRRfixn Registers. These registers are used to characterize fixed-size memory ranges in the first 1 Mbyte of physical-memory space.

PAT Register. This register allows memory-type characterization based on the virtual (linear) address. It is an extension to the PCD and PWT memory types supported by the legacy paging mechanism. The PAT mechanism provides the same memory-typing capabilities as the MTRRs, but with the added flexibility provided by the paging mechanism.

TOP_MEM and TOP_MEM2 Registers. These top-of-memory registers allow system software to specify physical addresses ranges as memory-mapped I/O locations.

Refer to Section 7.7 “Memory-Type Range Registers,” on page 194 for more information on using these registers.

3.2.4 Debug-Extension Registers

The debug-extension MSRs provide software-debug capability not available in the legacy debug registers (DR0–DR7). These MSRs allow single stepping and recording of control transfers to take place. The debug-extension registers perform the following functions:

DebugCtl Register. This MSR register provides control over control-transfer recording and single stepping, and external-breakpoint reporting and trace messages.

LastBranchx and LastIntx Registers. The four registers, LastBranchToIP, LastBranchFromIP, LastIntToIP, and LastIntFromIP, are all used to record the source and target of control transfers when branch recording is enabled.

Refer to Section 13.1.6 “Control-Transfer Breakpoint Features,” on page 368 for more information on using these debug registers.

3.2.5 Performance-Monitoring Registers

The time-stamp counter and performance-monitoring registers are useful in identifying performance bottlenecks. The number of performance counters can vary based on the implementation. These registers perform the following functions:

TSC Register. This register is used to count processor-clock cycles. It can be read using the RDMSR instruction, or it can be read using the either of the read time-stamp counter instructions, RDTSC or
RDTSCP. System software can make RDTSC or RDTSCP available for use by non-privileged software by clearing the time-stamp disable bit (CR4.TSD) to 0.

*PerfEvtSel Registers.* These registers are used to specify the events counted by the corresponding performance counter, and to control other aspects of its operation.

*PerfCtrn Registers.* These registers are performance counters that hold a count of processor, northbridge, or L2 cache events or the duration of events, under the control of the corresponding *PerfEvtSel* register. Each *PerfCtrn* register can be read using the RDMSR instruction, or they can be read using the *read performance-monitor counter* instruction, RDPMC. System software can make RDPMC available for use by non-privileged software by setting the performance-monitor counter enable bit (CR4.PCE) to 1.

Refer to Section 13.2.3 “Using Performance Counters,” on page 377 for more information on using these registers.

### 3.2.6 Machine-Check Registers

The machine-check registers control the detection and reporting of hardware machine-check errors. The types of errors that can be reported include cache-access errors, load-data and store-data errors, bus-parity errors, and ECC errors. Two types of machine-check MSRs are shown in Figure 3-9 on page 59.

The first type is global machine-check registers, which perform the following functions:

**MCG_CAP Register.** This register identifies the machine-check capabilities supported by the processor.

**MCG_CTL Register.** This register provides global control over machine-check-error reporting.

**MCG_STATUS Register.** This register reports global status on detected machine-check errors.

The second type is error-reporting register banks, which report on machine-check errors associated with a specific processor unit (or group of processor units). There can be different numbers of register banks for each processor implementation, and each bank is numbered from 0 to \(i\). The registers in each bank perform the following functions:

**MC\(_i\)_CTL Registers.** These registers control error-reporting.

**MC\(_i\)_STATUS Registers.** These registers report machine-check errors.

**MC\(_i\)_ADDR Registers.** These registers report the machine-check error address.

**MC\(_i\)_MISC Registers.** These registers report miscellaneous-error information.

Refer to Section 9.5 “Using MCA Features,” on page 286 for more information on using these registers.
3.2.7 Other MSRs

XSS is a supported MSR although currently there are no features implemented in this MSR.

3.3 Processor Feature Identification

The CPUID instruction provides information about the processor implementation and its capabilities. Software operating at any privilege level can execute the CPUID instruction to collect this information. Software can utilize this information to optimize performance.

The CPUID instruction supports multiple functions, each providing specific information about the processor implementation, including the vendor, model number, revision (stepping), features, cache organization, and name. The multifunction approach allows the CPUID instruction to return a detailed picture of the processor implementation and its capabilities—more detailed information than could be returned by a single function. This flexibility also allows for the addition of new CPUID functions in future processor generations.

The desired function number is loaded into the EAX register before executing the CPUID instruction. CPUID functions are divided into two types:

- **Standard functions** return information about features common to all x86 implementations, including the earliest features offered in the x86 architecture, as well as information about the presence of features such as support for the AVX and FMA instruction subsets. Standard function numbers are in the range 0000_0000h–0000_FFFFh.
- **Extended functions** return information about AMD-specific features such as long mode and the presence of features such as support for the FMA4 and XOP instruction subsets. Extended function numbers are in the range 8000_0000h–8000_FFFFh.

Feature information is returned in the EAX, EBX, ECX, and EDX registers. Some functions accept a second input parameter passed to the instruction in the ECX register.

In this and the other three volumes of this *Programmer’s Manual*, the notation `CPUID FnXXXX_XXXX_RRR[FieldName]_xYY` is used to represent the input parameters and return value that corresponds to a particular processor capability or feature.

In this notation, `XXXX_XXXX` represents the 32-bit value to be placed in the EAX register prior to executing the CPUID instruction. This value is the function number. `RRR` is either EAX, EBX, ECX, or EDX and represents the register to be examined after the execution of the instruction. If the contents of the entire 32-bit register provides the capability information, the notation `[FieldName]` is omitted, otherwise this provides the name of the field within the return value that represents the capability or feature.

When the field is a single bit, this is called a feature flag. Normally, if a feature flag bit is set, the corresponding processor feature is supported and if it is cleared, the feature is not supported. The optional input parameter passed to the CPUID instruction in the ECX register is represented by the
notation \_xYY appended after the return value notation. If a CPUID function does not accept this optional input parameter, this notation is omitted.

For more specific information on the CPUID instruction, see the instruction reference page in Volume 3. For a description of all feature flags related to instruction subset support, see Volume 3, Appendix D, "Instruction Subsets and CPUID Feature Flags." For a comprehensive list of all processor capabilities and feature flags, see Volume 3, Appendix E, "Obtaining Processor Information Via the CPUID Instruction."
4 Segmented Virtual Memory

The legacy x86 architecture supports a segment-translation mechanism that allows system software to relocate and isolate instructions and data anywhere in the virtual-memory space. A segment is a contiguous block of memory within the linear address space. The size and location of a segment within the linear address space is arbitrary. Instructions and data can be assigned to one or more memory segments, each with its own protection characteristics. The processor hardware enforces the rules dictating whether one segment can access another segment.

The segmentation mechanism provides ten segment registers, each of which defines a single segment. Six of these registers (CS, DS, ES, FS, GS, and SS) define user segments. User segments hold software, data, and the stack and can be used by both application software and system software. The remaining four segment registers (GDT, LDT, IDT, and TR) define system segments. System segments contain data structures initialized and used only by system software. Segment registers contain a base address pointing to the starting location of a segment, a limit defining the segment size, and attributes defining the segment-protection characteristics.

Although segmentation provides a great deal of flexibility in relocating and protecting software and data, it is often more efficient to handle memory isolation and relocation with a combination of software and hardware paging support. For this reason, most modern system software bypasses the segmentation features. However, segmentation cannot be completely disabled, and an understanding of the segmentation mechanism is important to implementing long-mode system software.

In long mode, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode:

- In compatibility mode, segmentation functions just as it does in legacy mode, using legacy 16-bit or 32-bit protected mode semantics.
- 64-bit mode, segmentation is disabled, creating a flat 64-bit virtual-address space. As will be seen, certain functions of some segment registers, particularly the system-segment registers, continue to be used in 64-bit mode.

4.1 Real Mode Segmentation

After reset or power-up, the processor always initially enters real mode. Protected modes are entered from real mode.

As noted in “Real Addressing” on page 10, real mode (real-address mode), provides a physical-memory space of 1 Mbyte. In this mode, a 20-bit physical address is determined by shifting a 16-bit segment selector to the left four bits and adding the 16-bit effective address.

Each 64K segment (CS, DS, ES, FS, GS, SS) is aligned on 16-byte boundaries. The segment base is the lowest address in a given segment, and is equal to the segment selector * 16. The POP and MOV instructions can be used to load a (possibly) new segment selector into one of the segment registers.
When this occurs, the selector is updated and the segment base is set to selector * 16. The segment limit and segment attributes are unchanged, but are normally 64K (the maximum allowable limit) and read/write data, respectively.

On FAR transfers, CS (code segment) selector is updated to the new value, and the CS segment base is set to selector * 16. The CS segment limit and attributes are unchanged, but are usually 64K and read/write, respectively.

If the interrupt descriptor table (IDT) is used to find the real mode IDT see “Real-Mode Interrupt Control Transfers” on page 244.

The GDT, LDT, and TSS (see below) are not used in real mode.

### 4.2 Virtual-8086 Mode Segmentation

Virtual-8086 mode supports 16-bit real mode programs running under protected mode (see below). It uses a simple form of memory segmentation, optional paging, and limited protection checking.

Programs running in virtual-8086 mode can access up to 1MB of memory space.

As with real mode segmentation, each 64K segment (CS, DS, ES, FS, GS, SS) is aligned on 16-byte boundaries. The segment base is the lowest address in a given segment, and is equal to the segment selector * 16. The POP and MOV instructions work exactly as in real mode and can be used to load a (possibly) new segment selector into one of the segment registers. When this occurs, the selector is updated and the segment base is set to selector * 16. The segment limit and segment attributes are unchanged, but are normally 64K (the maximum allowable limit) and read/write data, respectively.

FAR transfers, with the exception of interrupts and exceptions, operate as in real mode. On FAR transfers, the CS (code segment) selector is updated to the new value, and the CS segment base is set to selector * 16. The CS segment limit and attributes are unchanged, but are usually 64K and read/write, respectively. Interrupts and exceptions switch the processor to protected mode. (See Chapter 8, “Exceptions and Interrupts” for more information.)

### 4.3 Protected Mode Segmented-Memory Models

System software can use the segmentation mechanism to support one of two basic segmented-memory models: a flat-memory model or a multi-segmented model. These segmentation models are supported in legacy mode and in compatibility mode. Each type of model is described in the following sections.

#### 4.3.1 Multi-Segmented Model

In the multi-segmented memory model, each segment register can reference a unique base address with a unique segment size. Segments can be as small as a single byte or as large as 4 Gbytes. When page translation is used, multiple segments can be mapped to a single page and multiple pages can be mapped to a single segment. Figure 1-1 on page 6 shows an example of the multi-segmented model.
The multi-segmented memory model provides the greatest level of flexibility for system software using the segmentation mechanism.

Compatibility mode allows the multi-segmented model to be used in support of legacy software. However, in compatibility mode, the multi-segmented memory model is restricted to the first 4 Gbytes of virtual-memory space. Access to virtual memory above 4 Gbytes requires the use of 64-bit mode, which does not support segmentation.

4.3.2 Flat-Memory Model

The flat-memory model is the simplest form of segmentation to implement. Although segmentation cannot be disabled, the flat-memory model allows system software to bypass most of the segmentation mechanism. In the flat-memory model, all segment-base addresses have a value of 0 and the segment limits are fixed at 4 Gbytes. Clearing the segment-base value to 0 effectively disables segment translation, resulting in a single segment spanning the entire virtual-address space. All segment descriptors reference this single, flat segment. Figure 1-2 on page 7 shows an example of the flat-memory model.

4.3.3 Segmentation in 64-Bit Mode

In 64-bit mode, segmentation is disabled. The segment-base value is ignored and treated as 0 by the segmentation hardware. Likewise, segment limits and most attributes are ignored. There are a few exceptions. The CS-segment DPL, D, and L attributes are used (respectively) to establish the privilege level for a program, the default operand size, and whether the program is running in 64-bit mode or compatibility mode. The FS and GS segments can be used as additional base registers in address calculations, and those segments can have non-zero base-address values. This facilitates addressing thread-local data and certain system-software data structures. See “FS and GS Registers in 64-Bit Mode” on page 74 for details about the FS and GS segments in 64-bit mode. The system-segment registers are always used in 64-bit mode.

4.4 Segmentation Data Structures and Registers

Figure 4-1 on page 70 shows the following data structures used by the segmentation mechanism:

- **Segment Descriptors**—As the name implies, a segment descriptor describes a segment, including its location in virtual-address space, its size, protection characteristics, and other attributes.

- **Descriptor Tables**—Segment descriptors are stored in memory in one of three tables. The global-descriptor table (GDT) holds segment descriptors that can be shared among all tasks. Multiple local-descriptor tables (LDT) can be defined to hold descriptors that are used by specific tasks and are not shared globally. The interrupt-descriptor table (IDT) holds gate descriptors that are used to access the segments where interrupt handlers are located.

- **Task-State Segment**—A task-state segment (TSS) is a special type of system segment that contains task-state information and data structures for each task. For example, a TSS holds a copy of the GPRs and EFLAGS register when a task is suspended. A TSS also holds the pointers to privileged-
software stacks. The TSS and task-switch mechanism are described in Chapter 12, “Task Management.”

- **Segment Selectors**—Descriptors are selected for use from the descriptor tables using a segment selector. A segment selector contains an index into either the GDT or LDT. The IDT is indexed using an interrupt vector, as described in “Legacy Protected-Mode Interrupt Control Transfers” on page 245, and in “Long-Mode Interrupt Control Transfers” on page 255.

![Segmentation Data Structures](image)

**Figure 4-1. Segmentation Data Structures**

Figure 4-2 on page 71 shows the registers used by the segmentation mechanism. The registers have the following relationship to the data structures:

- **Segment Registers**—The six segment registers (CS, DS, ES, FS, GS, and SS) are used to point to the user segments. A segment selector selects a descriptor when it is loaded into one of the segment registers. This causes the processor to automatically load the selected descriptor into a software-invisible portion of the segment register.

- **Descriptor-Table Registers**—The three descriptor-table registers (GDTR, LDTR, and IDTR) are used to point to the system segments. The descriptor-table registers identify the virtual-memory location and size of the descriptor tables.

- **Task Register (TR)**—Describes the location and limit of the current task state segment (TSS).
4.5 Segment Selectors and Registers

4.5.1 Segment Selectors

Segment selectors are pointers to specific entries in the global and local descriptor tables. Figure 4-3 shows the segment selector format.

<table>
<thead>
<tr>
<th>Bits</th>
<th>Mnemonic</th>
<th>Description</th>
<th>R/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:3</td>
<td>SI</td>
<td>Selector Index</td>
<td>R/W</td>
</tr>
<tr>
<td>2</td>
<td>TI</td>
<td>Table Indicator</td>
<td>R/W</td>
</tr>
<tr>
<td>1:0</td>
<td>RPL</td>
<td>Requestor Privilege Level</td>
<td>R/W</td>
</tr>
</tbody>
</table>

**Figure 4-3. Segment Selector**

The selector format consists of the following fields:

A fourth system-segment register, the TR, points to the TSS. The data structures and registers associated with task-state segments are described in “Task-Management Resources” on page 336.
Selector Index Field. Bits 15:3. The selector-index field specifies an entry in the descriptor table. Descriptor-table entries are eight bytes long, so the selector index is scaled by 8 to form a byte offset into the descriptor table. The offset is then added to either the global or local descriptor-table base address (as indicated by the table-index bit) to form the descriptor-entry address in virtual-address space.

Some descriptor entries in long mode are 16 bytes long rather than 8 bytes (see “Legacy Segment Descriptors” on page 82 for more information on long-mode descriptor-table entries). These expanded descriptors consume two entries in the descriptor table. Long mode, however, continues to scale the selector index by eight to form the descriptor-table offset. It is the responsibility of system software to assign selectors such that they correctly point to the start of an expanded entry.

Table Indicator (TI) Bit. Bit 2. The TI bit indicates which table holds the descriptor referenced by the selector index. When TI=0 the GDT is used and when TI=1 the LDT is used. The descriptor-table base address is read from the appropriate descriptor-table register and added to the scaled selector index as described above.

Requestor Privilege-Level (RPL) Field. Bits 1:0. The RPL represents the privilege level (CPL) the processor is operating under at the time the selector is created.

RPL is used in segment privilege-checks to prevent software running at lesser privilege levels from accessing privileged data. See “Data-Access Privilege Checks” on page 99 and “Control-Transfer Privilege Checks” on page 102 for more information on segment privilege-checks.

Null Selector. Null selectors have a selector index of 0 and TI=0, corresponding to the first entry in the GDT. However, null selectors do not reference the first GDT entry but are instead used to invalidate unused segment registers. A general-protection exception (#GP) occurs if a reference is made to use a segment register containing a null selector in non-64-bit mode. By initializing unused segment registers with null selectors software can trap references to unused segments.

Null selectors can only be loaded into the DS, ES, FS and GS data-segment registers, and into the LDTR descriptor-table register. A #GP occurs if software attempts to load the CS register with a null selector or if software attempts to load the SS register with a null selector in non 64-bit mode or at CPL 3.

4.5.2 Segment Registers

Six 16-bit segment registers are provided for referencing up to six segments at one time. All software tasks require segment selectors to be loaded in the CS and SS registers. Use of the DS, ES, FS, and GS segments is optional, but nearly all software accesses data and therefore requires a selector in the DS register. Table 4-1 on page 73 lists the supported segment registers and their functions.
The processor maintains a *hidden portion* of the segment register in addition to the selector value loaded by software. This hidden portion contains the values found in the descriptor-table entry referenced by the segment selector. The processor loads the descriptor-table entry into the hidden portion when the segment register is loaded. By keeping the corresponding descriptor-table entry in hardware, performance is optimized for the majority of memory references.

Figure 4-4 shows the format of the visible and hidden portions of the segment register. Except for the FS and GS segment base, software cannot directly read or write the hidden portion (shown as gray-shaded boxes in Figure 4-4).

### Table 4-1. Segment Registers

<table>
<thead>
<tr>
<th>Segment Register</th>
<th>Encoding</th>
<th>Segment Register Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES</td>
<td>/0</td>
<td>References optional data-segment descriptor entry</td>
</tr>
<tr>
<td>CS</td>
<td>/1</td>
<td>References code-segment descriptor entry</td>
</tr>
<tr>
<td>SS</td>
<td>/2</td>
<td>References stack segment descriptor entry</td>
</tr>
<tr>
<td>DS</td>
<td>/3</td>
<td>References default data-segment descriptor entry</td>
</tr>
<tr>
<td>FS</td>
<td>/4</td>
<td>References optional data-segment descriptor entry</td>
</tr>
<tr>
<td>GS</td>
<td>/5</td>
<td>References optional data-segment descriptor entry</td>
</tr>
</tbody>
</table>

**CS Register.** The CS register contains the segment selector referencing the current code-segment descriptor entry. All instruction fetches reference the CS descriptor. When a new selector is loaded into the CS register, the current-privilege level (CPL) of the processor is set to that of the CS-segment descriptor-privilege level (DPL).

**Data-Segment Registers.** The DS register contains the segment selector referencing the default data-segment descriptor entry. The SS register contains the stack-segment selector. The ES, FS, and GS registers are optionally loaded with segment selectors referencing other data segments. Data accesses default to referencing the DS descriptor except in the following two cases:
• The ES descriptor is referenced for string-instruction destinations.
• The SS descriptor is referenced for stack operations.

4.5.3 Segment Registers in 64-Bit Mode

CS Register in 64-Bit Mode. In 64-bit mode, most of the hidden portion of the CS register is ignored. Only the L (long), D (default operation size), and DPL (descriptor privilege-level) attributes are recognized by 64-bit mode. Address calculations assume a CS.base value of 0. CS references do not check the CS.limit value, but instead check that the effective address is in canonical form.

DS, ES, and SS Registers in 64-Bit Mode. In 64-bit mode, the contents of the ES, DS, and SS segment registers are ignored. All fields (base, limit, and attribute) in the hidden portion of the segment registers are ignored.

Address calculations in 64-bit mode that reference the ES, DS, or SS segments are treated as if the segment base is 0. Instead of performing limit checks, the processor checks that all virtual-address references are in canonical form.

Neither enabling and activating long mode nor switching between 64-bit and compatibility modes changes the contents of the visible or hidden portions of the segment registers. These registers remain unchanged during 64-bit mode execution unless explicit segment loads are performed.

FS and GS Registers in 64-Bit Mode. Unlike the CS, DS, ES, and SS segments, the FS and GS segment overrides can be used in 64-bit mode. When FS and GS segment overrides are used in 64-bit mode, their respective base addresses are used in the effective-address (EA) calculation. The complete EA calculation then becomes (FS or GS).base + base + (scale * index) + displacement. The FS.base and GS.base values are also expanded to the full 64-bit virtual-address size, as shown in Figure 4-5. The resulting EA calculation is allowed to wrap across positive and negative addresses.

In 64-bit mode, FS-segment and GS-segment overrides are not checked for limit or attributes. Instead, the processor checks that all virtual-address references are in canonical form.

Figure 4-5. FS and GS Segment-Register Format—64-Bit Mode

In 64-bit mode, FS-segment and GS-segment overrides are not checked for limit or attributes. Instead, the processor checks that all virtual-address references are in canonical form.
Segment register-load instructions (MOV to Sreg and POP Sreg) load only a 32-bit base-address value into the hidden portion of the FS and GS segment registers. The base-address bits above the low 32 bits are cleared to 0 as a result of a segment-register load. When a null selector is loaded into FS or GS, the contents of the corresponding hidden descriptor register are not altered.

There are two methods to update the contents of the FS.base and GS.base hidden descriptor fields. The first is available exclusively to privileged software (CPL = 0). The FS.base and GS.base hidden descriptor-register fields are mapped to MSRs. Privileged software can load a 64-bit base address in canonical form into FS.base or GS.base using a single WRMSR instruction. The FS.base MSR address is C000_0100h while the GS.base MSR address is C000_0101h.

The second method of updating the FS and GS base fields is available to software running at any privilege level (when supported by the implementation and enabled by setting CR4[FSGSBASE]). The WRFSBASE and WRGSBASE instructions copy the contents of a GPR to the FS.base and GS.base fields respectively. When the operand size is 32 bits, the upper doubleword of the base is cleared. WRFSBASE and WRGSBASE are only supported in 64-bit mode.

The addresses written into the expanded FS.base and GS.base registers must be in canonical form. Any instruction that attempts to write a non-canonical address to these registers causes a general-protection exception (#GP) to occur.

When in compatibility mode, the FS and GS overrides operate as defined by the legacy x86 architecture regardless of the value loaded into the high 32 bits of the hidden descriptor-register base-address field. Compatibility mode ignores the high 32 bits when calculating an effective address.

4.6 Descriptor Tables

Descriptor tables are used by the segmentation mechanism when protected mode is enabled (CR0.PE=1). These tables hold descriptor entries that describe the location, size, and privilege attributes of a segment. All memory references in protected mode access a descriptor-table entry.

As previously mentioned, there are three types of descriptor tables supported by the x86 segmentation mechanism:

- Global descriptor table (GDT)
- Local descriptor table (LDT)
- Interrupt descriptor table (IDT)

Software establishes the location of a descriptor table in memory by initializing its corresponding descriptor-table register. The descriptor-table registers and the descriptor tables are described in the following sections.

4.6.1 Global Descriptor Table

Protected-mode system software must create a global descriptor table (GDT). The GDT contains code-segment and data-segment descriptor entries (user segments) for segments that can be shared by all
tasks. In addition to the user segments, the GDT can also hold gate descriptors and other system-
segment descriptors. System software can store the GDT anywhere in memory and should protect the
segment containing the GDT from non-privileged software.

Segment selectors point to the GDT when the table-index (TI) bit in the selector is cleared to 0. The
selector index portion of the segment selector references a specific entry in the GDT. Figure 4-6 on
page 76 shows how the segment selector indexes into the GDT. One special form of a segment selector
is the null selector. A null selector points to the first entry in the GDT (the selector index is 0 and
TI=0). However, null selectors do not reference memory, so the first GDT entry cannot be used to
describe a segment (see “Null Selector” on page 72 for information on using the null selector). The
first usable GDT entry is referenced with a selector index of 1.

Figure 4-6. Global and Local Descriptor-Table Access

4.6.2 Global Descriptor-Table Register

The global descriptor-table register (GDTR) points to the location of the GDT in memory and defines
its size. This register is loaded from memory using the LGDT instruction (see “LGDT and LIDT
Instructions” on page 164). Figure 4-7 shows the format of the GDTR in legacy mode and
compatibility mode.
Figure 4-7. GDTR and IDTR Format—Legacy Modes

Figure 4-8 on page 77 shows the format of the GDTR in 64-bit mode.

Figure 4-8. GDTR and IDTR Format—Long Mode

The GDTR contains two fields:

**Limit.** 2 bytes. These bits define the 16-bit limit, or size, of the GDT in bytes. The limit value is added to the base address to yield the ending byte address of the GDT. A general-protection exception (#GP) occurs if software attempts to access a descriptor beyond the GDT limit.

The offsets into the descriptor tables are not extended by the AMD64 architecture in support of long mode. Therefore, the GDTR and IDTR limit-field sizes are unchanged from the legacy sizes. The processor does check the limits in long mode during GDT and IDT accesses.

**Base Address.** 8 bytes. The base-address field holds the starting byte address of the GDT in virtual-memory space. The GDT can be located at any byte address in virtual memory, but system software should align the GDT on a quadword boundary to avoid the potential performance penalties associated with accessing unaligned data.

The AMD64 architecture increases the base-address field of the GDTR to 64 bits so that system software running in long mode can locate the GDT anywhere in the 64-bit virtual-address space. The processor ignores the high-order 4 bytes of base address when running in legacy mode.

4.6.3 Local Descriptor Table

Protected-mode system software can optionally create a local descriptor table (LDT) to hold segment descriptors belonging to a single task or even multiple tasks. The LDT typically contains code-
segment and data-segment descriptors as well as gate descriptors referenced by the specified task. Like the GDT, system software can store the LDT anywhere in memory and should protect the segment containing the LDT from non-privileged software.

Segment selectors point to the LDT when the table-index bit (TI) in the selector is set to 1. The selector index portion of the segment selector references a specific entry in the LDT (see Figure 4-6 on page 76). Unlike the GDT, however, a selector index of 0 references the first entry in the LDT (when TI=1, the selector is not a null selector).

LDTs are described by system-segment descriptor entries located in the GDT, and a GDT can contain multiple LDT descriptors. The LDT system-segment descriptor defines the location, size, and privilege rights for the LDT. Figure 4-9 on page 78 shows the relationship between the LDT and GDT data structures.

Loading a null selector into the LDTR is useful if software does not use an LDT. This causes a #GP if an erroneous reference is made to the LDT.

![Figure 4-9. Relationship between the LDT and GDT](image)

### 4.6.4 Local Descriptor-Table Register

The local descriptor-table register (LDTR) points to the location of the LDT in memory, defines its size, and specifies its attributes. The LDTR has two portions. A visible portion holds the LDT selector, and a hidden portion holds the LDT descriptor. When the LDT selector is loaded into the LDTR, the processor automatically loads the LDT descriptor from the GDT into the hidden portion of the LDTR. The LDTR is loaded in one of two ways:

- Using the LLDT instruction (see “LLDT and LTR Instructions” on page 164).
• Performing a task switch (see “Switching Tasks” on page 349).

Figure 4-10 on page 79 shows the format of the LDTR in legacy mode.

Figure 4-10. LDTR Format—Legacy Mode

The LDTR contains four fields:

**LDT Selector.** 2 bytes. These bits are loaded explicitly from the TSS during a task switch, or by using the LLDT instruction. The LDT selector must point to an LDT system-segment descriptor entry in the GDT. If it does not, a general-protection exception (#GP) occurs.

The following three fields are loaded automatically from the LDT descriptor in the GDT as a result of loading the LDT selector. The register fields are shown as shaded boxes in Figure 4-10 and Figure 4-11.

Figure 4-11. LDTR Format—Long Mode

The LDTR contains four fields:

**LDT Selector.** 2 bytes. These bits are loaded explicitly from the TSS during a task switch, or by using the LLDT instruction. The LDT selector must point to an LDT system-segment descriptor entry in the GDT. If it does not, a general-protection exception (#GP) occurs.

The following three fields are loaded automatically from the LDT descriptor in the GDT as a result of loading the LDT selector. The register fields are shown as shaded boxes in Figure 4-10 and Figure 4-11.
Base Address. The base-address field holds the starting byte address of the LDT in virtual-memory space. Like the GDT, the LDT can be located anywhere in system memory, but software should align the LDT on a quadword boundary to avoid performance penalties associated with accessing unaligned data.

The AMD64 architecture expands the base-address field of the LDTR to 64 bits so that system software running in long mode can locate an LDT anywhere in the 64-bit virtual-address space. The processor ignores the high-order 32 base-address bits when running in legacy mode. Because the LDTR is loaded from the GDT, the system-segment descriptor format (LDTs are system segments) has been expanded by the AMD64 architecture in support of 64-bit mode. See “Long Mode Descriptor Summary” on page 96 for more information on this expanded format. The high-order base-address bits are only loaded from 64-bit mode using the LLDT instruction (see “LLDT and LTR Instructions” on page 164 for more information on this instruction).

Limit. This field defines the limit, or size, of the LDT in bytes. The LDT limit as stored in the LDTR is 32 bits. When the LDT limit is loaded from the GDT descriptor entry, the 20-bit limit field in the descriptor is expanded to 32 bits and scaled based on the value of the descriptor granularity (G) bit. For details on the limit biasing and granularity, see “Granularity (G) Bit” on page 83.

If an attempt is made to access a descriptor beyond the LDT limit, a general-protection exception (#GP) occurs.

The offsets into the descriptor tables are not extended by the AMD64 architecture in support of long mode. Therefore, the LDTR limit-field size is unchanged from the legacy size. The processor does check the LDT limit in long mode during LDT accesses.

Attributes. This field holds the descriptor attributes, such as privilege rights, segment presence and segment granularity.

4.6.5 Interrupt Descriptor Table

The final type of descriptor table is the interrupt descriptor table (IDT). Multiple IDTs can be maintained by system software. System software selects a specific IDT by loading the interrupt descriptor table register (IDTR) with a pointer to the IDT. As with the GDT and LDT, system software can store the IDT anywhere in memory and should protect the segment containing the IDT from non-privileged software.

The IDT can contain only the following types of gate descriptors:

- Interrupt gates
- Trap gates
- Task gates.

The use of gate descriptors by the interrupt mechanism is described in Chapter 8, “Exceptions and Interrupts.” A general-protection exception (#GP) occurs if the IDT descriptor referenced by an interrupt or exception is not one of the types listed above.
IDT entries are selected using the interrupt vector number rather than a selector value. The interrupt vector number is scaled by the interrupt-descriptor entry size to form an offset into the IDT. The interrupt-descriptor entry size depends on the processor operating mode as follows:

- In long mode, interrupt descriptor-table entries are 16 bytes.
- In legacy mode, interrupt descriptor-table entries are eight bytes.

Figure 4-12 shows how the interrupt vector number indexes the IDT.

![Diagram of IDT indexing](image)

**Figure 4-12. Indexing an IDT**

### 4.6.6 Interrupt Descriptor-Table Register

The interrupt descriptor-table register (IDTR) points to the IDT in memory and defines its size. This register is loaded from memory using the LIDT instruction (see “LGDT and LIDT Instructions” on page 164). The format of the IDTR is identical to that of the GDTR in all modes. Figure 4-7 on page 77 shows the format of the IDTR in legacy mode. Figure 4-8 on page 77 shows the format of the IDTR in long mode.

The offsets into the descriptor tables are not extended by the AMD64 architecture in support of long mode. Therefore, the IDTR limit-field size is unchanged from the legacy size. The processor does check the IDT limit in long mode during IDT accesses.
### 4.7 Legacy Segment Descriptors

#### 4.7.1 Descriptor Format

Segment descriptors define, protect, and isolate segments from each other. There are two basic types of descriptors, each of which are used to describe different segment (or gate) types:

- **User Segments**—These include code segments and data segments. Stack segments are a type of data segment.
- **System Segments**—System segments consist of LDT segments and task-state segments (TSS). Gate descriptors are another type of system-segment descriptor. Rather than describing segments, gate descriptors point to program entry points.

Figure 4-13 shows the generic format for user-segment and system-segment descriptors. User and system segments are differentiated using the S bit. S=1 indicates a user segment, and S=0 indicates a system segment. Gray shading indicates the field or bit is reserved. The format for a gate descriptor differs from the generic segment descriptor, and is described separately in “Gate Descriptors” on page 88.

![Figure 4-13. Generic Segment Descriptor—Legacy Mode](image)

Figure 4-13 shows the fields in a generic, legacy-mode, 8-byte (two doubleword) segment descriptor. In this figure, the upper doubleword (located at byte offset +4) is shown on top and the lower doubleword (located at byte offset +0) is shown on the bottom. The fields are defined as follows:

**Segment Limit.** The 20-bit segment limit is formed by concatenating bits 19:16 of the upper doubleword with bits 15:0 of lower doubleword. The segment limit defines the segment size, in bytes. The granularity (G) bit controls how the segment-limit field is scaled (see “Granularity (G) Bit” on page 83). For data segments, the expand-down (E) bit determines whether the segment limit defines the lower or upper segment-boundary (see “Expand-Down (E) Bit” on page 86).

If software references a segment descriptor with an address beyond the segment limit, a general-protection exception (#GP) occurs. The #GP occurs if any part of the memory reference falls outside the segment limit. For example, a doubleword (4-byte) address reference causes a #GP if one or more bytes are located beyond the segment limit.

**Base Address.** The 32-bit base address is formed by concatenating bits 31:24 of the upper doubleword with bits 7:0 of the same doubleword and bits 15:0 of the lower doubleword. The segment-base address field locates the start of a segment in virtual-address space.
**Segmented Virtual Memory**

**S Bit and Type Field.** Bit 12 and bits 11:8 of the upper doubleword. The S and Type fields, together, specify the descriptor type and its access characteristics. Table 4-2 summarizes the descriptor types by S-field encoding and gives a cross reference to descriptions of the Type-field encodings.

<table>
<thead>
<tr>
<th>S Field</th>
<th>Descriptor Type</th>
<th>Type-Field Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (System)</td>
<td>LDT</td>
<td>See Table 4-5 on page 87</td>
</tr>
<tr>
<td></td>
<td>TSS</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Gate</td>
<td></td>
</tr>
<tr>
<td>1 (User)</td>
<td>Code</td>
<td>See Table 4-3 on page 85</td>
</tr>
<tr>
<td></td>
<td>Data</td>
<td>See Table 4-4 on page 86</td>
</tr>
</tbody>
</table>

**Descriptor Privilege-Level (DPL) Field.** Bits 14:13 of the upper doubleword. The DPL field indicates the descriptor-privilege level of the segment. DPL can be set to any value from 0 to 3, with 0 specifying the most privilege and 3 the least privilege. See “Data-Access Privilege Checks” on page 99 and “Control-Transfer Privilege Checks” on page 102 for more information on how the DPL is used during segment privilege-checks.

**Present (P) Bit.** Bit 15 of the upper doubleword. The segment-present bit indicates that the segment referenced by the descriptor is loaded in memory. If a reference is made to a descriptor entry when P = 0, a segment-not-present exception (#NP) occurs. This bit is set and cleared by system software and is never altered by the processor.

**Available To Software (AVL) Bit.** Bit 20 of the upper doubleword. This field is available to software, which can write any value to it. The processor does not set or clear this field.

**Default Operand Size (D/B) Bit.** Bit 22 of the upper doubleword. The default operand-size bit is found in code-segment and data-segment descriptors but not in system-segment descriptors. Setting this bit to 1 indicates a 32-bit default operand size, and clearing it to 0 indicates a 16-bit default size. The effect this bit has on a segment depends on the segment-descriptor type. See “Code-Segment Default-Operand Size (D) Bit” on page 85 for a description of the D bit in code-segment descriptors. “Data-Segment Default Operand Size (D/B) Bit” on page 87 describes the D bit in data-segment descriptors, including stack segments, where the bit is referred to as the “B” bit.

**Granularity (G) Bit.** Bit 23 of the upper doubleword. The granularity bit specifies how the segment-limit field is scaled. Clearing the G bit to 0 indicates that the limit field is not scaled. In this case, the limit equals the number of bytes available in the segment. Setting the G bit to 1 indicates that the limit field is scaled by 4 Kbytes (4096 bytes). Here, the limit field equals the number of 4-Kbyte blocks available in the segment.

Setting a limit of 0 indicates a 1-byte segment limit when G = 0. Setting the same limit of 0 when G = 1 indicates a segment limit of 4095.
**Reserved Bits.** Generally, software should clear all reserved bits to 0, so they can be defined in future revisions to the AMD64 architecture.

### 4.7.2 Code-Segment Descriptors

Figure 4-14 shows the code-segment descriptor format (gray shading indicates the bit is reserved). All software tasks require that a segment selector, referencing a valid code-segment descriptor, is loaded into the CS register. Code segments establish the processor operating mode and execution privilege-level. The segments generally contain only instructions and are execute-only, or execute and read-only. Software cannot write into a segment whose selector references a code-segment descriptor.

Code-segment descriptors have the S bit set to 1, identifying the segments as user segments. Type-field bit 11 differentiates code-segment descriptors (bit 11 set to 1) from data-segment descriptors (bit 11 cleared to 0). The remaining type-field bits (10:8) define the access characteristics for the code-segment, as follows:

**Conforming (C) Bit.** Bit 10 of the upper doubleword. Setting this bit to 1 identifies the code segment as conforming. When control is transferred to a higher-privilege conforming code-segment (C=1) from a lower-privilege code segment, the processor CPL does not change. Transfers to non-conforming code-segments (C = 0) with a higher privilege-level than the CPL can occur only through gate descriptors. See “Control-Transfer Privilege Checks” on page 102 for more information on conforming and non-conforming code-segments.

**Readable (R) Bit.** Bit 9 of the upper doubleword. Setting this bit to 1 indicates the code segment is both executable and readable as data. When this bit is cleared to 0, the code segment is executable, but attempts to read data from the code segment cause a general-protection exception (#GP) to occur.

**Accessed (A) Bit.** Bit 8 of the upper doubleword. The accessed bit is set to 1 by the processor when the descriptor is copied from the GDT or LDT into the CS register. This bit is only cleared by software. Table 4-3 on page 85 summarizes the code-segment type-field encodings.
Table 4-3. Code-Segment Descriptor Types

<table>
<thead>
<tr>
<th>Hex Value (Code/Data)</th>
<th>Type Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bit 11</td>
<td>Bit 10</td>
</tr>
<tr>
<td></td>
<td>Bit 22</td>
<td>Bit 21</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Code-Segment Default-Operand Size (D) Bit. Bit 22 of byte +4. In code-segment descriptors, the D bit selects the default operand size and address sizes. In legacy mode, when D=0 the default operand size and address size is 16 bits and when D=1 the default operand size and address size is 32 bits. Instruction prefixes can be used to override the operand size or address size, or both.

4.7.3 Data-Segment Descriptors

Figure 4-15 shows the data-segment descriptor format. Data segments contain non-executable information and can be accessed as read-only or read/write. They are referenced using the DS, ES, FS, GS, or SS data-segment registers. The DS data-segment register holds the segment selector for the default data segment. The ES, FS and GS data-segment registers hold segment selectors for additional data segments usable by the current software task.

The stack segment is a special form of data-segment register. It is referenced using the SS segment register and must be read/write. When loading the SS register, the processor requires that the selector reference a valid, writable data-segment descriptor.

Figure 4-15. Data-Segment Descriptor—Legacy Mode
Data-segment descriptors have the S bit set to 1, identifying them as user segments. Type-field bit 11 differentiates data-segment descriptors (bit 11 cleared to 0) from code-segment descriptors (bit 11 set to 1). The remaining type-field bits (10:8) define the data-segment access characteristics, as follows:

**Expand-Down (E) Bit.** Bit 10 of the upper doubleword. Setting this bit to 1 identifies the data segment as expand-down. In expand-down segments, the segment limit defines the lower segment boundary while the base is the upper boundary. Valid segment offsets in expand-down segments lie in the byte range limit+1 to FFFFh or FFFF_FFFFh, depending on the value of the data segment default operand size (D/B) bit.

Expand-down segments are useful for stacks, which grow in the downward direction as elements are pushed onto the stack. The stack pointer, ESP, is decremented by an amount equal to the operand size as a result of executing a PUSH instruction.

Clearing the E bit to 0 identifies the data segment as expand-up. Valid segment offsets in expand-up segments lie in the byte range 0 to segment limit.

**Writable (W) Bit.** Bit 9 of the upper doubleword. Setting this bit to 1 identifies the data segment as read/write. When this bit is cleared to 0, the segment is read-only. A general-protection exception (#GP) occurs if software attempts to write into a data segment when W=0.

**Accessed (A) Bit.** Bit 8 of the upper doubleword. The accessed bit is set to 1 by the processor when the descriptor is copied from the GDT or LDT into one of the data-segment registers or the stack-segment register. This bit is only cleared by software.

Table 4-4 summarizes the data-segment type-field encodings.

<table>
<thead>
<tr>
<th>Hex Value</th>
<th>Bit 11 (Code/Data)</th>
<th>Type Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Bit 10 Expand-Down</td>
<td>Bit 9 Writable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(E)</td>
<td>(W)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Data-Segment Default Operand Size (D/B) Bit. Bit 22 of the upper doubleword. For expand-down data segments (E=1), setting D=1 sets the upper bound of the segment at 0_FFFF_FFFFh. Clearing D=0 sets the upper bound of the segment at 0_FFFFh.

In the case where a data segment is referenced by the stack selector (SS), the D bit is referred to as the B bit. For stack segments, the B bit sets the default stack size. Setting B=1 establishes a 32-bit stack referenced by the 32-bit ESP register. Clearing B=0 establishes a 16-bit stack referenced by the 16-bit SP register.

4.7.4 System Descriptors

There are two general types of system descriptors: system-segment descriptors and gate descriptors. System-segment descriptors are used to describe the LDT and TSS segments. Gate descriptors do not describe segments, but instead hold pointers to code-segment descriptors. Gate descriptors are used for protected-mode control transfers between less-privileged and more-privileged software.

System-segment descriptors have the S bit cleared to 0. The type field is used to differentiate the various LDT, TSS, and gate descriptors from one another. Table 4-5 summarizes the system-segment type-field encodings.

<table>
<thead>
<tr>
<th>Hex Value</th>
<th>Type Field (Bits 11:8)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>Reserved (Illegal)</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>Available 16-bit TSS</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>LDT</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>Busy 16-bit TSS</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>16-bit Call Gate</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>Task Gate</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>16-bit Interrupt Gate</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>16-bit Trap Gate</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>Reserved (Illegal)</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>Available 32-bit TSS</td>
</tr>
<tr>
<td>A</td>
<td>1010</td>
<td>Reserved (Illegal)</td>
</tr>
<tr>
<td>B</td>
<td>1011</td>
<td>Busy 32-bit TSS</td>
</tr>
<tr>
<td>C</td>
<td>1100</td>
<td>32-bit Call Gate</td>
</tr>
<tr>
<td>D</td>
<td>1101</td>
<td>Reserved (Illegal)</td>
</tr>
<tr>
<td>E</td>
<td>1110</td>
<td>32-bit Interrupt Gate</td>
</tr>
<tr>
<td>F</td>
<td>1111</td>
<td>32-bit Trap Gate</td>
</tr>
</tbody>
</table>
Figure 4-16 shows the legacy-mode system-segment descriptor format used for referencing LDT and TSS segments (gray shading indicates the bit is reserved). This format is also used in compatibility mode. The system-segments are used as follows:

- The LDT typically holds segment descriptors belonging to a single task (see “Local Descriptor Table” on page 77).
- The TSS is a data structure for holding processor-state information. Processor state is saved in a TSS when a task is suspended, and state is restored from the TSS when a task is restarted. System software must create at least one TSS referenced by the task register, TR. See “Legacy Task-State Segment” on page 341 for more information on the TSS.

![Figure 4-16. LDT and TSS Descriptor—Legacy/Compatibility Modes](image)

### 4.7.5 Gate Descriptors

Gate descriptors hold pointers to code segments and are used to control access between code segments with different privilege levels. There are four types of gate descriptors:

- **Call Gates**—These gates (Figure 4-17 on page 89) are located in the GDT or LDT and are used to control access between code segments in the same task or in different tasks. See “Control Transfers Through Call Gates” on page 106 for information on how call gates are used to control access between code segments operating in the same task. The format of a call-gate descriptor is shown in Figure 4-17 on page 89.

- **Interrupt Gates** and **Trap Gates**—These gates (Figure 4-18 on page 89) are located in the IDT and are used to control access to interrupt-service routines. “Legacy Protected-Mode Interrupt Control Transfers” on page 245 contains information on using these gates for interrupt-control transfers. The format of interrupt-gate and trap-gate descriptors is shown in Figure 4-17 on page 89.

- **Task Gates**—These gates (Figure 4-19 on page 89) are used to control access between different tasks. They are also used to transfer control to interrupt-service routines if those routines are themselves a separate task. See “Task-Management Resources” on page 336 for more information on task gates and their use.
There are several differences between the gate-descriptor format and the system-segment descriptor format. These differences are described as follows, from least-significant to most-significant bit positions:

**Target Code-Segment Offset.** The 32-bit segment offset is formed by concatenating bits 31:16 of byte +4 with bits 15:0 of byte +0. The segment-offset field specifies the target-procedure entry point (offset) into the segment. This field is loaded into the EIP register as a result of a control transfer using the gate descriptor.

**Target Code-Segment Selector.** Bits 31:16 of byte +0. The segment-selector field identifies the target-procedure segment descriptor, located in either the GDT or LDT. The segment selector is loaded into the CS segment register as a result of a control transfer using the gate descriptor.

**TSS Selector.** Bits 31:16 of byte +0 (task gates only). This field identifies the target-task TSS descriptor, located in any of the three descriptor tables (GDT, LDT, and IDT).
Parameter Count (Call Gates Only). Bits 4:0 of byte +4. Legacy-mode call-gate descriptors contain a 5-bit parameter-count field. This field specifies the number of parameters to be copied from the currently-executing program stack to the target program stack during an automatic stack switch. Automatic stack switches are performed by the processor during a control transfer through a call gate to a greater privilege-level. The parameter size depends on the call-gate size as specified in the type field. 32-bit call gates copy 4-byte parameters, and 16-bit call gates copy 2-byte parameters. See “Stack Switching” on page 110 for more information on call-gate parameter copying.

4.8 Long-Mode Segment Descriptors

The interpretation of descriptor fields is changed in long mode, and in some cases the format is expanded. The changes depend on the operating mode (compatibility mode or 64-bit mode) and on the descriptor type. The following sections describe the changes.

4.8.1 Code-Segment Descriptors

Code segments continue to exist in long mode. Code segments and their associated descriptors and selectors are needed to establish the processor operating mode as well as execution privilege-level. The new L attribute specifies whether the processor is running in compatibility mode or 64-bit mode (see “Long (L) Attribute Bit” on page 91). Figure 4-20 shows the long-mode code-segment descriptor format. In compatibility mode, the code-segment descriptor is interpreted and behaves just as it does in legacy mode as described in “Code-Segment Descriptors” on page 84.

In Figure 4-20, gray shading indicates the code-segment descriptor fields that are ignored in 64-bit mode when the descriptor is used during a memory reference. However, the fields are loaded whenever the segment register is loaded in 64-bit mode.

![Figure 4-20. Code-Segment Descriptor—Long Mode](image)

Fields Ignored in 64-Bit Mode. Segmentation is disabled in 64-bit mode, and code segments span all of virtual memory. In this mode, code-segment base addresses are ignored. For the purpose of virtual-address calculations, the base address is treated as if it has a value of zero.

Segment-limit checking is not performed, and both the segment-limit field and granularity (G) bit are ignored. Instead, the virtual address is checked to see if it is in canonical-address form.

The readable (R) and accessed (A) attributes in the type field are also ignored.
Long (L) Attribute Bit. Bit 21 of byte +4. Long mode introduces a new attribute, the long (L) bit, in code-segment descriptors. This bit specifies that the processor is running in 64-bit mode (L=1) or compatibility mode (L=0). When the processor is running in legacy mode, this bit is reserved.

Compatibility mode maintains binary compatibility with legacy 16-bit and 32-bit applications. Compatibility mode is selected on a code-segment basis, and it allows legacy applications to coexist under the same 64-bit system software along with 64-bit applications running in 64-bit mode. System software running in long mode can execute existing 16-bit and 32-bit applications by clearing the L bit of the code-segment descriptor to 0.

When L=0, the legacy meaning of the code-segment D bit (see “Code-Segment Default-Operand Size (D) Bit” on page 85)—and the address-size and operand-size prefixes—are observed. Segmentation is enabled when L=0. From an application viewpoint, the processor is in a legacy 16-bit or 32-bit operating environment (depending on the D bit), even though long mode is activated.

If the processor is running in 64-bit mode (L=1), the only valid setting of the D bit is 0. This setting produces a default operand size of 32 bits and a default address size of 64 bits. The combination L=1 and D=1 is reserved for future use.

“Instruction Prefixes” in Volume 3 describes the effect of the code-segment L and D bits on default operand and address sizes when long mode is activated. These default sizes can be overridden with operand size, address size, and REX prefixes.

4.8.2 Data-Segment Descriptors

Data segments continue to exist in long mode. Figure 4-21 shows the long-mode data-segment descriptor format. In compatibility mode, data-segment descriptors are interpreted and behave just as they do in legacy mode.

In Figure 4-21, gray shading indicates the fields that are ignored in 64-bit mode when the descriptor is used during a memory reference. However, the fields are loaded whenever the segment register is loaded in 64-bit mode.

<table>
<thead>
<tr>
<th>31</th>
<th>24</th>
<th>23</th>
<th>22</th>
<th>21</th>
<th>20</th>
<th>19</th>
<th>16</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Base Address[15:0]</td>
<td>Segment Limit[15:0]</td>
<td></td>
<td>+0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4-21. Data-Segment Descriptor—Long Mode

Fields Ignored in 64-Bit Mode. Segmentation is disabled in 64-bit mode. The interpretation of the segment-base address depends on the segment register used:
• In data-segment descriptors referenced by the DS, ES and SS segment registers, the base-address field is ignored. For the purpose of virtual-address calculations, the base address is treated as if it has a value of zero.

• Data segments referenced by the FS and GS segment registers receive special treatment in 64-bit mode. For these segments, the base address field is not ignored, and a non-zero value can be used in virtual-address calculations. A 64-bit segment-base address can be specified using model-specific registers. See “FS and GS Registers in 64-Bit Mode” on page 74 for more information.

Segment-limit checking is not performed on any data segments in 64-bit mode, and both the segment-limit field and granularity (G) bit are ignored. The D/B bit is unused in 64-bit mode.

The expand-down (E), writable (W), and accessed (A) type-field attributes are ignored.

A data-segment-descriptor DPL field is ignored in 64-bit mode, and segment-privilege checks are not performed on data segments. System software can use the page-protection mechanisms to isolate and protect data from unauthorized access.

4.8.3 System Descriptors

In long mode, the allowable system-descriptor types encoded by the type field are changed. Some descriptor types are modified, and others are illegal. The changes are summarized in Table 4-6. An attempt to use an illegal descriptor type causes a general-protection exception (#GP).

Table 4-6. System-Segment Descriptor Types—Long Mode

<table>
<thead>
<tr>
<th>Hex Value</th>
<th>Type Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bit 11</td>
<td>Bit 10</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Note(s):

1. In 64-bit mode only. In compatibility mode, the type specifies a 32-bit LDT.
In long mode, the modified system-segment descriptor types are:

- The 32-bit LDT (02h), which is redefined as the 64-bit LDT.
- The available 32-bit TSS (09h), which is redefined as the available 64-bit TSS.
- The busy 32-bit TSS (0Bh), which is redefined as the busy 64-bit TSS.

In 64-bit mode, the LDT and TSS system-segment descriptors are expanded by 64 bits, as shown in Figure 4-22. In this figure, gray shading indicates the fields that are ignored in 64-bit mode. Expanding the descriptors allows them to hold 64-bit base addresses, so their segments can be located anywhere in the virtual-address space. The expanded descriptor can be loaded into the corresponding descriptor-table register (LDTR or TR) only from 64-bit mode. In compatibility mode, the legacy system-segment descriptor format, shown in Figure 4-16 on page 88, is used. See “LLDT and LTR Instructions” on page 164 for more information.

The 64-bit system-segment base address must be in canonical form. Otherwise, a general-protection exception occurs with a selector error-code, #GP(selector), when the system segment is loaded. System-segment limit values are checked by the processor in both 64-bit and compatibility modes, under the control of the granularity (G) bit.

Figure 4-22 shows that bits 12:8 of doubleword +12 must be cleared to 0. These bits correspond to the S and Type fields in a legacy descriptor. Clearing these bits to 0 corresponds to an illegal type in legacy...
mode and causes a #GP if an attempt is made to access the upper half of a 64-bit mode system-segment descriptor as a legacy descriptor or as the lower half of a 64-bit mode system-segment descriptor.

### 4.8.4 Gate Descriptors

As shown in Table 4-6 on page 92, the allowable gate-descriptor types are changed in long mode. Some gate-descriptor types are modified and others are illegal. The modified gate-descriptor types in long mode are:

- The 32-bit call gate (0Ch), which is redefined as the 64-bit call gate.
- The 32-bit interrupt gate (0Eh), which is redefined as the 64-bit interrupt gate.
- The 32-bit trap gate (0Fh), which is redefined as the 64-bit trap gate.

In long mode, several gate-descriptor types are illegal. An attempt to use these gates causes a general-protection exception (#GP) to occur. The illegal gate types are:

- The 16-bit call gate (04h).
- The task gate (05h).
- The 16-bit interrupt gate (06h).
- The 16-bit trap gate (07h).

In long mode, gate descriptors are expanded by 64 bits, allowing them to hold 64-bit offsets. The 64-bit call-gate descriptor is shown in Figure 4-23 and the 64-bit interrupt gate and trap gate are shown in Figure 4-24 on page 95. In these figures, gray shading indicates the fields that are ignored in long mode. The interrupt and trap gates contain an additional field, the IST, that is not present in the call gate—see “IST Field (Interrupt and Trap Gates)” on page 95.

![Figure 4-23. Call-Gate Descriptor—Long Mode](image-url)
The target code segment referenced by a long-mode gate descriptor must be a 64-bit code segment (CS.L=1, CS.D=0). If the target is not a 64-bit code segment, a general-protection exception, #GP(error), occurs. The error code reported depends on the gate type:

- Call gates report the target code-segment selector as the error code.
- Interrupt and trap gates report the interrupt vector number as the error code.

A general-protection exception, #GP(0), occurs if software attempts to reference a long-mode gate descriptor with a target-segment offset that is not in canonical form.

It is possible for software to store legacy and long mode gate descriptors in the same descriptor table. Figure 4-23 on page 94 shows that bits 12:8 of byte +12 in a long-mode call gate must be cleared to 0. These bits correspond to the S and Type fields in a legacy call gate. Clearing these bits to 0 corresponds to an illegal type in legacy mode and causes a #GP if an attempt is made to access the upper half of a 64-bit mode call-gate descriptor as a legacy call-gate descriptor.

It is not necessary to clear these same bits in a long-mode interrupt gate or trap gate. In long mode, the interrupt-descriptor table (IDT) must contain 64-bit interrupt gates or trap gates. The processor automatically indexes the IDT by scaling the interrupt vector by 16. This makes it impossible to access the upper half of a long-mode interrupt gate, or trap gate, as a legacy gate when the processor is running in long mode.

**IST Field (Interrupt and Trap Gates).** Bits 2:0 of byte +4. Long-mode interrupt gate and trap gate descriptors contain a new, 3-bit interrupt-stack-table (IST) field not present in legacy gate descriptors. The IST field is used as an index into the IST portion of a long-mode TSS. If the IST field is not 0, the index references an IST pointer in the TSS, which the processor loads into the RSP register when an interrupt occurs. If the IST index is 0, the processor uses the legacy stack-switching mechanism (with some modifications) when an interrupt occurs. See “Interrupt-Stack Table” on page 259 for more information.
Count Field (Call Gates). The count field found in legacy call-gate descriptors is not supported in long-mode call gates. In long mode, the field is reserved and should be cleared to zero.

4.8.5 Long Mode Descriptor Summary

System descriptors and gate descriptors are expanded by 64 bits to handle 64-bit base addresses in long mode or 64-bit mode. The mode in which the expansion occurs depends on the purpose served by the descriptor, as follows:

- **Expansion Only In 64-Bit Mode**—The system descriptors and pseudo-descriptors that are loaded into the GDTR, IDTR, LDTR, and TR registers are expanded only in 64-bit mode. They are not expanded in compatibility mode.
- **Expansion In Long Mode**—Gate descriptors (call gates, interrupt gates, and trap gates) are expanded in long mode (both 64-bit mode and compatibility mode). Task gates and 16-bit gate descriptors are illegal in long mode.

The AMD64 architecture redefines several of the descriptor-entry fields in support of long mode. The specific change depends on whether the processor is in 64-bit mode or compatibility mode. Table 4-7 summarizes the changes in the descriptor entry field when the descriptor entry is loaded into a segment register (as opposed to when the segment register is subsequently used to access memory).

<table>
<thead>
<tr>
<th>Descriptor Field</th>
<th>Descriptor Type</th>
<th>Long Mode</th>
<th>64-Bit Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Compatibility Mode</td>
<td></td>
</tr>
<tr>
<td>Limit</td>
<td>Code</td>
<td>Same as legacy x86</td>
<td>Same as legacy x86</td>
</tr>
<tr>
<td></td>
<td>Data</td>
<td>Same as legacy x86</td>
<td>Same as legacy x86</td>
</tr>
<tr>
<td></td>
<td>System</td>
<td>Same as legacy x86</td>
<td>Same as legacy x86</td>
</tr>
<tr>
<td>Offset</td>
<td>Gate</td>
<td>Expanded to 64 bits</td>
<td>Expanded to 64 bits</td>
</tr>
<tr>
<td>Base</td>
<td>Code</td>
<td>Same as legacy x86</td>
<td>Same as legacy x86</td>
</tr>
<tr>
<td></td>
<td>Data</td>
<td>Same as legacy x86</td>
<td>Same as legacy x86</td>
</tr>
<tr>
<td></td>
<td>System</td>
<td>Same as legacy x86</td>
<td>Same as legacy x86</td>
</tr>
<tr>
<td>Selector</td>
<td>Gate</td>
<td>Same as legacy x86</td>
<td>Same as legacy x86</td>
</tr>
<tr>
<td>IST₁</td>
<td>Gate</td>
<td>Interrupt and trap gates only. (New for long mode.)</td>
<td></td>
</tr>
<tr>
<td>S and Type</td>
<td>Code</td>
<td>Same as legacy x86</td>
<td>Same as legacy x86</td>
</tr>
<tr>
<td></td>
<td>Data</td>
<td>Same as legacy x86</td>
<td>Same as legacy x86</td>
</tr>
<tr>
<td></td>
<td>System</td>
<td>Types 02h, 09h, and 0Bh redefined Types 01h and 03h are illegal</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Gate</td>
<td>Types 0Ch, 0Eh, and 0Fh redefined Types 04h–07h are illegal</td>
<td></td>
</tr>
</tbody>
</table>

Note(s):
1. Not available (reserved) in legacy mode.
Table 4-7. Descriptor-Entry Field Changes in Long Mode (continued)

<table>
<thead>
<tr>
<th>Descriptor Field</th>
<th>Descriptor Type</th>
<th>Long Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compatibility Mode</td>
<td>64-Bit Mode</td>
</tr>
<tr>
<td>DPL</td>
<td>Same as legacy x86</td>
<td>Same as legacy x86</td>
</tr>
<tr>
<td>Present</td>
<td>Same as legacy x86</td>
<td>Same as legacy x86</td>
</tr>
<tr>
<td>Default Size</td>
<td>Same as legacy x86</td>
<td>D=0 Indicates 64-bit address, 32-bit data, D=1 Reserved</td>
</tr>
<tr>
<td>Long¹</td>
<td>Specifies compatibility mode</td>
<td>Specifies 64-bit mode</td>
</tr>
<tr>
<td>Granularity</td>
<td>Same as legacy x86</td>
<td>Same as legacy x86</td>
</tr>
<tr>
<td>Available</td>
<td>Same as legacy x86</td>
<td>Same as legacy x86</td>
</tr>
</tbody>
</table>

Note(s):
1. Not available (reserved) in legacy mode.

4.9 Segment-Protection Overview

The AMD64 architecture is designed to fully support the legacy segment-protection mechanism. The segment-protection mechanism provides system software with the ability to restrict program access into other software routines and data.

Segment-level protection remains enabled in compatibility mode. 64-bit mode eliminates most type checking, and limit checking is not performed, except on accesses to system-descriptor tables.

The preferred method of implementing memory protection in a long-mode operating system is to rely on the page-protection mechanism as described in “Page-Protection Checks” on page 149. System software still needs to create basic segment-protection data structures for 64-bit mode. These structures are simplified, however, by the use of the flat-memory model in 64-bit mode, and the limited segmentation checks performed when executing in 64-bit mode.
4.9.1 Privilege-Level Concept

Segment protection is used to isolate and protect programs and data from each other. The segment-protection mechanism supports four privilege levels in protected mode. The privilege levels are designated with a numerical value from 0 to 3, with 0 being the most privileged and 3 being the least privileged. System software typically assigns the privilege levels in the following manner:

- **Privilege-level 0 (most privilege)**—This level is used by critical system-software components that require direct access to, and control over, all processor and system resources. This can include platform firmware, memory-management functions, and interrupt handlers.

- **Privilege-levels 1 and 2 (moderate privilege)**—These levels are used by less-critical system-software services that can access and control a limited scope of processor and system resources. Software running at these privilege levels might include some device drivers and library routines. These software routines can call more-privileged system-software services to perform functions such as memory garbage-collection and file allocation.

- **Privilege-level 3 (least privilege)**—This level is used by application software. Software running at privilege-level 3 is normally prevented from directly accessing most processor and system resources. Instead, applications request access to the protected processor and system resources by calling more-privileged service routines to perform the accesses.

Figure 4-25 shows the relationship of the four privilege levels to each other.

![Privilege-Level Relationships](image)

**Figure 4-25. Privilege-Level Relationships**

4.9.2 Privilege-Level Types

There are three types of privilege levels the processor uses to control access to segments. These are CPL, DPL, and RPL.

**Current Privilege-Level.** The current privilege-level (CPL) is the privilege level at which the processor is currently executing. The CPL is stored in an internal processor register that is invisible to
software. Software changes the CPL by performing a control transfer to a different code segment with a new privilege level.

**Descriptor Privilege-Level.** The descriptor privilege-level (DPL) is the privilege level that system software assigns to individual segments. The DPL is used in privilege checks to determine whether software can access the segment referenced by the descriptor. In the case of gate descriptors, the DPL determines whether software can access the descriptor reference by the gate. The DPL is stored in the segment (or gate) descriptor.

**Requestor Privilege-Level.** The requestor privilege-level (RPL) reflects the privilege level of the program that created the selector. The RPL can be used to let a called program know the privilege level of the program that initiated the call. The RPL is stored in the selector used to reference the segment (or gate) descriptor.

The following sections describe how the CPL, DPL, and RPL are used by the processor in performing privilege checks on data accesses and control transfers. Failure to pass a protection check generally causes an exception to occur.

### 4.10 Data-Access Privilege Checks

#### 4.10.1 Accessing Data Segments

Before loading a data-segment register (DS, ES, FS, or GS) with a segment selector, the processor checks the privilege levels as follows to see if access is allowed:

1. The processor compares the CPL with the RPL in the data-segment selector and determines the effective privilege level for the data access. The processor sets the effective privilege level to the lowest privilege (numerically-higher value) indicated by the comparison.

2. The processor compares the effective privilege level with the DPL in the descriptor-table entry referenced by the segment selector. If the effective privilege level is greater than or equal to (numerically lower-than or equal-to) the DPL, then the processor loads the segment register with the data-segment selector. The processor automatically loads the corresponding descriptor-table entry into the hidden portion of the segment register.

   If the effective privilege level is lower than (numerically greater-than) the DPL, a general-protection exception (#GP) occurs and the segment register is not loaded.

Figure 4-26 on page 100 shows two examples of data-access privilege checks.
Example 1 in Figure 4-26 shows a failing data-access privilege check. The effective privilege level is 3 because CPL=3. This value is greater than the descriptor DPL, so access to the data segment is denied.

Example 2 in Figure 4-26 shows a passing data-access privilege check. Here, the effective privilege level is 0 because both the CPL and RPL have values of 0. This value is less than the descriptor DPL, so access to the data segment is allowed, and the data-segment register is successfully loaded.

4.10.2 Accessing Stack Segments

Before loading the stack segment register (SS) with a segment selector, the processor checks the privilege levels as follows to see if access is allowed:
1. The processor checks that the CPL and the stack-selector RPL are equal. If they are not equal, a general-protection exception (#GP) occurs and the SS register is not loaded.

2. The processor compares the CPL with the DPL in the descriptor-table entry referenced by the segment selector. The two values must be equal. If they are not equal, a #GP occurs and the SS register is not loaded.

Figure 4-27 shows two examples of stack-access privilege checks. In Example 1 the CPL, stack-selector RPL, and stack segment-descriptor DPL are all equal, so access to the stack segment using the SS register is allowed. In Example 2, the stack-selector RPL and stack segment-descriptor DPL are both equal. However, the CPL is not equal to the stack segment-descriptor DPL, and access to the stack segment through the SS register is denied.

Figure 4-27. Stack-Access Privilege-Check Examples
4.11 Control-Transfer Privilege Checks

Control transfers between code segments (also called far control transfers) cause the processor to perform privilege checks to determine whether the source program is allowed to transfer control to the target program. If the privilege checks pass, access to the target code-segment is granted. When access is granted, the target code-segment selector is loaded into the CS register. The rIP register is updated with the target CS offset taken from either the far-pointer operand or the gate descriptor. Privilege checks are not performed during near control transfers because such transfers do not change segments.

The following mechanisms can be used by software to perform far control transfers:

- System-software control transfers using the system-call and system-return instructions. See “SYSCALL and SYSRET” on page 159 and “SYSENTER and SYSEXIT (Legacy Mode Only)” on page 160 for more information on these instructions. SYSCALL and SYSRET are the preferred method of performing control transfers in long mode. SYSENTER and SYSEXIT are not supported in long mode.

- Direct control transfers using CALL and JMP instructions. These are discussed in the next section, “Direct Control Transfers.”

- Call-gate control transfers using CALL and JMP instructions. These are discussed in “Control Transfers Through Call Gates” on page 106.

- Return control transfers using the RET instruction. These are discussed in “Return Control Transfers” on page 113.

- Interrupts and exceptions, including the INTn and IRET instructions. These are discussed in Chapter 8, “Exceptions and Interrupts.”

- Task switches initiated by CALL and JMP instructions. Task switches are discussed in Chapter 12, “Task Management.” The hardware task-switch mechanism is not supported in long mode.

4.11.1 Direct Control Transfers

A direct control transfer occurs when software executes a far-CALL or a far-JMP instruction without using a call gate. The privilege checks and type of access allowed as a result of a direct control transfer depends on whether the target code segment is conforming or nonconforming. The code-segment-descriptor conforming (C) bit indicates whether or not the target code-segment is conforming (see “Conforming (C) Bit” on page 84 for more information on the conforming bit).

Privilege levels are not changed as a result of a direct control transfer. Program stacks are not automatically switched by the processor as they are with privilege-changing control transfers through call gates (see “Stack Switching” on page 110 for more information on automatic stack switching during privilege-changing control transfers).

Nonconforming Code Segments. Software can perform a direct control transfer to a nonconforming code segment only if the target code-segment descriptor DPL and the CPL are equal and the RPL is less than or equal to the CPL. Software must use a call gate to transfer control to a
A more-privileged, nonconforming code segment (see “Control Transfers Through Call Gates” on page 106 for more information).

In far calls and jumps, the far pointer (CS:rIP) references the target code-segment descriptor. Before loading the CS register with a nonconforming code-segment selector, the processor checks as follows to see if access is allowed:

1. \( DPL = CPL \) Check—The processor compares the target code-segment descriptor DPL with the currently executing program CPL. If they are equal, the processor performs the next check. If they are not equal, a general-protection exception (#GP) occurs.

2. \( RPL \leq CPL \) Check—The processor compares the target code-segment selector RPL with the currently executing program CPL. If the RPL is less than or equal to the CPL, access is allowed. If the RPL is greater than the CPL, a #GP exception occurs.

If access is allowed, the processor loads the CS and rIP registers with their new values and begins executing from the target location. The CPL is not changed—the target-CS selector RPL value is disregarded when the selector is loaded into the CS register.

Figure 4-28 on page 104 shows three examples of privilege checks performed as a result of a far control transfer to a nonconforming code-segment. In Example 1, access is allowed because CPL = DPL and RPL \( \leq CPL \). In Example 2, access is denied because CPL \( \neq DPL \). In Example 3, access is denied because RPL > CPL.
**Figure 4-28. Nonconforming Code-Segment Privilege-Check Examples**

**Conforming Code Segments.** On a direct control transfer to a conforming code segment, the target code-segment descriptor DPL can be lower than (at a greater privilege) the CPL. Before loading the
CS register with a conforming code-segment selector, the processor compares the target code-segment descriptor DPL with the currently-executing program CPL. If the DPL is less than or equal to the CPL, access is allowed. If the DPL is greater than the CPL, a #GP exception occurs.

On an access to a conforming code segment, the RPL is ignored and not involved in the privilege check.

When access is allowed, the processor loads the CS and rIP registers with their new values and begins executing from the target location. The CPL is not changed—the target CS-descriptor DPL value is disregarded when the selector is loaded into the CS register. The target program runs at the same privilege as the program that called it.

Figure 4-29 shows two examples of privilege checks performed as a result of a direct control transfer to a conforming code segment. In Example 1, access is allowed because the CPL of 3 is greater than the DPL of 0. As the target code selector is loaded into the CS register, the old CPL value of 3 replaces the target-code selector RPL value, and the target program executes with CPL=3. In Example 2, access is denied because CPL < DPL.

Figure 4-29. Conforming Code-Segment Privilege-Check Examples
4.11.2 Control Transfers Through Call Gates

Control transfers to more-privileged code segments are accomplished through the use of call gates. Call gates are a type of descriptor that contain pointers to code-segment descriptors and control access to those descriptors. System software uses call gates to establish protected entry points into system-service routines.

Transfer Mechanism. The pointer operand of a far-CALL or far-JMP instruction consists of two pieces: a code-segment selector (CS) and a code-segment offset (rIP). In a call-gate transfer, the CS selector points to a call-gate descriptor rather than a code-segment descriptor, and the rIP is ignored (but required by the instruction).

Figure 4-30 shows a call-gate control transfer in legacy mode. The call-gate descriptor contains segment-selector and segment-offset fields (see “Gate Descriptors” on page 88 for a detailed description of the call-gate format and fields). These two fields perform the same function as the pointer operand in a direct control-transfer instruction. The segment-selector field points to the target code-segment descriptor, and the segment-offset field is the instruction-pointer offset into the target code-segment. The code-segment base taken from the code-segment descriptor is added to the offset field in the call-gate descriptor to create the target virtual address (linear address).

Figure 4-30. Legacy-Mode Call-Gate Transfer Mechanism
Figure 4-31 shows a call-gate control transfer in long mode. The long-mode call-gate descriptor format is expanded by 64 bits to hold a full 64-bit offset into the virtual-address space. Only long-mode call gates can be referenced in long mode (64-bit mode and compatibility mode). The legacy-mode 32-bit call-gate types are redefined in long mode as 64-bit types, and 16-bit call-gate types are illegal.

**Figure 4-31. Long-Mode Call-Gate Access Mechanism**

A long-mode call gate must reference a 64-bit code-segment descriptor. In 64-bit mode, the code-segment descriptor base-address and limit fields are ignored. The target virtual-address is the 64-bit offset field in the expanded call-gate descriptor.

**Privilege Checks.** Before loading the CS register with the code-segment selector located in the call gate, the processor performs three privilege checks. The following checks are performed when either conforming or nonconforming code segments are referenced:

1. The processor compares the CPL with the call-gate DPL from the call-gate descriptor (DPLG). The CPL must be numerically *less than or equal to* DPLG for this check to pass. In other words, the following expression must be true: CPL \( \leq \) DPLG.
2. The processor compares the RPL in the call-gate selector with DPL_G. The RPL must be numerically \textit{less than or equal to} DPL_G for this check to pass. In other words, the following expression must be true: \( \text{RPL} \leq \text{DPL}_G \).

3. The processor compares the CPL with the target code-segment DPL from the code-segment descriptor (DPL_S). The type of comparison varies depending on the type of control transfer.
   - When a call—or a jump to a \textit{conforming} code segment—is used to transfer control through a call gate, the CPL must be numerically \textit{greater than or equal to} DPL_S for this check to pass. (This check prevents control transfers to less-privileged programs.) In other words, the following expression must be true: \( \text{CPL} \geq \text{DPL}_S \).
   - When a JMP instruction is used to transfer control through a call gate to a \textit{nonconforming} code segment, the CPL must be numerically \textit{equal to} DPL_S for this check to pass. (JMP instructions cannot change CPL.) In other words, the following expression must be true: \( \text{CPL} = \text{DPL}_S \).

Figure 4-32 on page 109 shows two examples of call-gate privilege checks. In Example 1, all privilege checks pass as follows:

- The call-gate DPL (DPL_G) is at the lowest privilege (3), specifying that software running at any privilege level (CPL) can access the gate.
- The selector referencing the call gate passes its privilege check because the RPL is numerically less than or equal to DPL_G.
- The target code segment is at the highest privilege level (DPL_S = 0). This means software running at any privilege level can access the target code segment through the call gate.
In Example 2, all privilege checks fail as follows:

- The call-gate DPL (\(DPL_G\)) specifies that only software at privilege-level 0 can access the gate. The current program does not have enough privilege to access the call gate because its CPL is 2.
- The selector referencing the call-gate descriptor does not have enough privilege to complete the reference. Its RPL is numerically greater than \(DPL_G\).
• The target code segment is at a lower privilege (DPLs = 3) than the currently running software (CPL = 2). Transitions from more-privileged software to less-privileged software are not allowed, so this privilege check fails as well.

Although all three privilege checks failed in Example 2, failing only one check is sufficient to deny access into the target code segment.

Stack Switching. The processor performs an automatic stack switch when a control transfer causes a change in privilege levels to occur. Switching stacks isolates more-privileged software stacks from less-privileged software stacks and provides a mechanism for saving the return pointer back to the program that initiated the call.

When switching to more-privileged software, as is done when transferring control using a call gate, the processor uses the corresponding stack pointer (privilege-level 0, 1, or 2) stored in the task-state segment (TSS). The format of the stack pointer stored in the TSS depends on the system-software operating mode:

• Legacy-mode system software stores a 32-bit ESP value (stack offset) and 16-bit SS selector register value in the TSS for each of three privilege levels 0, 1, and 2.
• Long-mode system software stores a 64-bit RSP value in the TSS for privilege levels 0, 1, and 2. No SS register value is stored in the TSS because in long mode a call gate must reference a 64-bit code-segment descriptor. 64-bit mode does not use segmentation, and the stack pointer consists solely of the 64-bit RSP. Any value loaded in the SS register is ignored.

See “Task-Management Resources” on page 336 for more information on the legacy-mode and long-mode TSS formats.

Figure 4-33 on page 111 shows a 32-bit stack in legacy mode before and after the automatic stack switch. This particular example assumes that parameters are passed from the current program to the target program. The process followed by legacy mode in switching stacks and copying parameters is:

1. The target code-segment DPL is read by the processor and used as an index into the TSS for selecting the new stack pointer (SS:ESP). For example, if DPL=1 the processor selects the SS:ESP for privilege-level 1 from the TSS.
2. The SS and ESP registers are loaded with the new SS:ESP values read from the TSS.
3. The old values of the SS and ESP registers are pushed onto the stack pointed to by the new SS:ESP.
4. The 5-bit count field is read from the call-gate descriptor.
5. The number of parameters specified in the count field (up to 31) are copied from the old stack to the new stack. The size of the parameters copied by the processor depends on the call-gate size: 32-bit call gates copy 4-byte parameters and 16-bit call gates copy 2-byte parameters.
6. The return pointer is pushed onto the stack. The return pointer consists of the current CS-register value and the EIP of the instruction following the calling instruction.
7. The CS register is loaded from the segment-selector field in the call-gate descriptor, and the EIP is loaded from the offset field in the call-gate descriptor.

8. The target program begins executing with the instruction referenced by new CS:EIP.

**Figure 4-33. Legacy-Mode 32-Bit Stack Switch, with Parameters**

Figure 4-34 shows a 32-bit stack in legacy mode before and after the automatic stack switch when no parameters are passed (count=0). Most software does not use the call-gate descriptor count-field to pass parameters. System software typically defines linkage mechanisms that do not rely on automatic parameter copying.

**Figure 4-34. 32-Bit Stack Switch, No Parameters—Legacy Mode**

Figure 4-35 on page 112 shows a long-mode stack switch. In long mode, all call gates must reference 64-bit code-segment descriptors, so a long-mode stack switch uses a 64-bit stack. The process of
switching stacks in long mode is similar to switching in legacy mode when no parameters are passed. The process is as follows:

1. The target code-segment DPL is read by the processor and used as an index into the 64-bit TSS for selecting the new stack pointer (RSP).

2. The RSP register is loaded with the new RSP value read from the TSS. The SS register is loaded with a null selector (SS = 0). Setting the new SS selector to null allows proper handling of nested control transfers in 64-bit mode. See “Nested Returns to 64-Bit Mode Procedures” on page 114 for additional information.

As in legacy mode, it is desirable to keep the stack-segment requestor privilege-level (SS.RPL) equal to the current privilege-level (CPL). When using a call gate to change privilege levels, the SS.RPL is updated to reflect the new CPL. The SS.RPL is restored from the return-target CS.RPL on the subsequent privilege-level-changing far return.

3. The old values of the SS and RSP registers are pushed onto the stack pointed to by the new RSP. The old SS value is popped on a subsequent far return. This allows system software to set up the SS selector for a compatibility-mode process by executing a RET (or IRET) that changes the privilege level.

4. The return pointer is pushed onto the stack. The return pointer consists of the current CS-register value and the RIP of the instruction following the calling instruction.

5. The CS register is loaded from the segment-selector field in the long-mode call-gate descriptor, and the RIP is loaded from the offset field in the long-mode call-gate descriptor.

The target program begins execution with the instruction referenced by the new RIP.

Figure 4-35. Stack Switch—Long Mode

All long-mode stack pushes resulting from a privilege-level-changing far call are eight-bytes wide and increment the RSP by eight. Long mode ignores the call-gate count field and does not support the automatic parameter-copy feature found in legacy mode. Software can access parameters on the old stack, if necessary, by referencing the old stack segment selector and stack pointer saved on the new process stack.
4.11.3 Return Control Transfers

Returns to calling programs can be performed by using the RET instruction. The following types of returns are possible:

- **Near Return**—Near returns perform control transfers within the same code segment, so the CS register is unchanged. The new offset is popped off the stack and into the rIP register. No privilege checks are performed.

- **Far Return, Same Privilege**—A far return transfers control from one code segment to another. When the original code segment is at the same privilege level as the target code segment, a far pointer (CS:rIP) is popped off the stack and the RPL of the new code segment (CS) is checked. If the requested privilege level (RPL) matches the current privilege level (CPL), then a return is made to the same privilege level. This prevents software from changing the CS value on the stack in an attempt to return to higher-privilege software.

- **Far Return, Less Privilege**—Far returns can change privilege levels, but only to a lower-privilege level. In this case a stack switch is performed between the current, higher-privilege program and the lower-privilege return program. The CS-register and rIP-register values are popped off the stack. The lower-privilege stack pointer is also popped off the stack and into the SS register and rSP register. The processor checks both the CS and SS privilege levels to ensure they are equal and at a lesser privilege than the current CS.

  In the case of nested returns to 64-bit mode, a null selector can be popped into the SS register. See “Nested Returns to 64-Bit Mode Procedures” on page 114.

  Far returns also check the privilege levels of the DS, ES, FS and GS selector registers. If any of these segment registers have a selector with a higher privilege than the return program, the segment register is loaded with the null selector.

**Stack Switching.** The stack switch performed by a far return to a lower-privilege level reverses the stack switch of a call gate to a higher-privilege level, except that parameters are never automatically copied as part of a return. The process followed by a far-return stack switch in long mode and legacy mode is:

1. The return code-segment RPL is read by the processor from the CS value stored on the stack to determine that a lower-privilege control transfer is occurring.

2. The return-program instruction pointer is popped off the current-program (higher privilege) stack and loaded into the CS and rIP registers.

3. The return instruction can include an immediate operand that specifies the number of additional bytes to be popped off of the stack. These bytes may correspond to the parameters pushed onto the stack previously by a call through a call gate containing a non-zero parameter-count field. If the return includes the immediate operand, then the stack pointer is adjusted upward by adding the specified number of bytes to the rSP.

4. The return-program stack pointer is popped off the current-program (higher privilege) stack and loaded into the SS and rSP registers. In the case of nested returns to 64-bit mode, a null selector can be popped into the SS register.
The operand size of a far return determines the size of stack pops when switching stacks. If a far return is used in 64-bit mode to return from a prior call through a long-mode call gate, the far return must use a 64-bit operand size. The 64-bit operand size allows the far return to properly read the stack established previously by the far call.

**Nested Returns to 64-Bit Mode Procedures.** In long mode, a far call that changes privilege levels causes the SS register to be loaded with a null selector (this is the same action taken by an interrupt in long mode). If the called procedure performs another far call to a higher-privileged procedure, or is interrupted, the null SS selector is pushed onto the stack frame, and another null selector is loaded into the SS register. Using a null selector in this way allows the processor to properly handle returns nested within 64-bit-mode procedures and interrupt handlers.

Normally, a RET that pops a null selector into the SS register causes a general-protection exception (#GP) to occur. However, in long mode, the null selector acts as a flag indicating the existence of nested interrupt handlers or other privileged software in 64-bit mode. Long mode allows RET to pop a null selector into SS from the stack under the following conditions:

- The target mode is 64-bit mode.
- The target CPL is less than 3.

In this case, the processor does not load an SS descriptor, and the null selector is loaded into SS without causing a #GP exception.

### 4.12 Limit Checks

Except in 64-bit mode, limit checks are performed by all instructions that reference memory. Limit checks detect attempts to access memory outside the current segment boundary, attempts at executing instructions outside the current code segment, and indexing outside the current descriptor table. If an instruction fails a limit check, either (1) a general-protection exception occurs for all other segment-limit violations or (2) a stack-fault exception occurs for stack-segment limit violations.

In 64-bit mode, segment limits are **not checked** during accesses to any segment referenced by the CS, DS, ES, FS, GS, and SS selector registers. Instead, the processor checks that the virtual addresses used to reference memory are in canonical-address form. In 64-bit mode, as with legacy mode and compatibility mode, descriptor-table limits are **checked**.

#### 4.12.1 Determining Limit Violations

To determine segment-limit violations, the processor checks a virtual (linear) address to see if it falls outside the valid range of segment offsets determined by the segment-limit field in the descriptor. If any part of an operand or instruction falls outside the segment-offset range, a limit violation occurs. For example, a doubleword access, two bytes from an upper segment boundary, causes a segment violation because half of the doubleword is outside the segment.
Three bits from the descriptor entry are used to control how the segment-limit field is interpreted: the granularity (G) bit, the default operand-size (D) bit, and for data segments, the expand-down (E) bit. See “Legacy Segment Descriptors” on page 82 for a detailed description of each bit.

For all segments other than expand-down segments, the minimum segment-offset is 0. The maximum segment-offset depends on the value of the G bit:

- If G=0 (byte granularity), the maximum allowable segment-offset is equal to the value of the segment-limit field.
- If G=1 (4096-byte granularity), the segment-limit field is first scaled by 4096 (1000h). Then 4095 (0FFFh) is added to the scaled value to arrive at the maximum allowable segment-offset, as shown in the following equation:
  \[
  \text{maximum segment-offset} = (\text{limit} \times 1000h) + 0FFFh
  \]
  For example, if the segment-limit field is 0100h, then the maximum allowable segment-offset is 
  \[(0100h \times 1000h) + 0FFFh = 10_{1000}h.\]

In both cases, the maximum segment-size is specified when the descriptor segment-limit field is 0F_FFFFh.

**Expand-Down Segments.** Expand-down data segments are supported in legacy mode and compatibility mode but not in 64-bit mode. With expand-down data segments, the maximum segment offset depends on the value of the D bit in the data-segment descriptor:

- If D=0 the maximum segment-offset is 0_FFFFh.
- If D=1 the maximum segment-offset is 0_FFFF_FFFFh.

The minimum allowable segment offset in expand-down segments depends on the value of the G bit:

- If G=0 (byte granularity), the minimum allowable segment offset is the segment-limit value plus 1. For example, if the segment-limit field is 0100h, then the minimum allowable segment-offset is 0101h.
- If G=1 (4096-byte granularity), the segment-limit value in the descriptor is first scaled by 4096 (1000h), and then 4095 (0FFFh) is added to the scaled value to arrive at a scaled segment-limit value. The minimum allowable segment-offset is this scaled segment-limit value plus 1, as shown in the following equation:
  \[
  \text{minimum segment-offset} = (\text{limit} \times 1000) + 0FFFh + 1
  \]
  For example, if the segment-limit field is 0100h, then the minimum allowable segment-offset is 
  \[(0100h \times 1000h) + 0FFFh + 1 = 10_{1000}h.\]

For expand-down segments, the maximum segment size is specified when the segment-limit value is 0.
4.12.2 Data Limit Checks in 64-bit Mode

In 64-bit mode, data reads and writes are not normally checked for segment-limit violations. When EFER.LMSLE = 1, reads and writes in 64-bit mode at CPL > 0, using the DS, ES, FS, or SS segments, have a segment-limit check applied.

This limit-check uses the 32-bit segment-limit to find the maximum allowable address in the top 4GB of the 64-bit virtual (linear) address space.

<table>
<thead>
<tr>
<th>Memory Address</th>
<th>Effect of Limit Check</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Address ≤ (0xFFFFFFFF_00000000h + 32-bit Limit)</td>
<td>Access OK.</td>
</tr>
<tr>
<td>Linear Address &gt; (0xFFFFFFFF_00000000h + 32-bit Limit)</td>
<td>Exception (#GP or #SS)</td>
</tr>
</tbody>
</table>

This segment-limit check does not apply to accesses through the GS segment, or to code reads. If the DS, ES, FS, or SS segment is null or expand-down, the effect of the limit check is undefined.

4.13 Type Checks

Type checks prevent software from using descriptors in invalid ways. Failing a type check results in an exception. Type checks are performed using five bits from the descriptor entry: the S bit and the 4-bit Type field. Together, these five bits are used to specify the descriptor type (code, data, segment, or gate) and its access characteristics. See “Legacy Segment Descriptors” on page 82 for a detailed description of the S bit and Type-field encodings. Type checks are performed by the processor in compatibility mode as well as legacy mode. Limited type checks are performed in 64-bit mode.

4.13.1 Type Checks in Legacy and Compatibility Modes

The type checks performed in legacy mode and compatibility mode are listed in the following sections.

Descriptor-Table Register Loads. Loads into the LDTR and TR descriptor-table registers are checked for the appropriate system-segment type. The LDTR can only be loaded with an LDT descriptor, and the TR only with a TSS descriptor. The checks are performed during any action that causes these registers to be loaded. This includes execution of the LLDT and LTR instructions and during task switches.

Segment Register Loads. The following restrictions are placed on the segment-descriptor types that can be loaded into the six user segment registers:

- Only code segments can be loaded into the CS register.
- Only writable data segments can be loaded into the SS register.
- Only the following segment types can be loaded into the DS, ES, FS, or GS registers:
  - Read-only or read/write data segments.
  - Readable code segments.
These checks are performed during any action that causes the segment registers to be loaded. This includes execution of the MOV segment-register instructions, control transfers, and task switches.

**Control Transfers.** Control transfers (branches and interrupts) place additional restrictions on the segment types that can be referenced during the transfer:

- The segment-descriptor type referenced by far CALLs and far JMPs must be one of the following:
  - A code segment
  - A call gate or a task gate
  - An available TSS (only allowed in legacy mode)
  - A task gate (only allowed in legacy mode)
- Only code-segment descriptors can be referenced by call-gate, interrupt-gate, and trap-gate descriptors.
- Only TSS descriptors can be referenced by task-gate descriptors.
- The link field (selector) in the TSS can only point to a TSS descriptor. This is checked during an IRET control transfer to a task.
- The far RET and far IRET instructions can only reference code-segment descriptors.
- The interrupt-descriptor table (IDT), which is referenced during interrupt control transfers, can only contain interrupt gates, trap gates, and task gates.

**Segment Access.** After a segment descriptor is successfully loaded into one of the segment registers, reads and writes into the segments are restricted in the following ways:

- Writes are not allowed into read-only data-segment types.
- Writes are not allowed into code-segment types (executable segments).
- Reads from code-segment types are not allowed if the readable (R) type bit is cleared to 0.

These checks are generally performed during execution of instructions that access memory.

### 4.13.2 Long Mode Type Check Differences

**Compatibility Mode and 64-Bit Mode.** The following type checks differ in long mode (64-bit mode and compatibility mode) as compared to legacy mode:

- **System Segments**—System-segment types are checked, but the following types that are valid in legacy mode are illegal in long mode:
  - 16-bit available TSS.
  - 16-bit busy TSS.
  - Type-field encoding of 00h in the upper half of a system-segment descriptor to indicate an illegal type and prevent access as a legacy descriptor.
- **Gates**—Gate-descriptor types are checked, but the following types that are valid in legacy mode are illegal in long mode:
- 16-bit call gate.
- 16-bit interrupt gate.
- 16-bit trap gate.
- Task gate.

**64-Bit Mode.** 64-bit mode disables segmentation, and most of the segment-descriptor fields are ignored. The following list identifies situations where type checks in 64-bit mode differ from those in compatibility mode and legacy mode:

- **Code Segments**—The readable (R) type bit is ignored in 64-bit mode. None of the legacy type-checks that prevent reads from or writes into code segments are performed in 64-bit mode.

- **Data Segments**—Data-segment type attributes are ignored in 64-bit mode. The writable (W) and expand-down (E) type bits are ignored. All data segments are treated as writable.
5 Page Translation and Protection

The x86 page-translation mechanism (or simply paging mechanism) enables system software to create separate address spaces for each process or application. These address spaces are known as virtual-address spaces. System software uses the paging mechanism to selectively map individual pages of physical memory into the virtual-address space using a set of hierarchical address-translation tables known collectively as page tables.

The paging mechanism and the page tables are used to provide each process with its own private region of physical memory for storing its code and data. Processes can be protected from each other by isolating them within the virtual-address space. A process cannot access physical memory that is not mapped into its virtual-address space by system software.

System software can use the paging mechanism to selectively map physical-memory pages into multiple virtual-address spaces. Mapping physical pages in this manner allows them to be shared by multiple processes and applications. The physical pages can be configured by the page tables to allow read-only access. This prevents applications from altering the pages and ensures their integrity for use by all applications.

Shared mapping is typically used to allow access of shared-library routines by multiple applications. A read-only copy of the library routine is mapped to each application virtual-address space, but only a single copy of the library routine is present in physical memory. This capability also allows a copy of the operating-system kernel and various device drivers to reside within the application address space. Applications are provided with efficient access to system services without requiring costly address-space switches.

The system-software portion of the address space necessarily includes system-only data areas that must be protected from accesses by applications. System software uses the page tables to protect this memory by designating the pages as supervisor pages. Such pages are only accessible by system software.

When the supervisor mode execution prevention (SMEP) feature is supported and enabled, attempted instruction fetches from user-mode accessible pages while in supervisor-mode triggers a page fault (#PF). This protects the integrity of system software by preventing the execution of instructions at a supervisor privilege level (CPL < 3) when these instructions could have been written or modified by user-mode code.

Finally, system software can use the paging mechanism to map multiple, large virtual-address spaces into a much smaller amount of physical memory. Each application can use the entire 32-bit or 64-bit virtual-address space. System software actively maps the most-frequently-used virtual-memory pages into the available pool of physical-memory pages. The least-frequently-used virtual-memory pages are swapped out to the hard drive. This process is known as demand-paged virtual memory.
5.1 Page Translation Overview

The legacy x86 architecture provides support for translating 32-bit virtual addresses into 32-bit physical addresses (larger physical addresses, such as 36-bit or 40-bit addresses, are supported as a special mode). The AMD64 architecture enhances this support to allow translation of 64-bit virtual addresses into 52-bit physical addresses, although processor implementations can support smaller virtual-address and physical-address spaces.

Virtual addresses are translated to physical addresses through hierarchical translation tables created and managed by system software. Each table contains a set of entries that point to the next-lower table in the translation hierarchy. A single table at one level of the hierarchy can have hundreds of entries, each of which points to a unique table at the next-lower hierarchical level. Each lower-level table can in turn have hundreds of entries pointing to tables further down the hierarchy. The lowest-level table in the hierarchy points to the translated physical page.

Figure 5-1 on page 121 shows an overview of the page-translation hierarchy used in long mode. Legacy mode paging uses a subset of this translation hierarchy (the page-map level-4 table does not exist in legacy mode and the PDP table may or may not be used, depending on which paging mode is enabled). As this figure shows, a virtual address is divided into fields, each of which is used as an offset into a translation table. The complete translation chain is made up of all table entries referenced by the virtual-address fields. The lowest-order virtual-address bits are used as the byte offset into the physical page.
Figure 5-1. Virtual to Physical Address Translation—Long Mode
The following physical-page sizes are supported: 4 Kbytes, 2 Mbytes, 4 Mbytes, and 1 Gbytes. In long mode 4-Kbyte, 2-MByte, and 1-GByte sizes are available. In legacy mode 4-Kbyte, 2-MByte, and 4-MByte sizes are available.

Virtual addresses are 32 bits long, and physical addresses up to the supported physical-address size can be used. The AMD64 architecture enhances the legacy translation support by allowing virtual addresses of up to 64 bits long to be translated into physical addresses of up to 52 bits long.

Currently, the AMD64 architecture defines a mechanism for translating 48-bit virtual addresses to 52-bit physical addresses. The mechanism used to translate a full 64-bit virtual address is reserved and will be described in a future AMD64 architectural specification.

### 5.1.1 Page-Translation Options

The form of page-translation support available to software depends on which paging features are enabled. Four controls are available for selecting the various paging alternatives:

- Page-Translation Enable (CR0.PG)
- Physical-Address Extensions (CR4.PAE)
- Page-Size Extensions (CR4.PSE)
- Long-Mode Active (EFER.LMA)

Not all paging alternatives are available in all modes. Table 5-1 summarizes the paging support available in each mode.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Long Mode (64-bit and compatibility modes)</td>
<td>Enabled</td>
<td>–</td>
<td>PDPE.PS=0</td>
<td>PDE.PS=0</td>
<td>4 Kbyte</td>
<td>64-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PDE.PS=1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PDPE.PS=1</td>
<td>2 Mbyte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Legacy Mode</td>
<td>Enabled</td>
<td>–</td>
<td>PDPE.PS=0</td>
<td>PDE.PS=0</td>
<td>4 Kbyte</td>
<td>32-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PDE.PS=1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Disabled</td>
<td>PDPE.PS=0</td>
<td>–</td>
<td>4 Kbyte</td>
<td>32-bit</td>
<td>32-bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PDE.PS=0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>PDE.PS=1</td>
<td>2 Mbyte</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 5.1.2 Page-Translation Enable (PG) Bit

Page translation is controlled by the PG bit in CR0 (bit 31). When CR0.PG is set to 1, page translation is enabled. When CR0.PG is cleared to 0, page translation is disabled.
The AMD64 architecture uses CR0.PG to activate and deactivate long mode when long mode is enabled. See “Enabling and Activating Long Mode” on page 444 for more information.

### 5.1.3 Physical-Address Extensions (PAE) Bit

Physical-address extensions are controlled by the PAE bit in CR4 (bit 5). When CR4.PAE is set to 1, physical-address extensions are enabled. When CR4.PAE is cleared to 0, physical-address extensions are disabled.

Setting CR4.PAE = 1 enables virtual addresses to be translated into physical addresses up to 52 bits long. This is accomplished by doubling the size of paging data-structure entries from 32 bits to 64 bits to accommodate the larger physical base-addresses for physical-pages.

PAE must be enabled before activating long mode. See “Enabling and Activating Long Mode” on page 444.

### 5.1.4 Page-Size Extensions (PSE) Bit

Page-size extensions are controlled by the PSE bit in CR4 (bit 4). Setting CR4.PSE to 1 allows operating-system software to use 4-Mbyte physical pages in the translation process. The 4-Mbyte physical pages can be mixed with standard 4-Kbyte physical pages or replace them entirely. The selection of physical-page size is made on a page-directory-entry basis. See “Page Size (PS) Bit” on page 143 for more information on physical-page size selection. When CR4.PSE is cleared to 0, page-size extensions are disabled.

The choice of 2 Mbyte or 4 Mbyte as the large physical-page size depends on the value of CR4.PSE and CR4.PAE, as follows:

- If physical-address extensions are enabled (CR4.PAE=1), the large physical-page size is 2 Mbytes, regardless of the value of CR4.PSE.
- If physical-address extensions are disabled (CR4.PAE=0) and CR4.PSE=1, the large physical-page size is 4 Mbytes.
- If both CR4.PAE=0 and CR4.PSE=0, the only available page size is 4 Kbytes.

The value of CR4.PSE is ignored when long mode is active. This is because physical-address extensions must be enabled in long mode, and the only available page sizes are 4 Kbytes and 2 Mbytes.

In legacy mode, physical addresses up to 40 bits long can be translated from 32-bit virtual addresses using 32-bit paging data-structure entries when 4-Mbyte physical-page sizes are selected. In this special case, CR4.PSE=1 and CR4.PAE=0. See “4-Mbyte Page Translation” on page 127 for a description of the 4-Mbyte PDE that supports 40-bit physical-address translation. The 40-bit physical-address capability is an AMD64 architecture enhancement over the similar capability available in the legacy x86 architecture.
5.1.5 Page-Directory Page Size (PS) Bit

The page directory offset entry (PDE) and page directory pointer offset entry (PDPE) are data structures used in page translation (see Figure 5-1 on page 121). The page-size (PS) bit in the PDE (bit 7, referred to as PDE.PS) selects between standard 4-Kbyte physical-page sizes and larger (2-Mbyte or 4-Mbyte) physical-page sizes. The page-size (also PS) bit in the PDPE (bit 7, referred to as PDPE.PS) selects between 2-Mbyte and 1-Gbyte physical-page sizes in long mode.

When PDE.PS is set to 1, large physical pages are used, and the PDE becomes the lowest level of the translation hierarchy. The size of the large page is determined by the values of CR4.PAE and CR4.PSE, as shown in Figure 5-1 on page 122. When PDE.PS is cleared to 0, standard 4-Kbyte physical pages are used, and the PTE is the lowest level of the translation hierarchy.

When PDPE.PS is set to 1, 1-Gbyte physical pages are used, and the PDPE becomes the lowest level of the translation hierarchy. Neither the PDE nor PTE are used for 1-Gbyte paging.

5.2 Legacy-Mode Page Translation

Legacy mode supports two forms of translation:

- **Normal (non-PAE) Paging**—This is used when physical-address extensions are disabled (CR4.PAE=0). Entries in the page translation table are 32 bits and are used to translate 32-bit virtual addresses into physical addresses as large as 40 bits.

- **PAE Paging**—This is used when physical-address extensions are enabled (CR4.PAE=1). Entries in the page translation table are 64 bits and are used to translate 32-bit virtual addresses into physical addresses as large as 52 bits.

Legacy paging uses up to three levels of page-translation tables, depending on the paging form used and the physical-page size. Entries within each table are selected using virtual-address bit fields. The legacy page-translation tables are:

- **Page Table**—Each page-table entry (PTE) points to a physical page. If 4-Kbyte pages are used, the page table is the lowest level of the page-translation hierarchy. PTEs are not used when translating 2-Mbyte or 4-Mbyte pages.

- **Page Directory**—If 4-Kbyte pages are used, each page-directory entry (PDE) points to a page table. If 2-Mbyte or 4-Mbyte pages are used, a PDE is the lowest level of the page-translation hierarchy and points to a physical page. In non-PAE paging, the page directory is the highest level of the translation hierarchy.

- **Page-Directory Pointer**—Each page-directory pointer entry (PDPE) points to a page directory. Page-directory pointers are only used in PAE paging (CR4.PAE=1), and are the highest level in the legacy page-translation hierarchy.

The translation-table-entry formats and how they are used in the various forms of legacy page translation are described beginning on page 126.
5.2.1 CR3 Register

The CR3 register is used to point to the base address of the highest-level page-translation table. The base address is either the page-directory pointer table or the page directory table. The CR3 register format depends on the form of paging being used. Figure 5-2 on page 125 shows the CR3 format when normal (non-PAE) paging is used (CR4.PAE=0). Figure 5-3 shows the CR3 format when PAE paging is used (CR4.PAE=1).

![Figure 5-2. Control Register 3 (CR3)—Non-PAE Paging Legacy-Mode](image)

![Figure 5-3. Control Register 3 (CR3)—PAE Paging Legacy-Mode](image)

The CR3 register fields for legacy-mode paging are:

- **Table Base Address Field.** This field points to the starting physical address of the highest-level page-translation table. The size of this field depends on the form of paging used:
  - *Normal (Non-PAE) Paging (CR4.PAE=0)—* This 20-bit field occupies bits 31:12, and points to the base address of the page-directory table. The page-directory table is aligned on a 4-Kbyte boundary, with the low-order 12 address bits 11:0 assumed to be 0. This yields a total base-address size of 32 bits.
  - *PAE Paging (CR4.PAE=1)—* This field is 27 bits and occupies bits 31:5. The CR3 register points to the base address of the page-directory-pointer table. The page-directory-pointer table is aligned on a 32-byte boundary, with the low 5 address bits 4:0 assumed to be 0.

- **Page-Level Writethrough (PWT) Bit.** Bit 3. Page-level writethrough indicates whether the highest-level page-translation table has a writeback or writethrough caching policy. When PWT=0, the table has a writeback caching policy. When PWT=1, the table has a writethrough caching policy.

- **Page-Level Cache Disable (PCD) Bit.** Bit 4. Page-level cache disable indicates whether the highest-level page-translation table is cacheable. When PCD=0, the table is cacheable. When PCD=1, the table is not cacheable.

- **Reserved Bits.** Reserved fields should be cleared to 0 by software when writing CR3.
5.2.2 Normal (Non-PAE) Paging

Non-PAE paging (CR4.PAE=0) supports 4-Kbyte and 4-Mbyte physical pages, as described in the following sections.

4-Kbyte Page Translation. 4-Kbyte physical-page translation is performed by dividing the 32-bit virtual address into three fields. Each of the upper two fields is used as an index into a two-level page-translation hierarchy. The virtual-address fields are used as follows, and are shown in Figure 5-4:

- Bits 31:22 index into the 1024-entry page-directory table.
- Bits 21:12 index into the 1024-entry page table.
- Bits 11:0 provide the byte offset into the physical page.

![Figure 5-4. 4-Kbyte Non-PAE Page Translation—Legacy Mode]

Figure 5-5 on page 127 shows the format of the PDE (page-directory entry), and Figure 5-6 on page 127 shows the format of the PTE (page-table entry). Each table occupies 4 Kbytes and can hold 1024 of the 32-bit table entries. The fields within these table entries are described in “Page-Translation-Table Entry Fields” on page 141.

Figure 5-5 shows bit 7 cleared to 0. This bit is the page-size bit (PS), and specifies a 4-Kbyte physical-page translation.
4-Mbyte Page Translation.  4-Mbyte page translation is only supported when page-size extensions are enabled (CR4.PSE=1) and physical-address extensions are disabled (CR4.PAE=0).

PSE defines a page-size bit in the 32-bit PDE format (PDE.PS). This bit is used by the processor during page translation to support both 4-Mbyte and 4-Kbyte pages. 4-Mbyte pages are selected when PDE.PS is set to 1, and the PDE points directly to a 4-Mbyte physical page. PTEs are not used in a 4-Mbyte page translation. If PDE.PS is cleared to 0, or if 4-Mbyte page translation is disabled, the PDE points to a PTE.

4-Mbyte page translation is performed by dividing the 32-bit virtual address into two fields. Each field is used as an index into a single-level page-translation hierarchy. The virtual-address fields are used as follows, and are shown in Figure 5-7 on page 128:

- Bits 31:22 index into the 1024-entry page-directory table.
- Bits 21:0 provide the byte offset into the physical page.
The AMD64 architecture modifies the legacy 32-bit PDE format in PSE mode to increase physical-address size support to 40 bits. This increase in address size is accomplished by using bits 20:13 to hold eight additional high-order physical-address bits. Bit 21 is reserved and must be cleared to 0.

Figure 5-8 shows the format of the PDE when PSE mode is enabled. The physical-page base-address bits are contained in a split field. The high-order, physical-page base-address bits 39:32 are located in PDE[20:13], and physical-page base-address bits 31:22 are located in PDE[31:22].

### Figure 5-8. 4-Mbyte PDE—Non-PAE Paging Legacy-Mode

5.2.3 PAE Paging

PAE paging is used when physical-address extensions are enabled (CR4.PAE=1). PAE paging doubles the size of page-translation table entries to 64 bits so that the table entries can hold larger physical
addresses (up to 52 bits). The size of each table remains 4 Kbytes, which means each table can hold 512 of the 64-bit entries. PAE paging also introduces a third-level page-translation table, known as the page-directory-pointer table (PDP).

The size of large pages in PAE-paging mode is 2 Mbytes rather than 4 Mbytes. PAE uses the page-directory page-size bit (PDE.PS) to allow selection between 4-Kbyte and 2-Mbyte page sizes. PAE automatically uses the page-size bit, so the value of CR4.PSE is ignored by PAE paging.

**4-Kbyte Page Translation.** With PAE paging, 4-Kbyte physical-page translation is performed by dividing the 32-bit virtual address into four fields, each of the upper three fields is used as an index into a 3-level page-translation hierarchy. The virtual-address fields are described as follows and are shown in Figure 5-9:

- Bits 31:30 index into a 4-entry page-directory-pointer table.
- Bits 29:21 index into the 512-entry page-directory table.
- Bits 20:12 index into the 512-entry page table.
- Bits 11:0 provide the byte offset into the physical page.

![Figure 5-9. 4-Kbyte PAE Page Translation—Legacy Mode](image)

Figures 5-10 through 5-12 show the legacy-mode 4-Kbyte translation-table formats:
Figure 5-10 shows the PDPE (page-directory-pointer entry) format.

Figure 5-11 shows the PDE (page-directory entry) format.

Figure 5-12 shows the PTE (page-table entry) format.

The fields within these table entries are described in “Page-Translation-Table Entry Fields” on page 141.

Figure 5-11 shows the PDE.PS bit cleared to 0 (bit 7), specifying a 4-Kbyte physical-page translation.

2-Mbyte page translation is performed by dividing the 32-bit virtual address into three fields. Each field is used as an index into a 2-level page-translation hierarchy. The virtual-address fields are described as follows and are shown in Figure 5-13 on page 131:

- Bits 31:30 index into the 4-entry page-directory-pointer table.
- Bits 29:21 index into the 512-entry page-directory table.
- Bits 20:0 provide the byte offset into the physical page.

**Figure 5-13. 2-Mbyte PAE Page Translation—Legacy Mode**

Figure 5-14 shows the format of the PDPE (page-directory-pointer entry) and Figure 5-15 on page 132 shows the format of the PDE (page-directory entry). PTEs are not used in 2-Mbyte page translations.

Figure 5-15 on page 132 shows the PDE.PS bit set to 1 (bit 7), specifying a 2-Mbyte physical-page translation.

**Figure 5-14. 2-Mbyte PDPE—PAE Paging Legacy-Mode**
5.3 Long-Mode Page Translation

Long-mode page translation requires the use of physical-address extensions (PAE). Before activating long mode, PAE must be enabled by setting CR4.PAE to 1. Activating long mode before enabling PAE causes a general-protection exception (#GP) to occur.

The PAE-paging data structures support mapping of 64-bit virtual addresses into 52-bit physical addresses. PAE expands the size of legacy page-directory entries (PDEs) and page-table entries (PTEs) from 32 bits to 64 bits, allowing physical-address sizes of greater than 32 bits.

The AMD64 architecture enhances the page-directory-pointer entry (PDPE) by defining previously reserved bits for access and protection control. A new translation table is added to PAE paging, called the page-map level-4 (PML4). The PML4 table precedes the PDP table in the page-translation hierarchy.

Because PAE is always enabled in long mode, the PS bit in the page directory entry (PDE.PS) selects between 4-Kbyte and 2-Mbyte page sizes, and the CR4.PSE bit is ignored. When 1-Gbyte pages are supported, the PDPE.PS bit selects the 1-Gbyte page size.

5.3.1 Canonical Address Form

The AMD64 architecture requires implementations supporting fewer than the full 64-bit virtual address to ensure that those addresses are in canonical form. An address is in canonical form if the address bits from the most-significant implemented bit up to bit 63 are all ones or all zeros. If the addresses of all bytes in a virtual-memory reference are not in canonical form, the processor generates a general-protection exception (#GP) or a stack fault (#SS) as appropriate.

5.3.2 CR3

In long mode, the CR3 register is used to point to the PML4 base address. CR3 is expanded to 64 bits in long mode, allowing the PML4 table to be located anywhere in the 52-bit physical-address space. Figure on page 133 shows the long-mode CR3 format.
Figure 5-16. Control Register 3 (CR3)—Long Mode

The CR3 register fields for long mode are:

**Table Base Address Field.** Bits 51:12. This 40-bit field points to the PML4 base address. The PML4 table is aligned on a 4-Kbyte boundary with the low-order 12 address bits (11:0) assumed to be 0. This yields a total base-address size of 52 bits. System software running on processor implementations supporting less than the full 52-bit physical-address space must clear the unimplemented upper base-address bits to 0.

**Page-Level Writethrough (PWT) Bit.** Bit 3. Page-level writethrough indicates whether the highest-level page-translation table has a writeback or writethrough caching policy. When PWT=0, the table has a writeback caching policy. When PWT=1, the table has a writethrough caching policy.

**Page-Level Cache Disable (PCD) Bit.** Bit 4. Page-level cache disable indicates whether the highest-level page-translation table is cacheable. When PCD=0, the table is cacheable. When PCD=1, the table is not cacheable.

**Process Context Identifier.** Bits 11:0. This 12-bit field determines the current Processor Context Identifier (PCID) when CR4.PCIDE=1.

**Reserved Bits.** Reserved fields should be cleared to 0 by software when writing CR3.

### 5.3.3 4-Kbyte Page Translation

In long mode, 4-Kbyte physical-page translation is performed by dividing the virtual address into six fields. Four of the fields are used as indices into the level page-translation hierarchy. The virtual-address fields are described as follows, and are shown in Figure 5-17 on page 134:

- Bits 63:48 are a sign extension of bit 47, as required for canonical-address forms.
• Bits 47:39 index into the 512-entry page-map level-4 table.
• Bits 38:30 index into the 512-entry page-directory pointer table.
• Bits 29:21 index into the 512-entry page-directory table.
• Bits 20:12 index into the 512-entry page table.
• Bits 11:0 provide the byte offset into the physical page.

Note: The sizes of the sign extension and the PML4 fields depend on the number of virtual address
bits supported by the implementation.

Figure 5-17. 4-Kbyte Page Translation—Long Mode

Figures 5-18 through 5-20 on page 135 and Figure 5-21 on page 136 show the long-mode 4-Kbyte
translation-table formats:
• Figure 5-18 on page 135 shows the PML4E (page-map level-4 entry) format.
• Figure 5-19 on page 135 shows the PDPE (page-directory-pointer entry) format.
• Figure 5-20 on page 135 shows the PDE (page-directory entry) format.
• Figure 5-21 on page 136 shows the PTE (page-table entry) format.

The fields within these table entries are described in “Page-Translation-Table Entry Fields” on
page 141.
Figure 5-20 on page 135 shows the PDE.PS bit (bit 7) cleared to 0, indicating a 4-Kbyte physical-page translation.

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>52</th>
<th>51</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>X</td>
<td>Available</td>
<td>Page-Directory-Pointer Base Address</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(This is an architectural limit. A given implementation may support fewer bits.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>12</td>
<td>11</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Page-Directory-Pointer Base Address</td>
<td>AVL</td>
<td>M</td>
<td>B</td>
<td>M</td>
</tr>
</tbody>
</table>

**Figure 5-18. 4-Kbyte PML4E—Long Mode**

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>52</th>
<th>51</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>X</td>
<td>Available</td>
<td>Page-Directory Base Address</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(This is an architectural limit. A given implementation may support fewer bits.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>12</td>
<td>11</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Page-Directory Base Address</td>
<td>AVL</td>
<td>I</td>
<td>G</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 5-19. 4-Kbyte PDPE—Long Mode**

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>52</th>
<th>51</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>X</td>
<td>Available</td>
<td>Page-Table Base Address</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(This is an architectural limit. A given implementation may support fewer bits.)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>12</td>
<td>11</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>Page-Table Base Address</td>
<td>AVL</td>
<td>I</td>
<td>G</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 5-20. 4-Kbyte PDE—Long Mode**
In long mode, 2-Mbyte physical-page translation is performed by dividing the virtual address into five fields. Three of the fields are used as indices into the level page-translation hierarchy. The virtual-address fields are described as follows, and are shown in Figure 5-22:

- Bits 63:48 are a sign extension of bit 47 as required for canonical address forms.
- Bits 47:39 index into the 512-entry page-map level-4 table.
- Bits 38:30 index into the 512-entry page-directory-pointer table.
- Bits 29:21 index into the 512-entry page-directory table.
- Bits 20:0 provide the byte offset into the physical page.
Figures 5-23 through 5-25 on page 138 show the long-mode 2-Mbyte translation-table formats (the PML4 and PDPT formats are identical to those used for 4-Kbyte page translations and are repeated here for clarity):

- Figure 5-23 on page 138 shows the PML4E (page-map level-4 entry) format.
- Figure 5-24 on page 138 shows the PDPE (page-directory-pointer entry) format.
- Figure 5-25 on page 138 shows the PDE (page-directory entry) format.

The fields within these table entries are described in “Page-Translation-Table Entry Fields” on page 141. PTEs are not used in 2-Mbyte page translations.

Figure 5-25 shows the PDE.PS bit (bit 7) set to 1, indicating a 2-Mbyte physical-page translation.
5.3.5 1-Gbyte Page Translation

In long mode, 1-Gbyte physical-page translation is performed by dividing the virtual address into four fields. Two of the fields are used as indices into the level page-translation hierarchy. The virtual-address fields are described as follows, and are shown in Figure 5-26 on page 139:

- Bits 63:48 are a sign extension of bit 47 as required for canonical address forms.
• Bits 47:39 index into the 512-entry page-map level-4 table.
• Bits 38:30 index into the 512-entry page-directory-pointer table.
• Bits 29:0 provide the byte offset into the physical page.

Figure 5-26. 1-Gbyte Page Translation—Long Mode

Figure 5-27 and Figure 5-28 on page 140 show the long mode 1-Gbyte translation-table formats (the PML4 format is identical to the one used for 4-Kbyte page translations and is repeated here for clarity):

• Figure 5-27 shows the PML4E (page-map level-4 entry) format.
• Figure 5-28 shows the PDPE (page-directory-pointer entry) format.

The fields within these table entries are described in “Page-Translation-Table Entry Fields” on page 141 in the current volume. PTEs and PDEs are not used in 1-Gbyte page translations.

Figure 5-28 on page 140 shows the PDPE.PS bit (bit 7) set to 1, indicating a 1-Gbyte physical-page translation.
1-Gbyte Paging Feature Identification. EDX bit 26 as returned by CPUID function 8000_0001h indicates 1-Gbyte page support. The EAX register as returned by CPUID function 8000_0019h reports the number of 1-Gbyte L1 TLB entries supported and EBX reports the number of 1-Gbyte L2 TLB entries. For more information using the CPUID instruction see Section 3.3 “Processor Feature Identification” on page 64.
5.4 Page-Translation-Table Entry Fields

The page-translation-table entries contain control and informational fields used in the management of the virtual-memory environment. Most fields are common across all translation table entries and modes and occupy the same bit locations. However, some fields are located in different bit positions depending on the page translation hierarchical level, and other fields have different sizes depending on which physical-page size, physical-address size, and operating mode are selected. Although these fields can differ in bit position or size, their meaning is consistent across all levels of the page translation hierarchy and in all operating modes.

5.4.1 Field Definitions

The following sections describe each field within the page-translation table entries.

Translation-Table Base Address Field. The translation-table base-address field points to the physical base address of the next-lower-level table in the page-translation hierarchy. Page data-structure tables are always aligned on 4-Kbyte boundaries, so only the address bits above bit 11 are stored in the translation-table base-address field. Bits 11:0 are assumed to be 0. The size of the field depends on the mode:

- In normal (non-PAE) paging (CR4.PAE=0), this field specifies a 32-bit physical address.
- In PAE paging (CR4.PAE=1), this field specifies a 52-bit physical address.

52 bits correspond to the maximum physical-address size allowed by the AMD64 architecture. If a processor implementation supports fewer than the full 52-bit physical address, software must clear the unimplemented high-order translation-table base-address bits to 0. For example, if a processor implementation supports a 40-bit physical-address size, software must clear bits 51:40 when writing a translation-table base-address field in a page data-structure entry.

Physical-Page Base Address Field. The physical-page base-address field points to the base address of the translated physical page. This field is found only in the lowest level of the page-translation hierarchy. The size of the field depends on the mode:

- In normal (non-PAE) paging (CR4.PAE=0), this field specifies a 32-bit base address for a physical page.
- In PAE paging (CR4.PAE=1), this field specifies a 52-bit base address for a physical page.

Physical pages can be 4 Kbytes, 2 Mbytes, 4 Mbytes, or 1-Gbyte and they are always aligned on an address boundary corresponding to the physical-page length. For example, a 2-Mbyte physical page is always aligned on a 2-Mbyte address boundary. Because of this alignment, the low-order address bits are assumed to be 0, as follows:

- 4-Kbyte pages, bits 11:0 are assumed 0.
- 2-Mbyte pages, bits 20:0 are assumed 0.
- 4-Mbyte pages, bits 21:0 are assumed 0.
- 1-Gbyte pages, bits 29:0 are assumed 0.
**Present (P) Bit.** Bit 0. This bit indicates whether the page-translation table or physical page is loaded in physical memory. When the P bit is cleared to 0, the table or physical page is not loaded in physical memory. When the P bit is set to 1, the table or physical page is loaded in physical memory.

Software clears this bit to 0 to indicate a page table or physical page is not loaded in physical memory. A page-fault exception (#PF) occurs if an attempt is made to access a table or page when the P bit is 0. System software is responsible for loading the missing table or page into memory and setting the P bit to 1.

When the P bit is 0, indicating a not-present page, all remaining bits in the page data-structure entry are available to software.

Entries with P cleared to 0 are never cached in TLB nor will the processor set the Accessed or Dirty bit for the table entry.

**Read/Write (R/W) Bit.** Bit 1. This bit controls read/write access to all physical pages mapped by the table entry. For example, a page-map level-4 R/W bit controls read/write access to all 128M \((512 \times 512 \times 512)\) physical pages it maps through the lower-level translation tables. When the R/W bit is cleared to 0, access is restricted to read-only. When the R/W bit is set to 1, both read and write access is allowed. See “Page-Protection Checks” on page 149 for a description of the paging read/write protection mechanism.

**User/Supervisor (U/S) Bit.** Bit 2. This bit controls user (CPL 3) access to all physical pages mapped by the table entry. For example, a page-map level-4 U/S bit controls the access allowed to all 128M \((512 \times 512 \times 512)\) physical pages it maps through the lower-level translation tables. When the U/S bit is cleared to 0, access is restricted to supervisor level (CPL 0, 1, 2). When the U/S bit is set to 1, both user and supervisor access is allowed. See “Page-Protection Checks” on page 149 for a description of the paging user/supervisor protection mechanism.

**Page-Level Writethrough (PWT) Bit.** Bit 3. This bit indicates whether the page-translation table or physical page to which this entry points has a writeback or writethrough caching policy. When the PWT bit is cleared to 0, the table or physical page has a writeback caching policy. When the PWT bit is set to 1, the table or physical page has a writethrough caching policy. See “Memory Caches” on page 185 for additional information on caching.

**Page-Level Cache Disable (PCD) Bit.** Bit 4. This bit indicates whether the page-translation table or physical page to which this entry points is cacheable. When the PCD bit is cleared to 0, the table or physical page is cacheable. When the PCD bit is set to 1, the table or physical page is not cacheable. See “Memory Caches” on page 185 for additional information on caching.

**Accessed (A) Bit.** Bit 5. This bit indicates whether the page-translation table or physical page to which this entry points has been accessed. The A bit is set to 1 by the processor the first time the table or physical page is either read from or written to. The A bit is never cleared by the processor. Instead, software must clear this bit to 0 when it needs to track the frequency of table or physical-page accesses.

**Dirty (D) Bit.** Bit 6. This bit is only present in the lowest level of the page-translation hierarchy. It indicates whether the physical page to which this entry points has been written. The D bit is set to 1 by
the processor the first time there is a write to the physical page. The D bit is never cleared by the processor. Instead, software must clear this bit to 0 when it needs to track the frequency of physical-page writes.

**Page Size (PS) Bit.** Bit 7. This bit is present in page-directory entries and long-mode page-directory-pointer entries. When the PS bit is set in the page-directory-pointer entry (PDPE) or page-directory entry (PDE), that entry is the lowest level of the page-translation hierarchy. When the PS bit is cleared to 0 in all levels above PTE, the lowest level of the page-translation hierarchy is the page-table entry (PTE), and the physical-page size is 4 Kbytes. The physical-page size is determined as follows:

- If EFER.LMA=1 and PDPE.PS=1, the physical-page size is 1 Gbyte.
- If CR4.PAE=0 and PDE.PS=1, the physical-page size is 4 Mbytes.
- If CR4.PAE=1 and PDE.PS=1, the physical-page size is 2 Mbytes.

See Table 5-1 on page 122 for a description of the relationship between the PS bit, PAE, physical-page sizes, and page-translation hierarchy.

**Global Page (G) Bit.** Bit 8. This bit is only present in the lowest level of the page-translation hierarchy. It indicates the physical page is a global page. The TLB entry for a global page (G=1) is not invalidated when CR3 is loaded either explicitly by a MOV CRn instruction or implicitly during a task switch. Use of the G bit requires the page-global enable bit in CR4 to be set to 1 (CR4.PGE=1). See “Global Pages” on page 146 for more information on the global-page mechanism.

**Available to Software (AVL) Bit.** These bits are not interpreted by the processor and are available for use by system software.

**Page-Attribute Table (PAT) Bit.** This bit is only present in the lowest level of the page-translation hierarchy, as follows:

- If the lowest level is a PTE (PDE.PS=0), PAT occupies bit 7.
- If the lowest level is a PDE (PDE.PS=1) or PDPE (PDPE.PS=1), PAT occupies bit 12.

The PAT bit is the high-order bit of a 3-bit index into the PAT register (Figure 7-10 on page 204). The other two bits involved in forming the index are the PCD and PWT bits. Not all processors support the PAT bit by implementing the PAT registers. See “Page-Attribute Table Mechanism” on page 204 for a description of the PAT mechanism and how it is used.

**Memory Protection Key (MPK) Bits.** Bits 62:59. When Memory Protection Keys are enabled (CR4.PKE=1), this 4-bit field selects the memory protection key for the physical page mapped by this entry. Ignored if memory protection keys are disabled (CR4.PKE=0). (See “Memory Protection Keys (MPK) Bit” on page 151 for a description of this mechanism.)

**No Execute (NX) Bit.** Bit 63. This bit is present in the translation-table entries defined for PAE paging, with the exception that the legacy-mode PDPE does not contain this bit. This bit is not supported by non-PAE paging.
The NX bit can only be set when the no-execute page-protection feature is enabled by setting EFER.NXE to 1 (see “Extended Feature Enable Register (EFER)” on page 55). If EFER.NXE=0, the NX bit is treated as reserved. In this case, a page-fault exception (#PF) occurs if the NX bit is not cleared to 0.

This bit controls the ability to execute code from all physical pages mapped by the table entry. For example, a page-map level-4 NX bit controls the ability to execute code from all 128M (512 × 512 × 512) physical pages it maps through the lower-level translation tables. When the NX bit is cleared to 0, code can be executed from the mapped physical pages. When the NX bit is set to 1, code cannot be executed from the mapped physical pages. See “No Execute (NX) Bit” on page 143 for a description of the no-execute page-protection mechanism.

**Reserved Bits.** Software should clear all reserved bits to 0. If the processor is in long mode, or if page-size and physical-address extensions are enabled in legacy mode, a page-fault exception (#PF) occurs if reserved bits are not cleared to 0.

### 5.4.2 Notes on Accessed and Dirty Bits

The processor never sets the Accessed bit or the Dirty bit for a not present page (P = 0). The ordering of Accessed and Dirty bit updates with respect to surrounding loads and stores is discussed below.

**Accessed (A) Bit.** The Accessed bit can be set for instructions that are speculatively executed by the processor.

For example, the Accessed bit may be set by instructions in a mispredicted branch path even though those instructions are never retired. Thus, software must not assume that the TLB entry has not been cached in the TLB, just because no instruction that accessed the page was successfully retired. Nevertheless, a table entry is never cached in the TLB without its Accessed bit being set at the same time.

The processor does not order Accessed bit updates with respect to loads done by other instructions.

**Dirty (D) Bit.** The Dirty bit is not updated speculatively. For instructions with multiple writes, the D bit may be set for any writes completed up to the point of a fault. In rare cases, the Dirty bit may be set even if a write was not actually performed, including MASKMOVQ with a mask of zero and certain x87 floating point instructions that cause an exception. Thus software can not assume that the page has actually been written even where PTE[D] is set to 1.

If PTE[D] is cleared to 0, software can rely on the fact that the page has not been written.

In general, Dirty bit updates are ordered with respect to other loads and stores, although not necessarily with respect to accesses to WC memory; in particular, they may not cause WC buffers to be flushed. However, to ensure compatibility with future processors, a serializing operation should be inserted before reading the D bit.
5.5 Translation-Lookaside Buffer (TLB)

When paging is enabled, every memory access has its virtual address automatically translated into a physical address using the page-translation hierarchy. Translation-lookaside buffers (TLBs), also known as page-translation caches, nearly eliminate the performance penalty associated with page translation. TLBs are special on-chip caches that hold the most-recently used virtual-to-physical address translations. Each memory reference (instruction and data) is checked by the TLB. If the translation is present in the TLB, it is immediately provided to the processor, thus avoiding external memory references for accessing page tables.

TLBs take advantage of the principle of locality. That is, if a memory address is referenced, it is likely that nearby memory addresses will be referenced in the near future. In the context of paging, the proximity of memory addresses required for locality can be broad—it is equal to the page size. Thus, it is possible for a large number of addresses to be translated by a small number of page translations. This high degree of locality means that almost all translations are performed using the on-chip TLBs.

System software is responsible for managing the TLBs when updates are made to the linear-to-physical mapping of addresses. A change to any paging data-structure entry is not automatically reflected in the TLB, and hardware snooping of TLBs during memory-reference cycles is not performed. Software must invalidate the TLB entry of a modified translation-table entry so that the change is reflected in subsequent address translations. TLB invalidation is described in “TLB Management” on page 146. Only privileged software running at CPL=0 can manage the TLBs.

5.5.1 Process Context Identifier

The Process Context Identifier (PCID) feature allows a logical processor to cache TLB mappings concurrently for multiple virtual address spaces. When enabled (by setting CR4.PCIDE=1), the processor associates the current 12-bit PCID with each TLB mapping it creates. Only entries matching the current PCID are used when performing address translations. In this way, the processor may retain cached TLB mappings for multiple contexts.

The current PCID is the value in CR3[11:0]. When PCIDs are enabled the system software can store 12-bit Process Context Identifiers in CR3 for different address spaces. Subsequently, when system software switches address spaces (by writing the page table base pointer in CR3[62:12]), the processor may use TLB mappings previously stored for that address space and PCID. A MOV to CR4 that clears CR4.PCIDE causes all cached entries in the TLB for the logical processor to be invalidated. When PCIDs are not enabled (CR4.PCIDE=0) the current PCID is always zero and all TLB mappings are associated with PCID=0.

Attempting to set CR4.PCIDE with a MOV to CR4 if EFER.LMA = 0 or CR3[11:0] <> 0 causes in a #GP exception. Attempting to clear CR0.PG with a MOV to CR0 if CR4.PCIDE is set causes a #GP exception. The presence of PCID functionality is indicated by CPUID Function 1, ECX[PCID]=1.
5.5.2 Global Pages

The processor invalidates the TLB whenever CR3 is loaded either explicitly or implicitly. After the TLB is invalidated, subsequent address references can consume many clock cycles until their translations are cached as new entries in the TLB. Invalidation of TLB entries for frequently-used or critical pages can be avoided by specifying the translations for those pages as global. TLB entries for global pages are not invalidated as a result of a CR3 load. Global pages are invalidated using the INVLPG instruction.

Global-page extensions are controlled by setting and clearing the PGE bit in CR4 (bit 7). When CR4.PGE is set to 1, global-page extensions are enabled. When CR4.PGE is cleared to 0, global-page extensions are disabled. When CR4.PGE=1, setting the global (G) bit in the translation-table entry marks the page as global.

The INVLPG instruction ignores the G bit and can be used to invalidate individual global-page entries in the TLB. To invalidate all entries, including global-page entries, disable global-page extensions (CR4.PGE=0).

5.5.3 TLB Management

Generally, unless system software modifies the linear-to-physical address mapping, the processor manages the TLB transparently to software. This includes allocating entries and replacing old entries with new entries. In general, software changes made to paging-data structures are not automatically reflected in the TLB. In these situations, it is necessary for software to invalidate TLB entries so that these changes will be propagated to the page-translation mechanism.

TLB entries can be explicitly invalidated using operations intended for that purpose or implicitly invalidated as a result of another operation. TLB invalidation has no effect on the associated page-translation tables in memory.

**Explicit Invalidations.** Three mechanisms are provided to explicitly invalidate the TLB:

- The *Invalidate TLB Entry* instruction (INVLPG) can be used to invalidate a specific entry within the TLB. This instruction invalidates an entry regardless of whether it is marked as global or not.

- The *Invalidate TLB entry in a Specified ASID* instruction (INVLPGA) operates similarly, but operates only on entries associated with the specified ASID. See “Invalidate Page, Alternate ASID” on page 484.

- The *Invalidate TLB with Broadcast* instruction (INVLPGB) can be used to invalidate a specified range of TLB entries on the local processor and broadcast the invalidation operation to remote processors. See INVLPGB in Volume 3.

- The *Invalidate TLB entries in Specified PCID* instruction (INVPCID) can be used to invalidate TLB entries of the specified Processor Context ID. See INVPCID in Volume 3.

- Updates to the CR3 register cause the entire TLB to be invalidated *except* for global pages. The CR3 register can be updated with the MOV CR3 instruction. CR3 is also updated during a task switch, with the updated CR3 value read from the TSS of the new task.
• The TLB_CONTROL field of a VMCB can request specific flushes of the TLB to occur when the VMRUN instruction is executed on that VMCB. See “TLB Flush” on page 483.

Implicit Invalidations. The following operations cause the entire TLB to be invalidated, including global pages:
• Modifying the CR0.PG bit (paging enable).
• Modifying the CR4.PAE bit (physical-address extensions), the CR4.PSE bit (page-size extensions), or the CR4.PGE bit (page-global enable).
• Entering SMM as a result of an SMI interrupt.
• Executing the RSM instruction to return from SMM.
• Updating a memory-type range register (MTRR) with the WRMSR instruction.
• External initialization of the processor.
• External masking of the A20 address bit (asserting the A20M# input signal).
• Writes to certain model-specific registers with the WRMSR instruction; see the BIOS and Kernel Developer’s Guide (BKDG) or Processor Programming Reference Manual applicable to your product for more information.
• A MOV to CR4 that changes CR4.PKE from 0 to 1.
• A MOV to CR4 that clears CR4.PCIDE from 1 to 0.

Invalidation of Table Entry Upgrades. If a table entry is updated to remove a page access constraint, such as removing supervisor, read-only, and/or no-execute restrictions, an invalidation is not required because the hardware will automatically detect the changes. If a table entry is updated and does not remove a permission violation, it is unpredictable whether the old or updated entry will be used until an invalidation is performed.

Speculative Caching of Address Translations. For performance reasons, AMD64 processors may speculatively load valid address translations into the TLB on false execution paths. Such translations are not based on references that a program makes from an “architectural state” perspective, but which the processor may make in speculatively following an instruction path which turns out to be mispredicted. In general, the processor may create a TLB entry for any linear address for which valid entries exist in the page table structure currently pointed to by CR3. This may occur for both instruction fetches and data references. Such entries remain cached in the TLBs and may be used in subsequent translations. Loading a translation speculatively will set the Accessed bit, if not already set. A translation will not be loaded speculatively if the Dirty bit needs to be set.

Caching of Upper Level Translation Table Entries. Similarly, to improve the performance of table walks on TLB misses, AMD64 processors may save upper level translation table entries in special table walk caching structures which are kept coherent with the tables in memory via the same mechanisms as the TLBs—by means of the INVLPG instruction, moves to CR3, and modification of paging control bits in CR0 and CR4. Like address translations in the TLB, these upper level entries may also be cached speculatively and by false-path execution. These entries are never cached if their P (present) bits are set to 0.
Under certain circumstances, an upper-level table entry that cannot ultimately lead to a valid translation (because there are no valid entries in the lower level table to which it points) may also be cached. This can happen while executing down a false path, when an in-progress table walk gets cancelled by the branch mispredict before the low level table entry that would cause a fault is encountered. Said another way, the fact that a page table has no valid entries does not guarantee that upper level table entries won't be accessed and cached in the processor, as long as those upper level entries are marked as present. For this reason, it is not safe to modify an upper level entry, even if no valid lower-level entries exist, without first clearing its present bit, followed by an INVLPG instruction.

**Use of Cached Entries When Reporting a Page Fault Exception.** On current AMD64 processors, when any type of page fault exception is encountered by the MMU, any cached upper-level entries that lead to the faulting entry are flushed (along with the TLB entry, if already cached) and the table walk is repeated to confirm the page fault using the table entries in memory. This is done because a table entry is allowed to be upgraded (by marking it as present, or by removing its write, execute or supervisor restrictions) without explicitly maintaining TLB coherency. Such an upgrade will be found when the table is re-walked, which resolves the fault. If the fault is confirmed on the re-walk however, a page fault exception is reported, and upper level entries that may have been cached on the re-walk are flushed.

**Handling of D-Bit Updates.** When the processor needs to set the D bit in the PTE for a TLB entry that is already marked as writable at all cached TLB levels, the table walk that is performed to access the PTE in memory may use cached upper level table entries. This differs from the fault situation previously described, in which cached entries aren't used to confirm the fault during the table walk.

**Invalidation of Cached Upper-level Entries by INVLPG.** The effect of INVLPG on TLB caching of upper-level page table entries is controlled by EFER[TCE] on processors that support the translation cache extension feature. If EFER[TCE] is 0, or if the processor does not support the translation cache extension feature, an INVLPG will flush all upper-level page table entries in the TLB as well as the target PTE. If EFER[TCE] is 1, INVLPG will flush only those upper-level entries that lead to the target PTE, along with the target PTE itself. INVLPGA may flush all upper-level entries regardless of the state of TCE. For further details, see Section 3.1.7 “Extended Feature Enable Register (EFER)” on page 55.

**Handling of PDPT Entries in PAE Mode.** When 32-bit PAE mode is enabled on AMD64 processors (CR4.PAE is set to 1) a third level of the address translation table hierarchy, the page directory pointer table (PDPT), is enabled. This table contains four entries. On current AMD64 processors, in native mode, these four entries are unconditionally loaded into the table walk cache whenever CR3 is written with the PDPT base address, and remain locked in. At this point they are also checked for reserved bit violations, and if such violations are present a general-protection exception (#GP) occurs.

Under SVM, however, when the processor is in guest mode with PAE enabled, the guest PDPT entries are not cached or validated at this point, but instead are loaded and checked on demand in the normal course of address translation, just like page directory and page table entries. Any reserved bit violations are detected at the point of use, and result in a page-fault (#PF) exception rather than a
general-protection (#GP) exception. The cached PDPT entries are subject to displacement from the
table walk cache and reloading from the PDPT, hence software must assume that the PDPT entries
may be read by the processor at any point while those tables are active. Future AMD processors may
implement this same behavior in native mode as well, rather than pre-loading the PDPT entries.

5.6 Page-Protection Checks

The AMD64 architecture provides five forms of page-level memory protection. The first form of
protection prevents non-privileged (user) code from accessing privileged (supervisor) code and data.
The second form of protection prevents writes into read-only address spaces. Two forms of page-level
memory protection prevent the processor from fetching instructions from pages that are either known
to contain non-executable data or that are accessible by user-mode code. The remaining form
(Memory Protection Keys) allows an application to manage page-based data access protections from
user mode.

Access protection checks are performed when a virtual address is translated into a physical address.
For those checks, the processor examines the page-level memory-protection bits in the translation
tables to determine if the access is allowed. The page table bits involved in these checks are:

• User/Supervisor (U/S)—See “User/Supervisor (U/S) Bit” on page 142.
• Read/Write (R/W)—See “Read/Write (R/W) Bit” on page 142.
• No-Execute (NX)—See “No Execute (NX) Bit” on page 143.
• Memory Protection Key (MPK)—See “Memory Protection Keys (MPK) Bit” on page 151.

Access protection actions taken by the processor are controlled by the following bits:

• Write-Protect enable (CR0.WP)—See “Write Protect (WP) Bit” on page 44.
• No-Execute Enable (EFER.NXE)—See “No-Execute Enable (NXE) Bit” on page 57.
• Supervisor-mode Execution Prevention enable (CR4.SMEP)—See “Supervisor Mode Execution
Prevention (SMEP)” on page 51.
• Protection Key Enable (CR4.PKE)—See “Protected-Mode Enable (PE) Bit” on page 43

These protection checks are available at all levels of the page-translation hierarchy.

5.6.1 User/Supervisor (U/S) Bit

The U/S bit in the page-translation tables determines the privilege level required to access the page. If
U/S=0, the page is considered a user page and if U/S=1 the page is considered a supervisor page.
Conceptually, user (non-privileged) pages correspond to a current privilege-level (CPL) of 3, or least-
privileged. Supervisor (privileged) pages correspond to a CPL of 0, 1, or 2, all of which are jointly
regarded as most-privileged.

When the processor is running at a CPL of 0, 1, or 2, it can access both user and supervisor pages.
However, when the processor is running at a CPL of 3, it can only access user pages. If an attempt is
made to access a supervisor page while the processor is running at CPL = 3, a page-fault exception (#PF) occurs.

See “Privilege-Level Concept” on page 98 for more information on processor privilege levels.

### 5.6.2 Read/Write (R/W) Bit

The R/W bit in the page-translation tables specifies the access type allowed for the page. If R/W=1, the page is read/write. If R/W = 0, the page is read-only. A page-fault exception (#PF) occurs if an attempt is made by user software to write to a read-only page. If supervisor software attempts to write a read-only page, the outcome depends on the value of the CR0.WP bit (described below).

### 5.6.3 No Execute (NX) Bit

The NX bit provides the ability to mark a page as non-executable. If the NX bit is set at any level of the page-table hierarchy in the table entries traversed during a table walk, the page mapped by those entries is a no-execute page. When no-execute protection is enabled, any attempt to fetch an instruction from a no-execute page results in a page-fault exception (#PF).

The no-execute protection check applies to all privilege levels. It does not distinguish between supervisor and user-level accesses.

The no-execute protection feature is supported only in PAE-paging mode. In 32-bit PAE mode, the NX bit is not supported at the Page Directory Pointer table level. In this mode, the value of the NX bit at the PDP level defaults to 0.

No-execute protection is enabled by setting the NXE bit in the EFER register to 1. Before setting this bit, system software must verify the processor supports the no-execute feature by checking the CPUID NX feature flag (CPUID Fn8000_0001_EDX[NX]).

### 5.6.4 Write Protect (CR0.WP) Bit

The ability to write to read-only pages is governed by the processor mode and whether write protection is enabled. If write protection is not enabled, a processor running at CPL 0, 1, or 2 can write to any physical page, even if it is marked as read-only. Enabling write protection by setting the WP bit in CR0 prevents supervisor code from writing into read-only pages, including read-only user-level pages.

A page-fault exception (#PF) occurs if software attempts to write (at any privilege level) into a read-only page while write protection is enabled.

### 5.6.5 Supervisor-Mode Execution Prevention (CR4.SMEP) Bit

When supported and enabled, a page-fault exception (#PF) is asserted if the processor attempts to fetch an instruction from a user page while running at CPL 0, 1, or 2. A user page is any page with the U/S bit set to 1, and thus accessible when the processor is running at CPL = 3.
Supervisor-mode execution prevention is enabled by setting the SMEP bit (bit 20) in the CR4 register to 1. Before setting this bit, system software must verify the processor supports the SMEP feature by checking the SMEP feature flag (CPUID Fn0000_0007_EBX[SMEP]_x0 = 1).

For more information using the CPUID instruction see Section 3.3 “Processor Feature Identification” on page 64.

5.6.6 Memory Protection Keys (MPK) Bit

The Memory Protection Key (MPK) feature provides a way for applications to impose page-based data access protections (read/write, read-only or no access), without requiring modification of page tables and subsequent TLB invalidations when the application changes protection domains.

When MPK is enabled (CR4.PKE=1), a protection key is located in bits 62:59 of final page table entry mapping each virtual address. This 4-bit protection key is used as an index (i) into the user-accessible PKRU register which contains 16 access-disable/write-disable (WDi/ADi) pairs.

![Figure 5-29. PKRU Register](image)

The WDi/ADi pairs operate as follows:

If ADi=0, data access is permitted

- If ADi=1, no data access is permitted (regardless of CPL)
- If WDi == 0, write access is allowed
- If WDi == 1: User-mode write access is not allowed. Supervisor access is controlled by CR0.WP:
  - If CR0.WP=1, supervisor-mode writes are not allowed
  - If CR0.WP=0, supervisor-mode writes are allowed

Software can use the RDPKRU and WRPKRU instructions to read and write the PKRU register. These instructions are not privileged and can be used in user mode or in supervisor mode.

The MPK mechanism is ignored in the following cases:

- if CR4.PKE=0
- if long mode is disabled (EFER.LMA=0)
- for instruction fetches
- for pages marked in the paging structures as read-only (R/W=0) or as supervisor addresses (U/S=0)
5.7 Protection Across Paging Hierarchy

The privilege level and access type specified at each level of the page-translation hierarchy have a combined effect on the protection of the translated physical page. Enabling and disabling write protection via CR0.WP further qualifies the protection effect on the physical page.

Table 5-2 shows the overall effect that privilege level and access type have on physical-page protection when write protection is disabled (CR0.WP=0). In this case, when any translation-table entry is specified as supervisor level, the physical page is a supervisor page and can only be accessed by software running at CPL 0, 1, or 2. Such a page allows read/write access even if all levels of the page-translation hierarchy specify read-only access.

Table 5-2. Physical-Page Protection, CR0.WP=0

<table>
<thead>
<tr>
<th>Page-Map Level-4 Entry</th>
<th>Page-Directory-Pointer Entry</th>
<th>Page-Directory Entry</th>
<th>Page-Table Entry</th>
<th>Effective Result on Physical Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>U/S</td>
<td>R/W</td>
<td>U/S</td>
<td>R/W</td>
<td>U/S</td>
</tr>
<tr>
<td>S</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>S</td>
</tr>
<tr>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>S</td>
</tr>
<tr>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>U</td>
<td>R</td>
<td>U</td>
<td>—</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>—</td>
<td>U</td>
<td>R</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>R</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>R/W</td>
<td>U</td>
<td>R/W</td>
<td>U</td>
</tr>
</tbody>
</table>

Note:
- S = Supervisor Level (CPL=0, 1, or 2), U = User Level (CPL = 3), R = Read-Only Access, R/W = Read/Write Access, — = Don’t Care.

If all table entries in the translation hierarchy are specified as user level the physical page is a user page, and both supervisor and user software can access it. In this case the physical page is read-only if any table entry in the translation hierarchy specifies read-only access. All table entries in the translation hierarchy must specify read/write access for the physical page to be read/write.

Table 5-3 shows the overall effect that privilege level and access type have on physical-page access when write protection is enabled (CR0.WP=1). When any translation-table entry is specified as supervisor level, the physical page is a supervisor page and can only be accessed by supervisor software. In this case, the physical page is read-only if any table entry in the translation hierarchy specifies read-only access. All table entries in the translation hierarchy must specify read/write access for the supervisor page to be read/write.
Table 5-3. Effect of CR0.WP=1 on Supervisor Page Access

<table>
<thead>
<tr>
<th>Page-Map Level-4 Entry</th>
<th>Page Directory-Pointer Entry</th>
<th>Page Directory Entry</th>
<th>Page Table Entry</th>
<th>Physical Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
<td>R/W</td>
</tr>
<tr>
<td>R</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>R</td>
</tr>
<tr>
<td>—</td>
<td>R</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>—</td>
<td>R</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>—</td>
<td>—</td>
<td>—</td>
<td>R</td>
<td>—</td>
</tr>
<tr>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
</tr>
</tbody>
</table>

Note:  
R = Read-Only Access Type, W = Read/Write Access Type, — = Don’t Care.  
Physical page is in supervisor mode, as determined by U/S settings in Table 5-2.

5.7.1 Access to User Pages when CR0.WP=1

As shown in Table 5-2 on page 152, read/write access to user-level pages behaves the same as when write protection is disabled (CR0.WP=0), with one critical difference. When write protection is enabled, supervisor programs cannot write into read-only user pages.

5.8 Effects of Segment Protection

Segment-protection and page-protection checks are performed serially by the processor, with segment-privilege checks performed first, followed by page-protection checks. Page-protection checks are not performed if a segment-protection violation is found. If a violation is found during either segment-protection or page-protection checking, an exception occurs and no memory access is performed. Segment-protection violations cause either a general-protection exception (#GP) or a stack exception (#SS) to occur. Page-protection violations cause a page-fault exception (#PF) to occur.
6 System Instructions

System instructions provide control over the resources used to manage the processor operating environment. This includes memory management, memory protection, task management, interrupt and exception handling, system-management mode, software debug and performance analysis, and model-specific features. Most instructions used to access these resources are privileged and can only be executed while the processor is running at CPL=0, although some instructions can be executed at any privilege level.

Table 6-1 summarizes the instructions used for system management. These include all privileged instructions, instructions whose privilege requirement is under the control of system software, non-privileged instructions that are used primarily by system software, and instructions used to transfer control to system software. Most of the instructions listed in Table 6-1 are summarized in this chapter, although a few are introduced elsewhere in this manual, as indicated in the Reference column of Table 6-1.

For details on individual system instructions, see “System Instruction Reference” in Volume 3.

### Table 6-1. System Management Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Privilege</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CPL=0</td>
<td>O/S¹</td>
</tr>
<tr>
<td>ARPL</td>
<td>Adjust Requestor Privilege Level</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CLGI</td>
<td>Clear Global Interrupt Flag</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CLI</td>
<td>Clear Interrupt Flag</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CLTS</td>
<td>Clear Task-Switched Flag in CR0</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>HLT</td>
<td>Halt</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>INT3</td>
<td>Interrupt to Debug Vector</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>INVD</td>
<td>Invalidate Caches</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>INVLPGB</td>
<td>Invalidate TLB Entries with Broadcast</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>INVLPGA</td>
<td>Invalidate TLB Entry in a Specified ASID</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>INVLPCID</td>
<td>Invalidate TLB Entries in Specified Processor Context</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

**Note:**
1. The operating system controls the privilege required to use the instruction.
Table 6-1. System Management Instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Privilege</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRETx</td>
<td>Interrupt Return (all forms)</td>
<td>X</td>
<td>“Returning From Interrupt Procedures” on page 252</td>
</tr>
<tr>
<td>LAR</td>
<td>Load Access-Rights Byte</td>
<td>X</td>
<td>“Checking Access Rights” on page 164</td>
</tr>
<tr>
<td>LGDT</td>
<td>Load Global-Descriptor-Table Register</td>
<td>X</td>
<td>“LGDT and LIDT Instructions” on page 164</td>
</tr>
<tr>
<td>LIDT</td>
<td>Load Interrupt-Descriptor-Table Register</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>LLDT</td>
<td>Load Local-Descriptor-Table Register</td>
<td>X</td>
<td>“LLDT and LTR Instructions” on page 164</td>
</tr>
<tr>
<td>LMSW</td>
<td>Load Machine-Status Word</td>
<td>X</td>
<td>“LMSW and SMSW Instructions” on page 162</td>
</tr>
<tr>
<td>LSL</td>
<td>Load Segment Limit</td>
<td>X</td>
<td>“Checking Segment Limits” on page 165</td>
</tr>
<tr>
<td>LTR</td>
<td>Load Task Register</td>
<td>X</td>
<td>“LLDT and LTR Instructions” on page 164</td>
</tr>
<tr>
<td>MONITOR</td>
<td>Setup Monitor Address</td>
<td>X</td>
<td>--</td>
</tr>
<tr>
<td>MOV CRₙ</td>
<td>Move to/from Control Registers</td>
<td>X</td>
<td>“MOV CRₙ Instructions” on page 162</td>
</tr>
<tr>
<td>MOV DRₙ</td>
<td>Move to/from Debug Registers</td>
<td>X</td>
<td>“Accessing Debug Registers” on page 163</td>
</tr>
<tr>
<td>MWAIT</td>
<td>Monitor Wait</td>
<td>X</td>
<td>--</td>
</tr>
<tr>
<td>RDFSBASE</td>
<td>Read FS Base Address</td>
<td>X</td>
<td>“RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE Instructions” on page 164</td>
</tr>
<tr>
<td>RDGSBASE</td>
<td>Read GS Base Address</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>RDMSR</td>
<td>Read Model-Specific Register</td>
<td>X</td>
<td>“RDTSC and WRMSR Instructions” on page 163</td>
</tr>
<tr>
<td>RDPMC</td>
<td>Read Performance-Monitor Counter</td>
<td>X</td>
<td>“RDPMS Instruction” on page 163</td>
</tr>
<tr>
<td>RDTSC</td>
<td>Read Time-Stamp Counter</td>
<td>X</td>
<td>“RDTSC Instruction” on page 163</td>
</tr>
<tr>
<td>RDTSCP</td>
<td>Read Time-Stamp Counter and Processor ID</td>
<td>X</td>
<td>“RDTSCP Instruction” on page 163</td>
</tr>
<tr>
<td>RSM</td>
<td>Return from System-Management Mode</td>
<td>X</td>
<td>“Leaving SMM” on page 306</td>
</tr>
<tr>
<td>SGDT</td>
<td>Store Global-Descriptor-Table Register</td>
<td>X</td>
<td>“SGDT and SIDT Instructions” on page 164</td>
</tr>
<tr>
<td>SIDT</td>
<td>Store Interrupt-Descriptor-Table Register</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Note:

1. The operating system controls the privilege required to use the instruction.
Table 6-1. System Management Instructions (continued)

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Name</th>
<th>Privilege</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKINIT</td>
<td>Secure Init and Jump with Attestation</td>
<td>X</td>
<td>“Security” on page 508</td>
</tr>
<tr>
<td>SLDT</td>
<td>Store Local-Descriptor-Table Register</td>
<td>X</td>
<td>“SLDT and STR Instructions” on page 164</td>
</tr>
<tr>
<td>SMSW</td>
<td>Store Machine-Status Word</td>
<td>X</td>
<td>“LMSW and SMSW Instructions” on page 162</td>
</tr>
<tr>
<td>STI</td>
<td>Set Interrupt Flag</td>
<td>X</td>
<td>“CLI and STI Instructions” on page 162</td>
</tr>
<tr>
<td>STGI</td>
<td>Set Global Interrupt Flag</td>
<td>X</td>
<td>“Global Interrupt Flag, STGI and CLGI Instructions” on page 484</td>
</tr>
<tr>
<td>STR</td>
<td>Store Task Register</td>
<td>X</td>
<td>“SLDT and STR Instructions” on page 164</td>
</tr>
<tr>
<td>SWAPGS</td>
<td>Swap GS and KernelGSbase Registers</td>
<td>X</td>
<td>“SWAPGS Instruction” on page 161</td>
</tr>
<tr>
<td>SYSCALL</td>
<td>Fast System Call</td>
<td>X</td>
<td>“SYSCALL and SYSRET” on page 159</td>
</tr>
<tr>
<td>SYSENDER</td>
<td>System Call</td>
<td>X</td>
<td>“SYSENDER and SYSEXIT (Legacy Mode Only)” on page 159</td>
</tr>
<tr>
<td>SYSEXIT</td>
<td>System Return</td>
<td>X</td>
<td>“SYSCALL and SYSRET” on page 159</td>
</tr>
<tr>
<td>SYSRET</td>
<td>Fast System Return</td>
<td>X</td>
<td>“SYSCALL and SYSRET” on page 159</td>
</tr>
<tr>
<td>VERR</td>
<td>Verify Segment for Reads</td>
<td>X</td>
<td>“Checking Read/Write Rights” on page 165</td>
</tr>
<tr>
<td>VERW</td>
<td>Verify Segment for Writes</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>VMLOAD</td>
<td>Load State from VMCB</td>
<td>X</td>
<td>“VMSAVE and VMLOAD Instructions” on page 474</td>
</tr>
<tr>
<td>VMMCAL</td>
<td>Call VMM</td>
<td>X</td>
<td>“VMMCALL Instruction” on page 485</td>
</tr>
<tr>
<td>VMRUN</td>
<td>Run Virtual Machine</td>
<td>X</td>
<td>“VMRUN Instruction” on page 456</td>
</tr>
<tr>
<td>VMSAVE</td>
<td>Save State to VMCB</td>
<td>X</td>
<td>“VMSAVE and VMLOAD Instructions” on page 474</td>
</tr>
<tr>
<td>WBINVD</td>
<td>Writeback and Invalidate Caches</td>
<td>X</td>
<td>“Cache Management” on page 166</td>
</tr>
<tr>
<td>WBNIOINVD</td>
<td>Writeback No Invalidate</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>WRFSBASE</td>
<td>Write FS Base Address</td>
<td>X</td>
<td>“RDFSMBASE, RDGSBASE, WRFSBASE, and WRGSBASE Instructions” on page 164</td>
</tr>
<tr>
<td>WRGSBASE</td>
<td>Write GS Base Address</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>WRMSR</td>
<td>Write Model-Specific Register</td>
<td>X</td>
<td>“RDMRSR and WRMSR Instructions” on page 163</td>
</tr>
</tbody>
</table>

Note:
1. The operating system controls the privilege required to use the instruction.
User Mode Instruction Prevention (UMIP)

This security mode restricts certain instructions so that they do not reveal information about structures that are controlled by the processor when it is at CPL=0. The presence of the UMIP feature is indicated by CPUID Function 0000_0007, ECX[2]=1. This mode is enabled by setting CR4 bit 11 to a 1. Attempts to set CR4 bit 11 when the UMIP feature is not supported result in a #GP fault. Once CR4[11] is set to 1, the SGDT, SIDT, SLDT, SMSW and STR instructions become available only at CPL=0. Any attempt to execute them with CPL>0 results in a #GP fault with error code 0.

The following instructions are summarized in this chapter but are not categorized as system instructions, because of their importance to application programming:

- The CPUID instruction returns information critical to system software in initializing the operating environment. It is fully described in Section 3.3, “Processor Feature Identification,” on page 64.
- The PUSHF and POPF instructions set and clear certain rFLAGS bits depending on the processor operating mode and privilege level. These dependencies are described in “POPF and PUSHF Instructions” on page 162.
- The MOV, PUSH, and POP instructions can be used to load and store segment registers, as described in “MOV, POP, and PUSH Instructions” on page 163.

6.1 Fast System Call and Return

Operating systems can use both paging and segmentation to implement protected memory models. Segment descriptors provide the necessary memory protection and privilege checking for segment accesses. By setting segment-descriptor fields appropriately, operating systems can enforce access restrictions as needed.

A disadvantage of segment-based protection and privilege checking is the overhead associated with loading a new segment selector (and its corresponding descriptor) into a segment register. Even when using the flat-memory model, this overhead still occurs when switching between privilege levels because code segments (CS) and stack segments (SS) are reloaded with different segment descriptors.

To initiate a call to the operating system, an application transfers control to the operating system through a gate descriptor (call, interrupt, trap, or task gate). In the past, control was transferred using either a far CALL instruction or a software interrupt. Transferring control through one of these gates is slowed by the segmentation-related overhead, as is the later return using a far RET or IRET instruction. The following checks are performed when control is transferred in this manner:

- Selectors, gate descriptors, and segment descriptors are in the proper form.
- Descriptors lie within the bounds of the descriptor tables.
- Gate descriptors reference the appropriate segment descriptors.
• The caller, gate, and target privileges all allow the control transfer to take place.
• The stack created by the call has sufficient properties to allow the transfer to take place.

In addition to these call-gate checks, other checks are made involving the task-state segment when a task switch occurs.

6.1.1 SYSCALL and SYSRET

**SYSCALL and SYSRET Instructions.** SYSCALL and SYSRET are low-latency system call and return instructions. These instructions assume the operating system implements a flat-memory model, which greatly simplifies calls to and returns from the operating system. This simplification comes from eliminating unneeded checks, and by loading pre-determined values into the CS and SS segment registers (both visible and hidden portions). As a result, SYSCALL and SYSRET can take fewer than one-fourth the number of internal clock cycles to complete than the legacy CALL and RET instructions. SYSCALL and SYSRET are particularly well-suited for use in 64-bit mode, which requires implementation of a paged, flat-memory model.

SYSCALL and SYSRET require that the code-segment base, limit, and attributes (except for DPL) are consistent for all application and system processes. Only the DPL is allowed to vary. The processor assumes (but does not check) that the SYSCALL target CS segment descriptor entry has DPL=0 and the SYSRET target CS segment descriptor entry has DPL=3.

For details on the SYSCALL and SYSRET instructions, see “System Instruction Reference” in Volume 3.

**SYSCALL and SYSRET MSRs.** The STAR, LSTAR, and CSTAR registers are model-specific registers (MSRs) used to specify the target address of a SYSCALL instruction as well as the CS and SS selectors of the called and returned procedures. The SFMASK register is used in long mode to specify how rFLAGS is handled by these instructions. Figure 6-1 shows the STAR, LSTAR, CSTAR, and SFMASK register formats.

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STAR</td>
<td>C000_0081h SYSRET CS and SS, SYSCALL CS and SS, 32-bit SYSCALL Target EIP</td>
</tr>
<tr>
<td>LSTAR</td>
<td>C000_0082h Target RIP for 64-Bit-Mode Calling Software</td>
</tr>
<tr>
<td>CSTAR</td>
<td>C000_0083h Target RIP for Compatibility-Mode Calling Software</td>
</tr>
<tr>
<td>SFMASK</td>
<td>C000_0084h Reserved, RAZ, SYSCALL Flag Mask</td>
</tr>
</tbody>
</table>

*Figure 6-1. STAR, LSTAR, CSTAR, and MASK MSRs*
• **STAR**—The STAR register has the following fields (unless otherwise noted, all bits are read/write):
  - **SYSRET CS and SS Selectors**—Bits 63:48. This field is used to specify both the CS and SS selectors loaded into CS and SS during SYSRET. If SYSRET is returning to 32-bit mode (either legacy or compatibility), this field is copied directly into the CS selector field. If SYSRET is returning to 64-bit mode, the CS selector is set to this field + 16. SS.Sel is set to this field + 8, regardless of the target mode. Because SYSRET always returns to CPL 3, the RPL bits 49:48 should be initialized to 11b.
  - **SYSCALL CS and SS Selectors**—Bits 47:32. This field is used to specify both the CS and SS selectors loaded into CS and SS during SYSCALL. This field is copied directly into CS.Sel. SS.Sel is set to this field + 8. Because SYSCALL always switches to CPL 0, the RPL bits 33:32 should be initialized to 00b.
  - **32-bit SYSCALL Target EIP**—Bits 31:0. This is the target EIP of the called procedure.

The legacy STAR register is not expanded in long mode to provide a 64-bit target RIP address. Instead, long mode provides two new STAR registers—long STAR (LSTAR) and compatibility STAR (CSTAR)—that hold a 64-bit target RIP.

• **LSTAR and CSTAR**—The LSTAR register holds the target RIP of the called procedure in long mode when the calling software is in 64-bit mode. The CSTAR register holds the target RIP of the called procedure in long mode when the calling software is in compatibility mode. The WRMSR instruction is used to load the target RIP into the LSTAR and CSTAR registers. If the RIP written to either of the MSRs is not in canonical form, a #GP fault is generated on the WRMSR instruction.

• **SFMASK**—The SFMASK register is used to specify which RFLAGS bits are cleared during a SYSCALL. In long mode, SFMASK is used to specify which RFLAGS bits are cleared when SYSCALL is executed. If a bit in SFMASK is set to 1, the corresponding bit in RFLAGS is cleared to 0. If a bit in SFMASK is cleared to 0, the corresponding RFLAGS bit is not modified.

### 6.1.2 SYSENTER and SYSEXIT (Legacy Mode Only)

**SYSENTER and SYSEXIT Instructions.** Like SYSCALL and SYSRET, SYSENTER and SYSEXIT are low-latency system call and return instructions designed for use by system and application software implementing a flat-memory model. However, these instructions are illegal in long mode and result in an undefined opcode exception (#UD) if software attempts to use them. Software should use the SYSCALL and SYSRET instructions when running in long mode.

**SYSENTER and SYSEXIT MSRs.** Three model-specific registers (MSRs) are used to specify the target address and stack pointers for the SYSENTER instruction as well as the CS and SS selectors of the called and returned procedures. The register fields are:

• **SYSENTER Target CS**—Holds the CS selector of the called procedure.

• **SYSENTER Target ESP**—Holds the called-procedure stack pointer. The SS selector is updated automatically to point to the next descriptor entry after the SYSENTER Target CS, and ESP is the offset into that stack segment.