AMD64 Technology

AMD64 Architecture Programmer's Manual Volume 2: System Programming

Publication No.	Revision	Date
24593	**3.08**	**April 2003**

© 2002, 2003 Advanced Micro Devices, Inc. All rights reserved.

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

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

Trademarks

AMD, the AMD arrow logo, AMD Athlon, AMD Duron, and combinations thereof, and 3DNow! are trademarks, and Am486, $Am5_x86$, and AMD-K6 are registered trademarks of Advanced Micro Devices, Inc.

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

Windows NT is a registered trademark of Microsoft Corporation.

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

Contents

Figures xiii

Tables xix

Revision History xxi

Preface xxiii

About This Book xxiii
Audience xxiii
Contact Information xxiii
Organization xxiv
Definitions xxv
Related Documents xxxvi

1 System-Programming Overview **1**

1.1 Memory Model 1
Memory Addressing 2
Memory Organization 4
Canonical Address Form 5
1.2 Memory Management 6
Segmentation 6
Paging 8
Mixing Segmentation and Paging 10
Real Addressing 11
1.3 Operating Modes 12
Long Mode 14
64-Bit Mode 15
Compatibility Mode 15
Legacy Modes 16
System Management Mode (SMM) 17
1.4 System Registers 17
1.5 System-Data Structures 20
1.6 Interrupts 22
1.7 Additional System-Programming Facilities 24
Hardware Multitasking 24
Machine Check 25
Software Debugging 26
Performance Monitoring 26

2 x86 and AMD64 Architecture Differences **29**

2.1 Operating Modes 29
Long Mode 29
Legacy Mode 30
System-Management Mode 30

2.2 Memory Model . . . 31
Memory Addressing . . . 31
Page Translation . . . 31
Segmentation . . . 33
2.3 Protection Checks . . . 35
2.4 Registers . . . 35
General-Purpose Registers . . . 35
128-Bit Media Registers . . . 36
Flags Register . . . 36
Instruction Pointer . . . 36
Stack Pointer . . . 36
Control Registers . . . 36
Debug Registers . . . 36
Extended Feature Register (EFER) . . . 37
Memory Type Range Registers (MTRRs) . . . 37
Other Model-Specific Registers (MSRs) . . . 37
2.5 Instruction Set . . . 37
REX Prefixes . . . 37
Segment-Override Prefixes in 64-Bit Mode . . . 38
Operands and Results . . . 38
Address Calculations . . . 38
Instructions that Reference RSP . . . 39
Branches . . . 40
NOP Instruction . . . 43
Single-Byte INC and DEC Instructions . . . 43
MOVSXD Instruction . . . 43
Invalid Instructions . . . 44
FXSAVE and FXRSTOR Instructions . . . 45
2.6 Interrupts and Exceptions . . . 46
Interrupt Descriptor Table . . . 46
Stack Frame Pushes . . . 46
Stack Switching . . . 46
IRET Instruction . . . 47
Task-Priority Register (CR8) . . . 48
New Exception Conditions . . . 48
2.7 Hardware Task Switching . . . 48
2.8 Long-Mode vs. Legacy-Mode Differences . . . 49

3 System Resources . . . 51

3.1 System-Control Registers . . . 51
CR0 Register . . . 53
CR2 and CR3 Registers . . . 56
CR4 Register . . . 58
CR1 and CR5–CR7 Registers . . . 62
64-Bit-Mode Extended Control Registers . . . 62
CR8 (Task Priority Register, TPR) . . . 62
RFLAGS Register . . . 62
Extended Feature Enable Register (EFER) . . . 67

3.2 Model-Specific Registers (MSRs) 69
System Configuration Register (SYSCFG) 70
System-Linkage Registers 72
Memory-Typing Registers 72
Debug-Extension Registers 73
Performance-Monitoring Registers 73
Machine-Check Registers 74
3.3 Processor Feature Identification 75

4 Segmented Virtual Memory 77

4.1 Real Mode Segmentation 78
4.2 Virtual-8086 Mode Segmentation 78
4.3 Protected Mode Segmented-Memory Models 79
Multi-Segmented Model 79
Flat-Memory Model 80
Segmentation in 64-bit mode 80
4.4 Segmentation Data Structures and Registers 80
4.5 Segment Selectors and Registers 82
Segment Selectors 82
Segment Registers 84
Segment Registers in 64-bit Mode 85
4.6 Descriptor Tables 87
Global Descriptor Table 88
Global Descriptor-Table Register 89
Local Descriptor Table 90
Local Descriptor-Table Register 91
Interrupt Descriptor Table 94
Interrupt Descriptor-Table Register 95
4.7 Legacy Segment Descriptors 95
Descriptor Format 95
Code-Segment Descriptors 98
Data-Segment Descriptors 100
System Descriptors 102
Gate Descriptors 104
4.8 Long-Mode Segment Descriptors 106
Code-Segment Descriptors 106
Data-Segment Descriptors 108
System Descriptors 109
Gate Descriptors 111
Long Mode Descriptor Summary 114
4.9 Segment-Protection Overview 116
Privilege-Level Concept 117
Privilege-Level Types 118
4.10 Data-Access Privilege Checks 119
Accessing Data Segments 119
Accessing Stack Segments 121
4.11 Control-Transfer Privilege Checks 122
Direct Control Transfers 123

Control Transfers Through Call Gates 127
Return Control Transfers 135
4.12 Limit Checks 138
Determining Limit Violations 138
4.13 Type Checks 139
Type Checks in Legacy and Compatibility Modes 140
Long Mode Type Check Differences 141

5 Page Translation and Protection 143

5.1 Page Translation Overview 144
Page-Translation Options 146
Page-Translation Enable (PG) Bit 147
Physical-Address Extensions (PAE) Bit 147
Page-Size Extensions (PSE) Bit 147
Page-Directory Page Size (PS) Bit 148
5.2 Legacy-Mode Page Translation 148
CR3 Register 149
Normal (Non-PAE) Paging 150
PAE Paging 154
5.3 Long-Mode Page Translation 158
Canonical Address Form 159
CR3 159
4-Kbyte Page Translation 160
2-Mbyte Page Translation 163
5.4 Page-Translation-Table Entry Fields 166
Field Definitions 166
5.5 Translation-Lookaside Buffer (TLB) 170
Global Pages 171
TLB Management 171
5.6 Page-Protection Checks 172
No Execute (NX) Bit 173
User/Supervisor (U/S) Bit 173
Read/Write (R/W) Bit 174
Write Protect (CR0.WP) Bit 174
5.7 Protection Across Paging Hierarchy 174
Access to User Pages when CR0.WP=1 176
5.8 Effects of Segment Protection 176

6 System-Management Instructions 177

6.1 Fast System Call and Return 179
SYSCALL and SYSRET 180
SYSENTER and SYSEXIT (Legacy Mode Only) 182
SWAPGS Instruction 183
6.2 System Status and Control 184
Processor Feature Identification (CPUID) 184
Accessing Control Registers 184
Accessing the RFLAGs Register 185
Accessing Debug Registers 185

Accessing Model-Specific Registers 186
6.3 Segment Register and Descriptor Register Access 186
Accessing Segment Registers. 186
Accessing Descriptor-Table Registers 187
6.4 Protection Checking 187
Checking Access Rights 187
Checking Segment Limits. 188
Checking Read/Write Rights 188
Adjusting Access Rights. 188
6.5 Processor Halt 188
6.6 Cache and TLB Management 189
Cache Management. 189
TLB Invalidation 189

7 Memory System 191

7.1 Memory-Access Ordering 194
Read Ordering 195
Write Ordering. 195
Read/Write Barriers 196
7.2 Memory Coherency and Protocol 197
Special Coherency Considerations 199
7.3 Memory Types 200
7.4 Buffering and Combining Memory Writes 204
Write Buffering 204
Write Combining 205
7.5 Memory Caches 206
Cache Organization and Operation 207
Cache Control Mechanisms 210
Cache and Memory Management Instructions. 213
Serializing Instructions. 214
7.6 Memory-Type Range Registers 215
MTRR Type Fields. 216
MTRRs 217
Using MTRRs. 224
MTRRs and Page Cache Controls 225
MTRRs in Multi-Processing Environments. 227
7.7 Page-Attribute Table Mechanism 228
PAT Register 228
PAT Indexing 229
Identifying PAT Support. 230
PAT Accesses 230
Combined Effect of MTRRs and PAT 231
7.8 Memory-Mapped I/O 232
Extended Fixed-Range MTRR Type-Field Encodings 233
IORRs 235
IORR Overlapping. 237
Top of Memory 237

8 Exceptions and Interrupts. 241

8.1 Overview 241
8.2 General Characteristics 242
Precision 242
Instruction Restart 242
Types of Exceptions 243
Masking External Interrupts 243
Masking Floating-Point and Media Instructions 244
Disabling Exceptions 244
8.3 Vectors 245
#DE—Divide-By-Zero-Error Exception (Vector 0) 248
#DB—Debug Exception (Vector 1) 248
NMI—Non-Maskable-Interrupt Exception (Vector 2) 250
#BP—Breakpoint Exception (Vector 3) 250
#OF—Overflow Exception (Vector 4) 250
#BR—Bound-Range Exception (Vector 5) 251
#UD—Invalid-Opcode Exception (Vector 6) 251
#NM—Device-Not-Available Exception (Vector 7) 252
#DF—Double-Fault Exception (Vector 8) 252
Coprocessor-Segment-Overrun Exception (Vector 9) 254
#TS—Invalid-TSS Exception (Vector 10) 254
#NP—Segment-Not-Present Exception (Vector 11) 255
#SS—Stack Exception (Vector 12) 256
#GP—General-Protection Exception (Vector 13) 257
#PF—Page-Fault Exception (Vector 14) 259
#MF—x87 Floating-Point Exception-Pending (Vector 16) . . 260
#AC—Alignment-Check Exception (Vector 17) 261
#MC—Machine-Check Exception (Vector 18) 262
#XF—SIMD Floating-Point Exception (Vector 19) 263
User-Defined Interrupts (Vectors 32–255) 264
8.4 Exceptions During a Task Switch 265
8.5 Error Codes 265
Selector-Error Code 265
Page-Fault Error Code 266
8.6 Priorities 267
Floating-Point Exception Priorities 269
External Interrupt Priorities 270
8.7 Real-Mode Interrupt Control Transfers 272
8.8 Legacy Protected-Mode Interrupt Control Transfers 274
Locating the Interrupt Handler 274
Interrupt To Same Privilege 275
Interrupt To Higher Privilege 276
Privilege Checks 278
Returning From Interrupt Procedures 281
8.9 Virtual-8086 Mode Interrupt Control Transfers 282
Protected-Mode Handler Control Transfer 283
Virtual-8086 Handler Control Transfer 284

8.10 Long-Mode Interrupt Control Transfers 285
Interrupt Gates and Trap Gates 285
Locating the Interrupt Handler 286
Interrupt Stack Frame 287
Interrupt-Stack Table 290
Returning From Interrupt Procedures 292
8.11 Virtual Interrupts 293
Virtual-8086 Mode Extensions 294
Protected Mode Virtual Interrupts 298
Effect of Instructions that Modify EFLAGS.IF 298

9 Machine Check Mechanism 303

9.1 Determining Machine-Check Support 303
9.2 Machine-Check Errors 304
Error Sources 304
9.3 Machine Check MSRs 305
Global Status and Control Registers 306
Error-Reporting Register Banks 308
Error Codes 310
9.4 Initializing the Machine-Check Mechanism 313
9.5 Using Machine Check Features 314
Handling Machine Check Exceptions 314
Reporting Correctable Machine Check Errors 316

10 System-Management Mode 319

SMM Differences 320
10.1 SMM Resources 320
SMRAM 320
SMBASE Register 321
SMRAM State-Save Area 322
SMM-Revision Identifier 328
10.2 Using SMM 329
System-Management Interrupt (SMI) 329
SMM Operating-Environment 329
Exceptions and Interrupts 330
Invalidating the Caches 332
Saving Additional Processor State 332
Operating in Protected Mode and Long Mode 333
Auto-Halt Restart 333
I/O Instruction Restart 334
10.3 Leaving SMM 335

11 128-Bit, 64-Bit, and x87 Programming 337

11.1 Overview of System-Software Considerations 337
11.2 Determining Media and x87 Feature Support 337
11.3 Enabling 128-Bit Media Instructions 338
11.4 Media and x87 Processor State 339
128-Bit Media State 339

64-Bit Media State . . . 341
x87 State . . . 342
Saving Media and x87 Processor State . . . 344

12 Task Management . . . 357

12.1 Hardware Multitasking Overview . . . 357
12.2 Task-Management Resources . . . 358
TSS Selector . . . 360
TSS Descriptor . . . 360
Task Register . . . 361
Legacy Task-State Segment . . . 363
64-Bit Task State Segment . . . 368
Task Gate Descriptor (Legacy Mode Only) . . . 371
12.3 Hardware Task-Management in Legacy Mode . . . 371
Task Memory-Mapping . . . 371
Switching Tasks . . . 373
Task Switches Using Task Gates . . . 375
Nesting Tasks . . . 377

13 Debug and Performance Resources . . . 381

13.1 Software-Debug Resources . . . 382
Debug Registers . . . 383
13.2 Breakpoints . . . 392
Setting Breakpoints . . . 392
Using Breakpoints . . . 394
Breakpoint Instruction (INT3) . . . 399
Control-Transfer Breakpoint Features . . . 399
13.3 Performance Optimization . . . 401
Performance Counters . . . 402
Performance Event-Select Registers . . . 403
Using Performance Counters . . . 406
Time-Stamp Counter . . . 406

14 Processor Initialization and Long-Mode Activation . . . 409

14.1 Reset and Initialization . . . 409
Built-In Self Test (BIST) . . . 409
Clock Multiplier Selection . . . 410
Processor Initialization State . . . 410
Multiple Processor Initialization . . . 413
Fetching the First Instruction . . . 413
14.2 Hardware Configuration . . . 414
Processor Implementation Information . . . 414
Enabling Internal Caches . . . 414
Initializing Media and x87 Processor State . . . 415
Model-Specific Initialization . . . 417
14.3 Initializing Real Mode . . . 418
14.4 Initializing Protected Mode . . . 419
14.5 Initializing Long Mode . . . 420

14.6 Enabling and Activating Long Mode 422
Activating Long Mode 423
Consistency Checks 424
Updating System Descriptor Table References 424
Relocating Page-Translation Tables 425
14.7 Leaving Long Mode 425
14.8 Long-Mode Initialization Example 426

Appendix A MSR Cross-Reference 433
A.1 MSR Cross-Reference by MSR Address 433
A.2 System-Software MSRs 437
A.3 Memory-Typing MSRs 438
A.4 Machine-Check MSRs 441
A.5 Software-Debug MSRs 442
A.6 Performance-Monitoring MSRs 443

Index 445

Figures

Figure 1-1. Segmented-Memory Model 7
Figure 1-2. Flat Memory Model 8
Figure 1-3. Paged Memory Model 9
Figure 1-4. 64-Bit Flat, Paged-Memory Model 11
Figure 1-5. Real-Address Memory Model 12
Figure 1-6. Operating Modes of the AMD64 Architecture 14
Figure 1-7. System Registers 19
Figure 1-8. System-Data Structures 21
Figure 3-1. Control Register 0 (CR0) 53
Figure 3-2. Control Register 2 (CR2)—Legacy-Mode 57
Figure 3-3. Control Register 2 (CR2)—Long Mode 57
Figure 3-4. Control Register 3 (CR3)—Legacy-Mode Non-PAE Paging . . . 57
Figure 3-5. Control Register 3 (CR3)—Legacy-Mode PAE Paging 57
Figure 3-6. Control Register 3 (CR3)—Long Mode 58
Figure 3-7. Control Register 4 (CR4) 58
Figure 3-8. RFLAGS Register 63
Figure 3-9. Extended Feature Enable Register (EFER) 68
Figure 3-10. AMD64 Architecture Model-Specific Registers 70
Figure 3-11. System-Configuration Register (SYSCFG) 71
Figure 4-1. Segmentation Data Structures 81
Figure 4-2. Segment and Descriptor-Table Registers 82
Figure 4-3. Segment Selector 83
Figure 4-4. Segment-Register Format 85
Figure 4-5. FS and GS Segment-Register Format—64-Bit Mode 86
Figure 4-6. Global and Local Descriptor-Table Access 89
Figure 4-7. GDTR and IDTR Format—Legacy Modes 89
Figure 4-8. GDTR and IDTR Format—Long Mode 90
Figure 4-9. Relationship between the LDT and GDT 91
Figure 4-10. LDTR Format—Legacy Mode 92
Figure 4-11. LDTR Format—Long Mode 92
Figure 4-12. Indexing an IDT 95

Figure 4-13. Generic Segment Descriptor—Legacy Mode 96
Figure 4-14. Code-Segment Descriptor—Legacy Mode 99
Figure 4-15. Data-Segment Descriptor—Legacy Mode 101
Figure 4-16. LDT and TSS Descriptor—Legacy/Compatibility Modes 104
Figure 4-17. Call-Gate Descriptor—Legacy Mode 105
Figure 4-18. Interrupt-Gate and Trap-Gate Descriptors—Legacy Mode . . 105
Figure 4-19. Task-Gate Descriptor—Legacy Mode 105
Figure 4-20. Code-Segment Descriptor—Long Mode 107
Figure 4-21. Data-Segment Descriptor—Long Mode 108
Figure 4-22. System-Segment Descriptor—64-Bit Mode 111
Figure 4-23. Call-Gate Descriptor—Long Mode 112
Figure 4-24. Interrupt-Gate and Trap-Gate Descriptors—Long Mode 113
Figure 4-25. Privilege-Level Relationships 118
Figure 4-26. Data-Access Privilege-Check Examples 120
Figure 4-27. Stack-Access Privilege-Check Examples 122
Figure 4-28. Nonconforming Code-Segment Privilege-Check Examples . . 125
Figure 4-29. Conforming Code-Segment Privilege-Check Examples 127
Figure 4-30. Legacy-Mode Call-Gate Transfer Mechanism 128
Figure 4-31. Long-Mode Call-Gate Access Mechanism 129
Figure 4-32. Privilege-Check Examples for Call Gates 131
Figure 4-33. Legacy-Mode 32-Bit Stack Switch, with Parameters 133
Figure 4-34. 32-Bit Stack Switch, No Parameters—Legacy Mode 134
Figure 4-35. Stack Switch—Long Mode 135
Figure 5-1. Virtual to Physical Address Translation—Long Mode 145
Figure 5-2. Control Register 3 (CR3)—Non-PAE Paging Legacy-Mode . . 149
Figure 5-3. Control Register 3 (CR3)—PAE Paging Legacy-Mode 149
Figure 5-4. 4-Kbyte Non-PAE Page Translation—Legacy Mode 151
Figure 5-5. 4-Kbyte PDE—Non-PAE Paging Legacy-Mode 151
Figure 5-6. 4-Kbyte PTE—Non-PAE Paging Legacy-Mode 152
Figure 5-7. 4-Mbyte Page Translation—Non-PAE Paging Legacy-Mode . 153
Figure 5-8. 4-Mbyte PDE—Non-PAE Paging Legacy-Mode 153
Figure 5-9. 4-Kbyte PAE Page Translation—Legacy Mode 155
Figure 5-10. 4-Kbyte PDPE—PAE Paging Legacy-Mode 156

Figure 5-11. 4-Kbyte PDE—PAE Paging Legacy-Mode 156
Figure 5-12. 4-Kbyte PTE—PAE Paging Legacy-Mode 156
Figure 5-13. 2-Mbyte PAE Page Translation—Legacy Mode 157
Figure 5-14. 2-Mbyte PDPE—PAE Paging Legacy-Mode. 158
Figure 5-15. 2-Mbyte PDE—PAE Paging Legacy-Mode 158
Figure 5-16. Control Register 3 (CR3)—Long Mode 159
Figure 5-17. 4-Kbyte Page Translation—Long Mode 161
Figure 5-18. 4-Kbyte PML4E—Long Mode 162
Figure 5-19. 4-Kbyte PDPE—Long Mode 162
Figure 5-20. 4-Kbyte PDE—Long Mode. 162
Figure 5-21. 4-Kbyte PTE—Long Mode 163
Figure 5-22. 2-Mbyte Page Translation—Long Mode. 164
Figure 5-23. 2-Mbyte PML4E—Long Mode 165
Figure 5-24. 2-Mbyte PDPE—Long Mode 165
Figure 5-25. 2-Mbyte PDE—Long Mode 165
Figure 6-1. STAR, LSTAR, CSTAR, and MASK MSRs. 181
Figure 6-2. SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP MSRs . 183
Figure 7-1. Processor and Memory System 192
Figure 7-2. MOESI State Transitions 198
Figure 7-3. Cache Organization Example 208
Figure 7-4. MTRR Mapping of Physical Memory 218
Figure 7-5. Fixed-Range MTRR 219
Figure 7-6. MTRRphysBase*n* Register. 221
Figure 7-7. MTRRphysMask*n* Register 222
Figure 7-8. MTRR defType Register Format 223
Figure 7-9. MTRR Capability Register Format 225
Figure 7-10. PAT Register. 228
Figure 7-11. Extended MTRR Type-Field Format (Fixed-Range MTRRs) 233
Figure 7-12. IORRBase*n* Register 236
Figure 7-13. IORRMask*n* Register 237
Figure 7-14. Memory Organization Using Top-of-Memory Registers 238
Figure 7-15. Top-of-Memory Registers (TOP_MEM, TOP_MEM2) 239
Figure 8-1. Control Register 2 (CR2) 259

Figure 8-2. Selector Error Code 265
Figure 8-3. Page-Fault Error Code 266
Figure 8-4. Task Priority Register (CR8) 271
Figure 8-5. Real-Mode Interrupt Control Transfer 272
Figure 8-6. Stack After Interrupt in Real Mode 273
Figure 8-7. Protected-Mode Interrupt Control Transfer 275
Figure 8-8. Stack After Interrupt to Same Privilege Level 276
Figure 8-9. Stack After Interrupt to Higher Privilege 278
Figure 8-10. Privilege-Check Examples for Interrupts 280
Figure 8-11. Stack After Virtual-8086 Mode Interrupt to Protected Mode 284
Figure 8-12. Long-Mode Interrupt Control Transfer 287
Figure 8-13. Long-Mode Stack After Interrupt—Same Privilege 289
Figure 8-14. Long-Mode Stack After Interrupt—Higher Privilege. 290
Figure 8-15. Long-Mode IST Mechanism 292
Figure 9-1. MCG_CAP Register 306
Figure 9-2. MCG_STATUS Register. 307
Figure 9-3. MCG_CTL Register 308
Figure 9-4. MCi_CTL Registers 309
Figure 9-5. MC*i*_STATUS Register 310
Figure 10-1. Default SMRAM Memory Map 321
Figure 10-2. SMBASE Register. 322
Figure 10-3. SMM-Revision Identifier 328
Figure 11-1. 128-Bit Media-Instruction State 340
Figure 11-2. 64-Bit Media-Instruction State 341
Figure 11-3. x87-Instruction State 343
Figure 11-4. FSAVE/FNSAVE Image (32-Bit, Protected Mode) 346
Figure 11-5. FSAVE/FNSAVE Image (32-Bit, Real/Virtual-8086 Modes) . 347
Figure 11-6. FSAVE/FNSAVE Image (16-Bit, Protected Mode) 348
Figure 11-7. FSAVE/FNSAVE Image (16-Bit, Real/Virtual-8086 Modes) . 349
Figure 11-8. FXSAVE and FXRSTOR Image (Long Mode) 351
Figure 11-9. FXSAVE and FXRSTOR Image (Legacy Mode) 352
Figure 12-1. Task-Management Resources 359
Figure 12-2. Task-Segment Selector 360

Figure 12-3. TR Format, Legacy Mode 362
Figure 12-4. TR Format, Long Mode 362
Figure 12-5. Relationship between the TSS and GDT 363
Figure 12-6. Legacy 32-bit TSS 364
Figure 12-7. I/O-Permission Bitmap Example 368
Figure 12-8. Long Mode TSS Format 370
Figure 12-9. Task-Gate Descriptor, Legacy Mode Only 371
Figure 12-10. Privilege-Check Examples for Task Gates 377
Figure 13-1. Address-Breakpoint Registers (DR0–DR3) 384
Figure 13-2. Debug-Status Register (DR6) 385
Figure 13-3. Debug-Control Register (DR7) 386
Figure 13-4. Debug-Control MSR (DebugCtlMSR) 389
Figure 13-5. Control-Transfer Recording MSRs 391
Figure 13-6. Performance Counter (PerfCtr*n*) 402
Figure 13-7. Performance Event-Select Register (PerfEvtSel*n*) 404
Figure 13-8. Time-Stamp Counter (TSC) 406

Tables

Table 1-1. Operating Modes 13
Table 1-2. Interrupts and Exceptions 24
Table 2-1. Instructions That Reference RSP 40
Table 2-2. 64-Bit Mode Near Branches, Default 64-Bit Operand Size 41
Table 2-3. Invalid Instructions in 64-Bit Mode 44
Table 2-4. Invalid Instructions in Long Mode 45
Table 2-5. Reassigned Instructions in 64-Bit Mode 45
Table 2-6. Differences Between Long Mode and Legacy Mode 50
Table 4-1. Segment Registers 84
Table 4-2. Descriptor Types 97
Table 4-3. Code-Segment Descriptor Types 100
Table 4-4. Data-Segment Descriptor Types 102
Table 4-5. System-Segment Descriptor Types (S=0)—Legacy Mode . . . 103
Table 4-6. System-Segment Descriptor Types—Long Mode 110
Table 4-7. Descriptor-Entry Field Changes in Long Mode 115
Table 5-1. Supported Paging Alternatives (CR0.PG=1) 146
Table 5-2. Physical-Page Protection, CR0.WP=0 175
Table 5-3. Effect of CR0.WP=1 on Supervisor Page Access 176
Table 6-1. System-Management Instructions 177
Table 7-1. Memory Access by Memory Type 203
Table 7-2. Caching Policy by Memory Type 203
Table 7-3. AMD64 Architecture Cache-Operating Modes 211
Table 7-4. MTRR Type Field Encodings 217
Table 7-5. Fixed-Range MTRR Address Ranges 220
Table 7-6. Combined MTRR and Page-Level Memory Type with Unmodified PAT MSR 226
Table 7-7. PAT Type Encodings 229
Table 7-8. PAT-Register PA-Field Indexing 230
Table 7-9. Combined Effect of MTRR and PAT Memory Types 232
Table 7-10. Extended Fixed-Range MTRR Type Encodings 235
Table 8-1. Interrupt-Vector Source and Cause 246
Table 8-2. Interrupt-Vector Classification 247
Table 8-3. Double-Fault Exception Conditions 253
Table 8-4. Invalid-TSS Exception Conditions 255

Table 8-5. Stack Exception Error Codes 257
Table 8-6. General-Protection Exception Conditions 258
Table 8-7. Data-Type Alignment 262
Table 8-8. Simultaneous Interrupt Priorities 268
Table 8-9. Simultaneous Floating-Point Exception Priorities 270
Table 8-10. Virtual-8086 Mode Interrupt Mechanisms 283
Table 8-11. Effect of Instructions that Modify the IF Bit 299
Table 10-1. AMD64 Architecture SMM State-Save Area 323
Table 10-2. Legacy SMM State-Save Area (Not used by AMD64 Architecture) 326
Table 10-3. SMM Register Initialization 330
Table 11-1. Deriving FSAVE Tag Field from FXSAVE Tag Field 355
Table 12-1. Effects of Task Nesting 378
Table 13-1. Breakpoint-Setting Examples 393
Table 13-2. Breakpoint Location by Condition 395
Table 13-3. Operating-System Mode and User Mode Bits 404
Table 14-1. Initial Processor State 410
Table 14-2. Initial State of Segment-Register Attributes 413
Table 14-3. x87 Floating-Point State Initialization 416
Table 14-4. Processor Operating Modes 422
Table 14-5. Long-Mode Consistency Checks 424
Table A-1. MSRs of the AMD64 Architecture 433
Table A-2. System-Software MSR Cross-Reference 437
Table A-3. Memory-Typing MSR Cross-Reference 439
Table A-4. Machine-Check MSR Cross-Reference 441
Table A-5. Software-Debug MSR Cross-Reference 442
Table A-6. Performance-Monitoring MSR Cross-Reference 443

Revision History

Date	Revision	Description
April 2003	3.08	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 Section 7.2.
September 2002	3.07	Made numerous small grammatical changes and factual clarifications. Added Revision History.

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.

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

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.

Contact Information

To submit questions or comments concerning this document, contact our technical documentation staff at AMD64.Feedback@amd.com.

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-Management 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.
- *128-Bit, 64-Bit, and x87 Programming*—The resources and functions that support use (by application software) and state-saving (by the operation system) of the 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.

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.

Definitions

Some of the following definitions assume a knowledge of the legacy x86 architecture. See "Related Documents" on page xxxvi for descriptions of the legacy x86 architecture.

Terms and Notation

1011b

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

F0EAh

A hexadecimal value—in this example a 2-byte value.

[1,2)

A range that includes the left-most value (in this case, 1) but excludes the right-most value (in this case, 2).

7–4

A bit range, from bit 7 to 4, inclusive. The high-order bit is shown first.

128-bit media instructions

Instructions that use the 128-bit XMM registers. These are a combination of the SSE and SSE2 instruction sets.

64-bit media instructions

Instructions that use the 64-bit MMX™ registers. These are primarily a combination of MMX and 3DNow!™ instruction

sets, with some additional instructions from the SSE and SSE2 instruction sets.

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.

#GP(0)

Notation indicating a general-protection exception (#GP) with error code of 0.

absolute

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

biased exponent

The sum of a floating-point value's exponent and a constant bias for a particular floating-point data type. The bias makes the range of the biased exponent always positive, which allows reciprocation without overflow.

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.

CR0–CR4

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

CR0.PE = 1

Notation indicating that the PE bit of the CR0 register has a value of 1.

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*.

DS:rSI

The contents of a memory location whose segment address is in the DS register and whose offset relative to that segment is in the rSI register.

EFER.LME = 0

Notation indicating that the LME bit of the EFER register has a value of 0.

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.

element

See *vector*.

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.

FF /0

Notation indicating that FF is the first byte of an opcode, and a subopcode in the ModR/M byte has a value of 0.

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.

IDT

Interrupt descriptor table.

IGN

Ignore. Field is ignored.

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 xxxvi 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, a general-protection exception (#GP) occurs.

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 direct memory offset. In other words, a displacement that is added to the base of a code segment (for absolute addressing) or to an instruction pointer (for addressing relative to the instruction pointer, as in RIP-relative addressing).

msb

Most-significant bit.

MSB

Most-significant byte.

multimedia instructions

A combination of *128-bit media instructions* and *64-bit media instructions*.

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.

packed

See *vector*.

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

Read as zero (0), regardless of what is written.

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*.

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. Compare *moffset*.

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).

SIMD

Single instruction, multiple data. See *vector*.

SSE

Streaming SIMD extensions instruction set. See *128-bit media instructions* and *64-bit media instructions*.

SSE2

Extensions to the SSE instruction set. See *128-bit media instructions* and *64-bit media instructions*.

sticky bit

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

TOP

The x87 top-of-stack pointer.

TPR

Task-priority register (CR8).

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 operand. Most of the 128-bit

and 64-bit media instructions use vectors as operands. Vectors are also called *packed* or *SIMD* (single-instruction multiple-data) 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*.

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*.

EBP

Extended base pointer register.

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.

RAZ

Read as zero (0), regardless of what is written.

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, a new register introduced in the AMD64 architecture to speed interrupt management.

TR

Task register.

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

- Peter Abel, *IBM PC Assembly Language and Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
- Rakesh Agarwal, *80x86 Architecture & Programming: Volume II*, Prentice-Hall, Englewood Cliffs, NJ, 1991.
- AMD data sheets and application notes for particular hardware implementations of the AMD64 architecture.
- AMD, *AMD-K6™ MMX™ Enhanced Processor Multimedia Technology*, Sunnyvale, CA, 2000.
- AMD, *3DNow!™ Technology Manual*, Sunnyvale, CA, 2000.
- AMD, *AMD Extensions to the 3DNow!™ and MMX™ Instruction Sets*, Sunnyvale, CA, 2000.
- AMD, *SYSCALL and SYSRET Instruction Specification Application Note*, Sunnyvale, CA, 1998.
- Don Anderson and Tom Shanley, *Pentium Processor System Architecture*, Addison-Wesley, New York, 1995.
- Nabajyoti Barkakati and Randall Hyde, *Microsoft Macro Assembler Bible*, Sams, Carmel, Indiana, 1992.

- Barry B. Brey, *8086/8088, 80286, 80386, and 80486 Assembly Language Programming*, Macmillan Publishing Co., New York, 1994.
- Barry B. Brey, *Programming the 80286, 80386, 80486, and Pentium Based Personal Computer*, Prentice-Hall, Englewood Cliffs, NJ, 1995.
- Ralf Brown and Jim Kyle, *PC Interrupts,* Addison-Wesley, New York, 1994.
- Penn Brumm and Don Brumm, *80386/80486 Assembly Language Programming*, Windcrest McGraw-Hill, 1993.
- Geoff Chappell, *DOS Internals,* Addison-Wesley, New York, 1994.
- Chips and Technologies, Inc. *Super386 DX Programmer's Reference Manual*, Chips and Technologies, Inc., San Jose, 1992.
- John Crawford and Patrick Gelsinger, *Programming the 80386*, Sybex, San Francisco, 1987.
- Cyrix Corporation, *5x86 Processor BIOS Writer's Guide*, Cyrix Corporation, Richardson, TX, 1995.
- 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.
- Jeffrey P. Doyer, *Introduction to Protected Mode Programming*, course materials for an onsite class, 1992.
- Ray Duncan, *Extending DOS: A Programmer's Guide to Protected-Mode DOS*, Addison Wesley, NY, 1991.
- William B. Giles, *Assembly Language Programming for the Intel 80xxx Family*, Macmillan, New York, 1991.
- Frank van Gilluwe, *The Undocumented PC,* Addison-Wesley, New York, 1994.
- John L. Hennessy and David A. Patterson, *Computer Architecture*, Morgan Kaufmann Publishers, San Mateo, CA, 1996.
- Thom Hogan, *The Programmer's PC Sourcebook*, Microsoft Press, Redmond, WA, 1991.

- Hal Katircioglu, *Inside the 486, Pentium, and Pentium Pro*, Peer-to-Peer Communications, Menlo Park, CA, 1997.
- IBM Corporation, *486SLC™ Microprocessor Data Sheet*, IBM Corporation, Essex Junction, VT, 1993.
- IBM Corporation, *486SLC2™ Microprocessor Data Sheet*, IBM Corporation, Essex Junction, VT, 1993.
- IBM Corporation, *80486DX2 Processor Floating Point Instructions*, IBM Corporation, Essex Junction, VT, 1995.
- IBM Corporation, *80486DX2 Processor BIOS Writer's Guide*, IBM Corporation, Essex Junction, VT, 1995.
- IBM Corporation, *Blue Lightening 486DX2 Data Book*, IBM Corporation, Essex Junction, VT, 1994.
- Institute of Electrical and Electronics Engineers, *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985.
- Institute of Electrical and Electronics Engineers, *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, ANSI/IEEE Std 854-1987.
- Muhammad Ali Mazidi and Janice Gillispie Mazidi, *80X86 IBM PC and Compatible Computers*, Prentice-Hall, Englewood Cliffs, NJ, 1997.
- Hans-Peter Messmer, *The Indispensable Pentium Book,* Addison-Wesley, New York, 1995.
- Karen Miller, *An Assembly Language Introduction to Computer Architecture: Using the Intel Pentium*, Oxford University Press, New York, 1999.
- Stephen Morse, Eric Isaacson, and Douglas Albert, *The 80386/387 Architecture*, John Wiley & Sons, New York, 1987.
- NexGen Inc., *Nx586 Processor Data Book*, NexGen Inc., Milpitas, CA, 1993.
- NexGen Inc., *Nx686 Processor Data Book*, NexGen Inc., Milpitas, CA, 1994.
- Bipin Patwardhan, *Introduction to the Streaming SIMD Extensions in the Pentium III*, www.x86.org/articles/sse_pt1/simd1.htm, June, 2000.
- Peter Norton, Peter Aitken, and Richard Wilton, *PC Programmer's Bible,* Microsoft Press, Redmond, WA, 1993.
- *PharLap 386\ASM Reference Manual*, Pharlap, Cambridge MA, 1993.

- *PharLap TNT DOS-Extender Reference Manual*, Pharlap, Cambridge MA, 1995.
- Sen-Cuo Ro and Sheau-Chuen Her, *i386/i486 Advanced Programming*, Van Nostrand Reinhold, New York, 1993.
- Tom Shanley, *Protected Mode System Architecture*, Addison Wesley, NY, 1996.
- SGS-Thomson Corporation, *80486DX Processor SMM Programming Manual*, SGS-Thomson Corporation, 1995.
- Walter A. Triebel, *The 80386DX Microprocessor*, Prentice-Hall, Englewood Cliffs, NJ, 1992.
- John Wharton, *The Complete x86*, MicroDesign Resources, Sebastopol, California, 1994.
- Web sites and newsgroups:
  - www.amd.com
  - news.comp.arch
  - news.comp.lang.asm.x86
  - news.intel.microprocessors
  - news.microsoft

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 116.

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 full backward-compatibility with existing software.

1.1 Memory Model

The AMD64 architecture memory model allows 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

```
Logical Address = Segment Selector : 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 6 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

```
Effective Address = Base + (Scale x Index) + 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 5).

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 5.

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 8 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.

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 11. Real mode is available only from legacy mode (see “Legacy Modes” on page 16).
- *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 6) and page translation (see "Paging" on page 8) 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 147), 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 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 supports 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 shows an example of segmented memory. Segmentation is described in Chapter 4, "Segmented Virtual Memory."

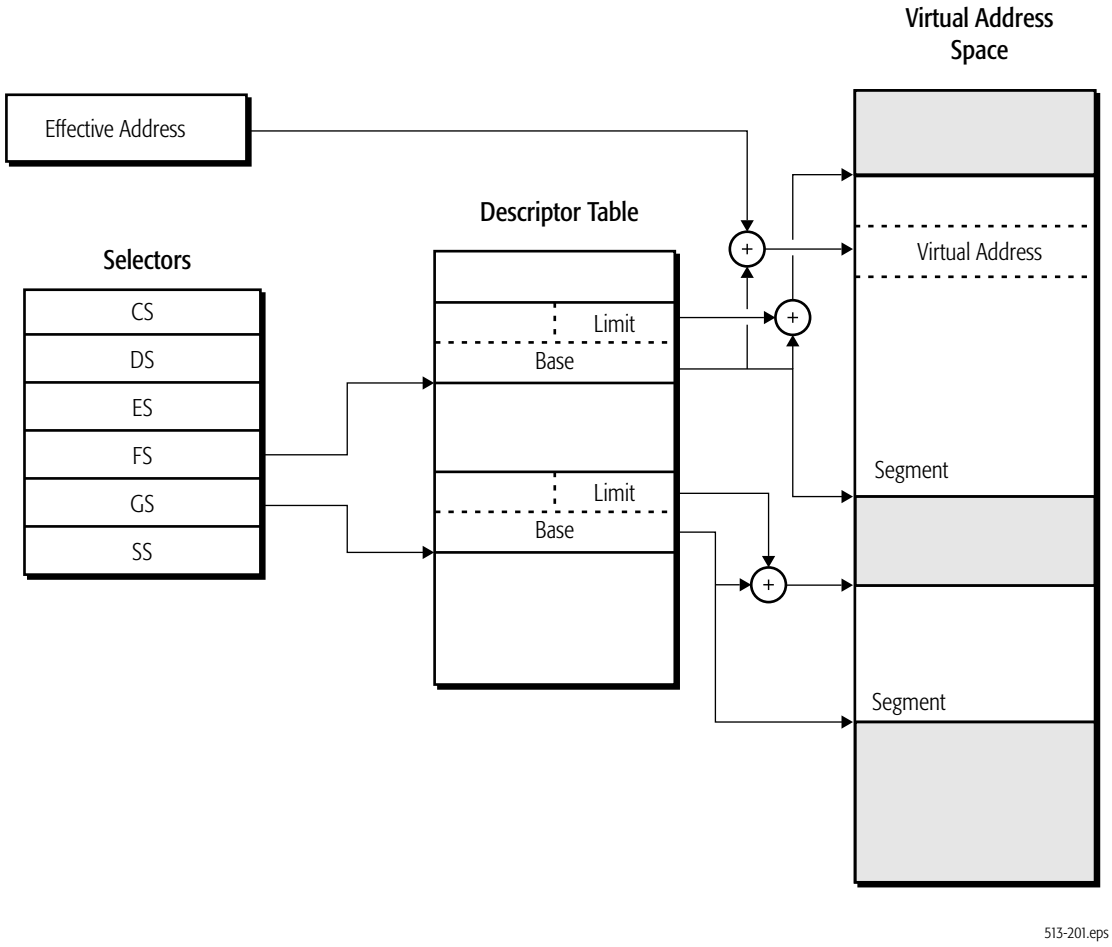


Figure 1-1. Segmented-Memory Model

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 8 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.

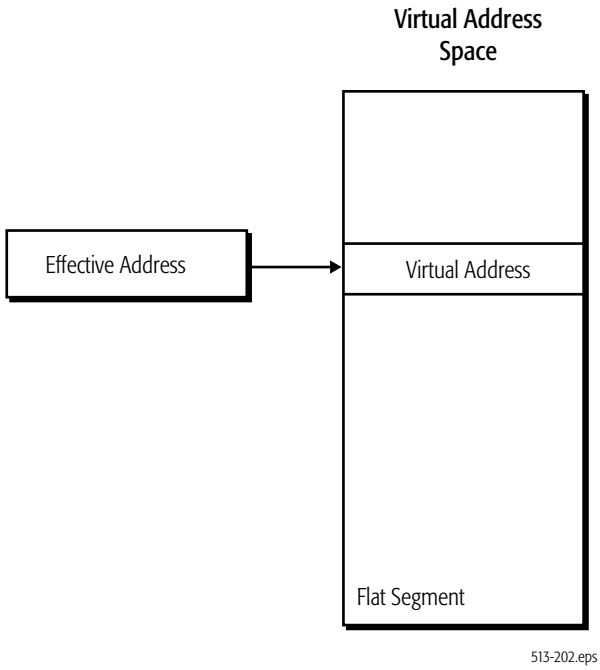


Figure 1-2. Flat Memory Model

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 shows an example of paged memory with three levels in the translation-table hierarchy. Paging is described in Chapter 5, “Page Translation and Protection.”

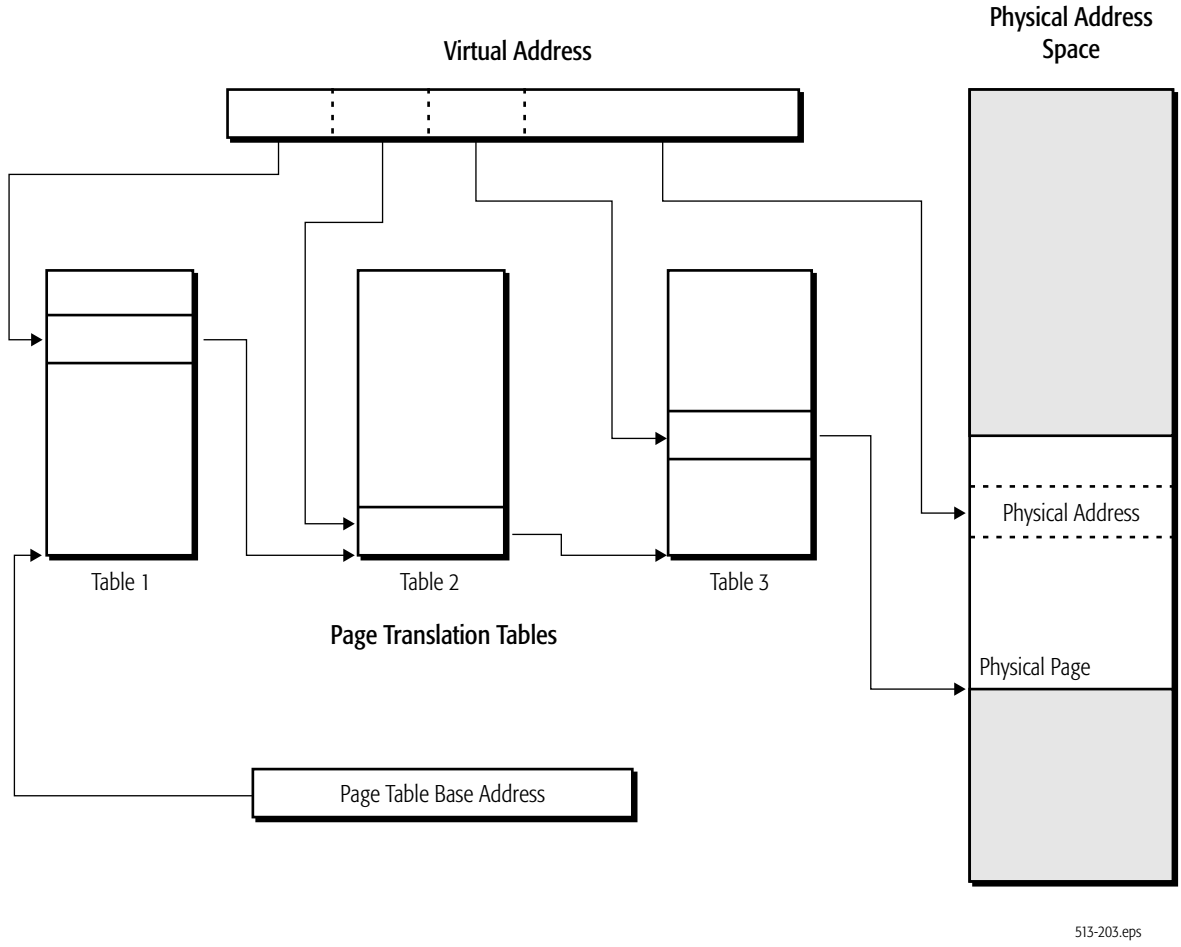


Figure 1-3. Paged Memory Model

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 on page 11 shows an example of this model.

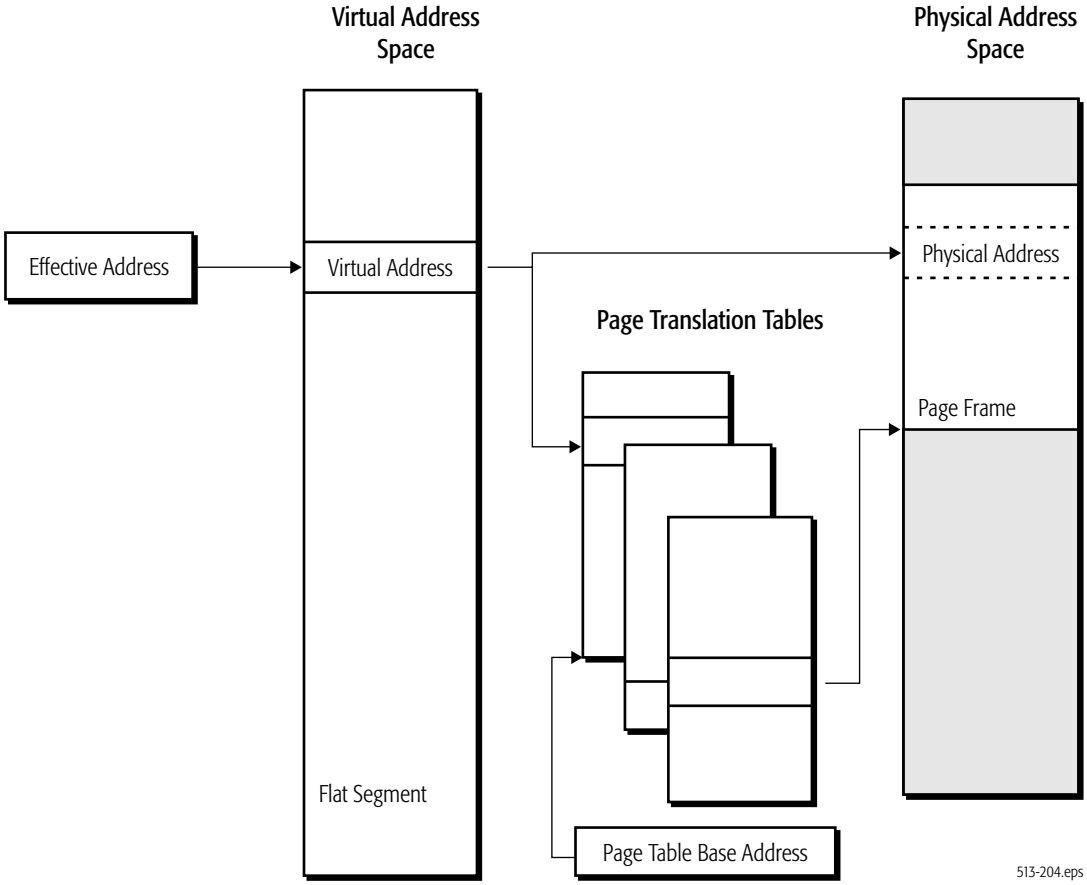


Figure 1-4. 64-Bit Flat, Paged-Memory Model

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.

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

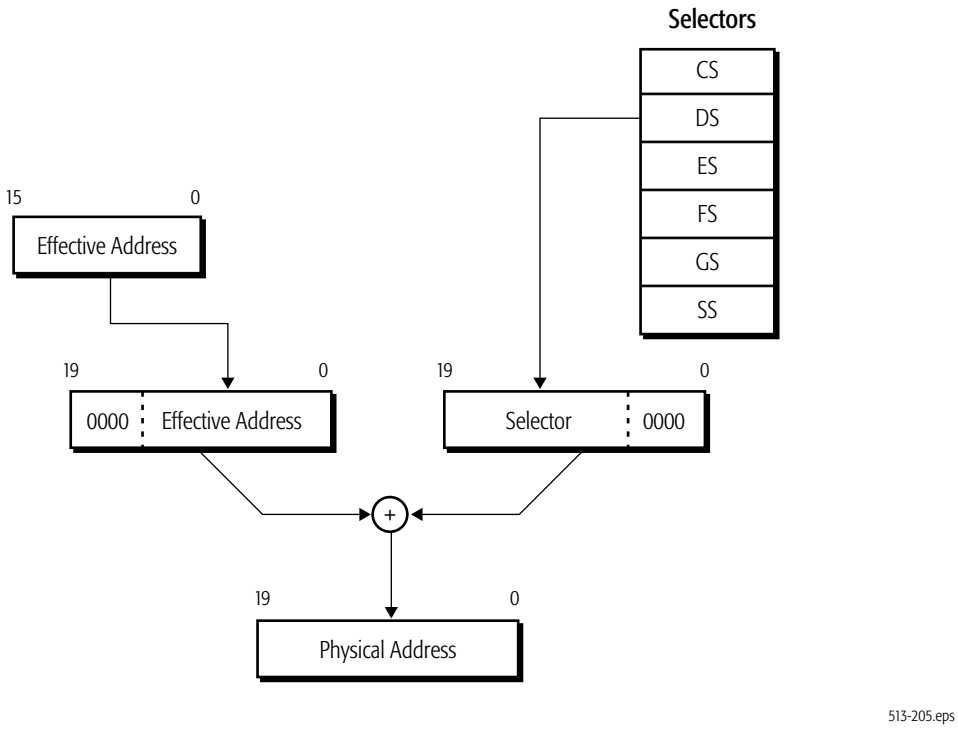


Figure 1-5. Real-Address Memory Model

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 14. Each operating mode is described in the following sections.

Table 1-1. Operating Modes

Mode		System Software Required	Application Recompile Required	Defaults[1]		Register Extensions[2]	Maximum GPR Width (bits)
				Address Size (bits)	Operand Size (bits)		
Long Mode[3]	64-Bit Mode	New 64-bit OS	yes	64	32	yes	64
	Compatibility Mode		no	32		no	32
				16	16		
Legacy Mode	Protected Mode	Legacy 32-bit OS	no	32	32	no	32
				16	16		
	Virtual-8086 Mode			16	16		32
	Real Mode	Legacy 16-bit OS					

Note:

1. *Defaults can be overridden in most modes using an instruction prefix or system control bit.*
2. *Register extensions includes eight new GPRs and eight new XMM registers (also called SSE registers).*
3. *Long mode supports only x86 protected mode. It does not support x86 real mode or virtual-8086 mode.*

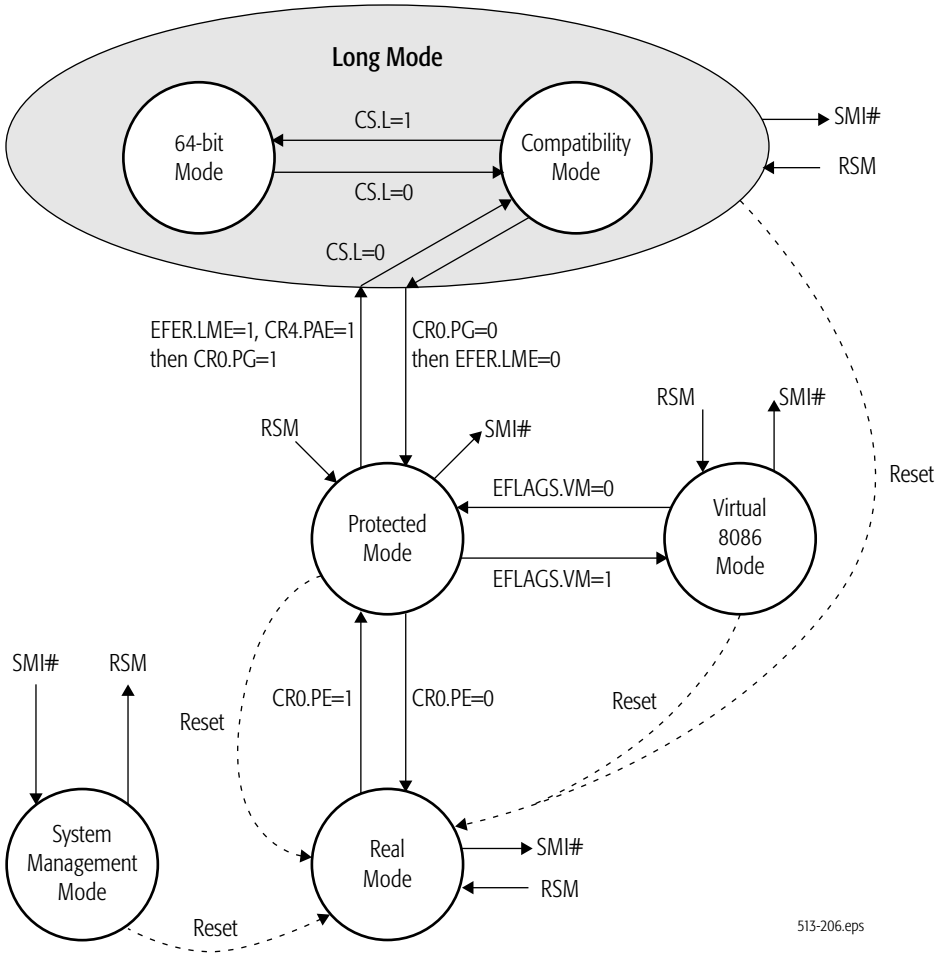


Figure 1-6. Operating Modes of the AMD64 Architecture

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 new features:

- 64-bit virtual addresses (processor implementations can have fewer).
- Register extensions through a new instruction prefix (REX):
  - Adds eight GPRs (R8–R15).
  - Widens GPRs to 64 bits.
  - Adds eight 128-bit streaming SIMD extension (SSE) registers (XMM8–XMM15).
- 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 13.

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

L*egacy mode* consists of three submodes: real mode, protected mode, and virtual-8086 mode. Protected mode can be either paged or unpaged. L*egacy 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 the basic input-output system (BIOS) 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 19 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 19, the system registers include:

- *Control Registers*—These registers are used to control system operation and some system features. See "System-Control Registers" on page 51 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 62 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 87 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 361 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 383 for details.

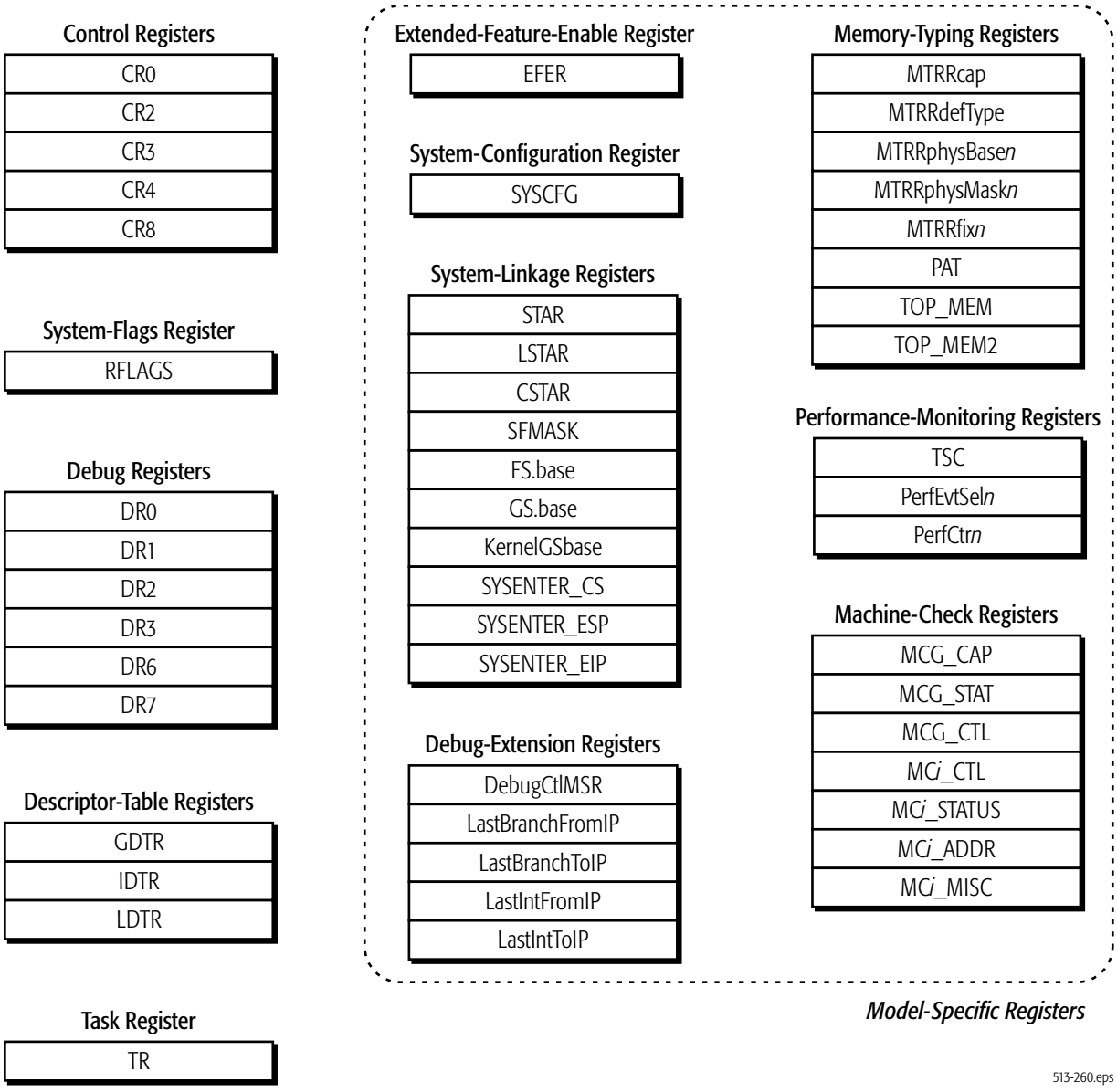


Figure 1-7. System Registers

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*n* control registers. In particular, EFER

is used to control activation of long mode. See "Extended Feature Enable Register (EFER)" on page 67 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 70 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 179 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 217 for details.
- *Debug-Extension Registers*—These registers control additional software-debug reporting features. See "Debug Registers" on page 383 for details.
- *Performance-Monitoring Registers*—Performance-monitoring registers are used to count processor and system events, or the duration of events. See "Performance Optimization" on page 401 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 MSRs" on page 305 for more information.

1.5 System-Data Structures

Figure 1-8 on page 21 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.

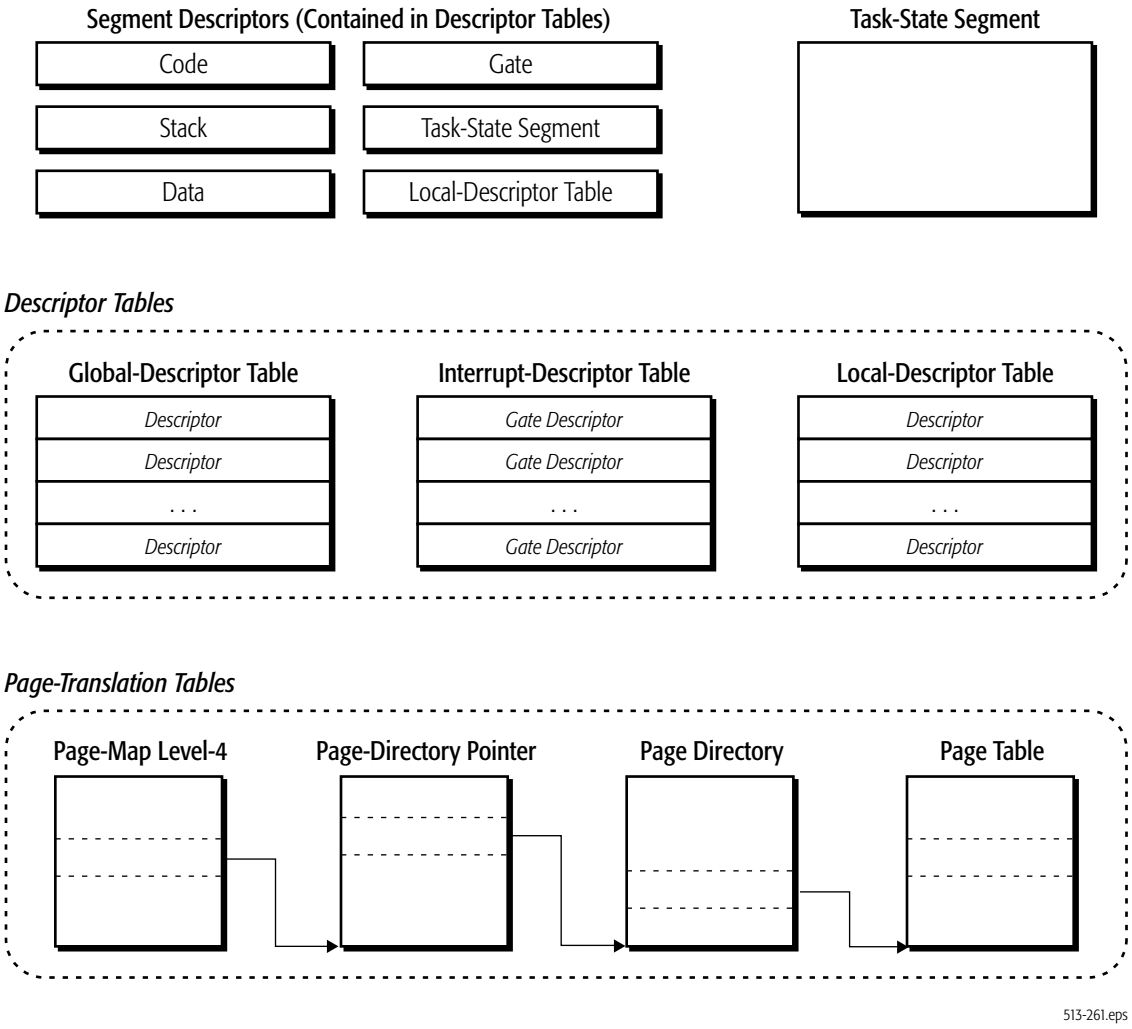


Figure 1-8. System-Data Structures

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 95, and “Long-

Mode Segment Descriptors" on page 106 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 87 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 363, and "64-Bit Task State Segment" on page 368 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 144 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 274, and "Long-Mode Interrupt Control Transfers" on page 285.

Table 1-2 on page 24 shows the supported interrupts and exceptions, ordered by their vector number. Refer to "Vectors" on page 245 for a complete description of each interrupt, and a description of the interrupt mechanism.

Table 1-2. Interrupts and Exceptions

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

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 Mechanism." Much of the error-reporting capabilities is implementation dependent. For more information, developers of machine-check error-reporting software should also refer to the BIOS writer's guide for a specific implementation.

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 382. Some processors provide additional, implementation-specific debug support. For more information, refer to the BIOS writer's guide for the specific implementation.

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 Optimization” on page 401. The specific events that can be monitored are generally implementation specific. For more information, refer to the BIOS writer’s guide for the specific implementation.

2 x86 and AMD64 Architecture Differences

The AMD64 architecture provides 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 12 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. L*egacy 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.

See Chapter 10, "System-Management Mode," for more information on the SMM differences.

2.2 Memory Model

The AMD64 architecture provides significant 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.

Implementations can support fewer than 52 physical-address bits. The first implementation of the AMD64 architecture, for example, supports 40-bit physical addressing in both long mode and legacy mode.

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 158 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. (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 148 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, and LTR 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 109 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 111 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 172 and "Segment-Protection Overview" on page 116 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 37 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.

2.4.2 128-Bit Media Registers

In 64-bit mode, eight additional 128-bit XMM registers are available, XMM8–XMM15. A REX instruction prefix is used to access these registers. In compatibility and legacy modes, the XMM registers consist of the eight 128-bit legacy registers, XMM0–XMM7.

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 (CR*n*). 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 a new family of instruction-prefix bytes used in 64-bit mode to:

- Specify the new GPRs and 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 39 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 86 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. See "Operands" in Volume 1 for details.

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. 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 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 on page 40 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

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

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 on page 41 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 2-2. 64-Bit Mode Near Branches, Default 64-Bit Operand Size

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

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 180 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 183 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 182 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 "General-Purpose Instruction Reference" 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 45 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

Mnemonic	Opcode (hex)	Description
AAA	37	ASCII Adjust After Addition
AAD	D5	ASCII Adjust Before Division
AAM	D4	ASCII Adjust After Multiply
AAS	3F	ASCII Adjust After Subtraction
BOUND	62	Check Array Bounds
CALL (far)	9A	Procedure Call Far (far absolute)
DAA	27	Decimal Adjust after Addition
DAS	2F	Decimal Adjust after Subtraction
INTO	CE	Interrupt to Overflow Vector
JMP (far)	EA	Jump Far (absolute)
LAHF	9F	Load Status Flags into AH Register
LDS	C5	Load DS Segment Register
LES	C4	Load ES Segment Register
POP DS	1F	Pop Stack into DS Segment
POP ES	07	Pop Stack into ES Segment
POP SS	17	Pop Stack into SS Segment
POPA, POPAD	61	Pop All to GPR Words or Doublewords
PUSH CS	0E	Push CS Segment Selector onto Stack
PUSH DS	1E	Push DS Segment Selector onto Stack

Table 2-3. Invalid Instructions in 64-Bit Mode (continued)

Mnemonic	Opcode (hex)	Description
PUSH ES	06	Push ES Segment Selector onto Stack
PUSH SS	16	Push SS Segment Selector onto Stack
PUSHA, PUSHAD	60	Push All GPR Words or Doublewords onto Stack
SAHF	9E	Store AH into Flags
Redundant Grp1 (undocumented)	82	Redundant encoding of group1 Eb,Ib opcodes
SALC (undocumented)	D6	Set AL According to CF

Table 2-4. Invalid Instructions in Long Mode

Mnemonic	Opcode (hex)	Description
SYSENTER	0F 34	System Call
SYSEXIT	0F 35	System Return

Table 2-5 lists the instructions that are no longer valid in 64-bit mode because their opcodes have been reassigned. The reassigned opcodes are used in 64-bit mode as REX instruction prefixes.

Table 2-5. Reassigned Instructions in 64-Bit Mode

Mnemonic	Opcode (hex)	Description
ARPL	63	Opcode for MOVSXD instruction in 64-bit mode. In all other modes, this the Adjust Requestor Privilege Level instruction opcode.
DEC and INC	40-4F	Decrement by 1, Increment by 1. Two-byte versions of DEC and INC are still valid.

2.5.11 FXSAVE and FXRSTOR Instructions

The FXSAVE and FXRSTOR instructions are used to save and restore the entire 128-bit media, 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–XMM15 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.

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 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 285 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 270 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 245 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 360 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 via 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 368 for additional information.

2.8 Long-Mode vs. Legacy-Mode Differences

Table 2-6 on page 50 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

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

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 "Segmentation Data Structures and Registers" on page 80.
- Page-translation tables are described in "Legacy-Mode Page Translation" on page 148 and "Long-Mode Page Translation" on page 158.
- The task-state segment is described in "Legacy Task-State Segment" on page 363 and "64-Bit Task State Segment" on page 368.

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 CR*n* 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.

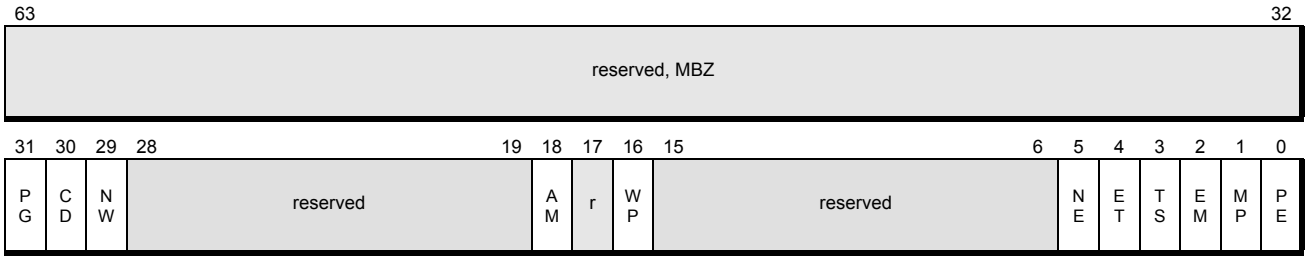
In 64-bit mode, the high 32 bits of CR0 and CR4 are reserved and must be written with zeros. Writing a 1 to any of the high 32 bits results in a general-protection exception, #GP(0). All 64 bits of CR2 are writable. However, the MOV CR*n* instructions *do not* check that addresses written to CR2 are 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. The legacy CR0 register is identical to the low 32 bits of the register shown in Figure 3-1 (CR0 bits 31–0).



Bits	Mnemonic	Description	R/W
63–32	reserved	Reserved, Must be Zero	
31	PG	Paging	R/W
30	CD	Cache Disable	R/W
29	NW	Not Writethrough	R/W
28–19	reserved	Reserved	
18	AM	Alignment Mask	R/W
17	reserved	Reserved	
16	WP	Write Protect	R/W
15-6	reserved	Reserved	
5	NE	Numeric Error	R/W
4	ET	Extension Type	R
3	TS	Task Switched	R/W
2	EM	Emulation	R/W
1	MP	Monitor Coprocessor	R/W
0	PE	Protection Enabled	R/W

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 "Segment-Protection Overview" on page 116 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. 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 “Page-Protection Checks” on page 172 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 eFLAGS.AC=1. Alignment checking can be disabled by clearing either AM or eFLAGS.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 writing back and invalidating the caches before setting CD to 1 (this avoids caching the instructions that set CD to 1).

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 "Page-Level Writethrough (PWT) Bit" on page 168 and "Page-Level Cache Disable (PCD) Bit" on page 168 for information on page-level cache control.

See "Memory Caches" on page 206 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 "Page Translation Overview" on page 144 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 Figures 3-2 and 3-3, and the CR3 (page-translation-table base address) register, shown in Figures 3-4, 3-5, and 3-6, are used only by the page-translation mechanism.

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

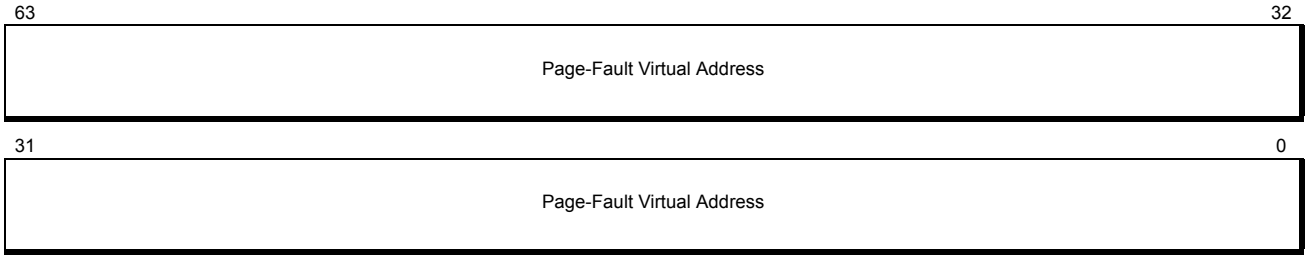


Figure 3-3. Control Register 2 (CR2)—Long Mode

See “CR2 Register” on page 259 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.

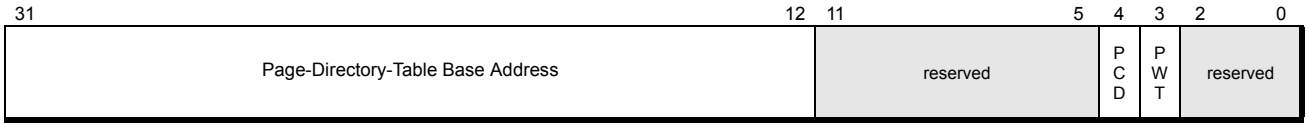


Figure 3-4. Control Register 3 (CR3)—Legacy-Mode Non-PAE Paging

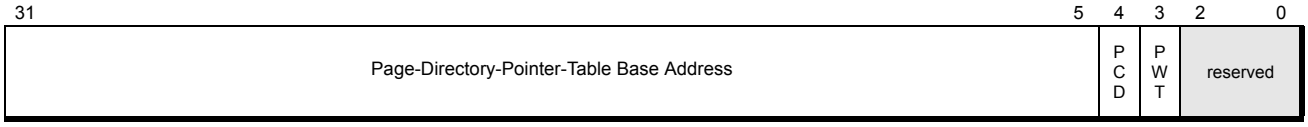


Figure 3-5. Control Register 3 (CR3)—Legacy-Mode PAE Paging

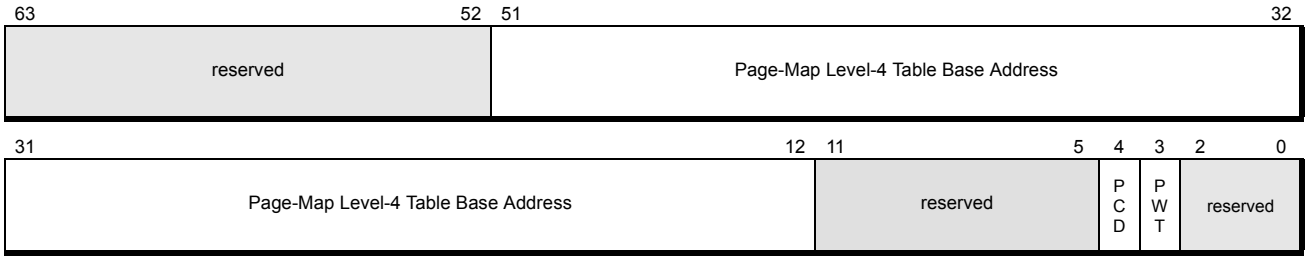
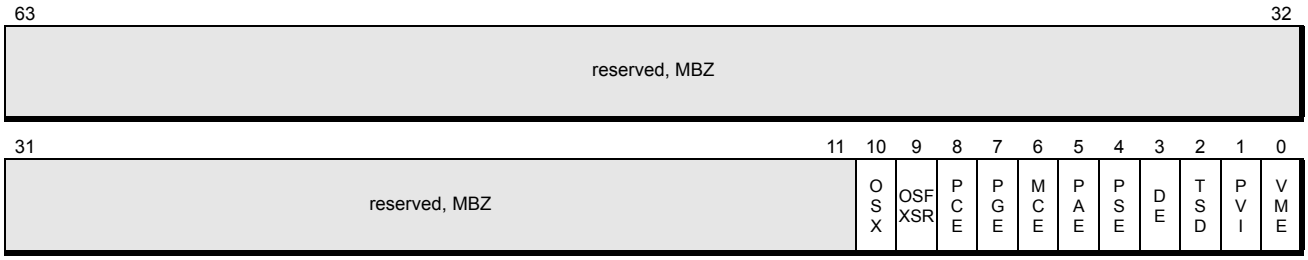


Figure 3-6. Control Register 3 (CR3)—Long Mode

The legacy CR3 register is described in "CR3 Register" on page 149, and the long-mode CR3 register is described in "CR3" on page 159.

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 shown in Figure 3-7 (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.



Bits	Mnemonic	Description	R/W
63–11	reserved	Reserved, Must be Zero	
10	OSXMMEX-CPT	Operating System Unmasked Exception Support	R/W
9	OSFXSR	Operating System FXSAVE/FXRSTOR Support	R/W
8	PCE	Performance-Monitoring Counter Enable	R/W
7	PGE	Page-Global Enable	R/W
6	MCE	Machine Check Enable	R/W
5	PAE	Physical-Address Extension	R/W
4	PSE	Page Size Extensions	R/W
3	DE	Debugging Extensions	R/W
2	TSD	Time Stamp Disable	R/W
1	PVI	Protected-Mode Virtual Interrupts	R/W
0	VME	Virtual-8086 Mode Extensions	R/W

Figure 3-7. Control Register 4 (CR4)

The function of the CR4 control bits are (all bits are read/write):

Virtual-8086 Mode Extensions (VME) Bit. 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 (INT*n* instructions) using the interrupt-redirection bitmap in the TSS.

Protected-Mode Virtual Interrupts (PVI) Bit. 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 INT*n* interception.

PVI enhancements are also supported in long mode. See "Virtual Interrupts" on page 293 for more information on using PVI.

Time-Stamp Disable (TSD) Bit. 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 instruction. When TSD is set to 1, only software running at privilege-level 0 can execute the RDTSC instruction.

Debugging Extensions (DE) Bit. 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. 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 "4-Mbyte Page Translation" on page 152 for more information on 4-Mbyte page translation.

Physical-Address Extension (PAE) Bit. 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 "PAE Paging" on page 154 for more information on physical-address extensions.

Machine-Check Enable (MCE) Bit. 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 Mechanism," 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. 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 "Global Pages" on page 171 for information on using the global-page mechanism.

Performance-Monitoring Counter Enable (PCE) Bit. 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 four performance-monitoring MSRs, PerfCTR[3:0]. Clearing PCE to 0 allows only the most-privileged software (CPL=0) to use the RDPMC instruction.

FXSAVE/FXRSTOR Support (OSFXSR) Bit. Bit 9. System software must set the OSFXSR bit to 1 to enable use of the 128-bit media 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 128-bit media 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. Bit 10. System software must set the OSXMMEXCPT bit to 1 when it supports the SIMD floating-point exception (#XF) for handling of unmasked 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 128-bit media unmasked floating-point exceptions.

3.1.4 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).

3.1.5 64-Bit-Mode Extended Control Registers

In 64-bit mode, additional encodings for control registers are available. 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 Prefix-Byte Fields" 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).

3.1.6 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 "External Interrupt Priorities" on page 270 for a description of the TPR and how system software can use the TPR for controlling external interrupts.

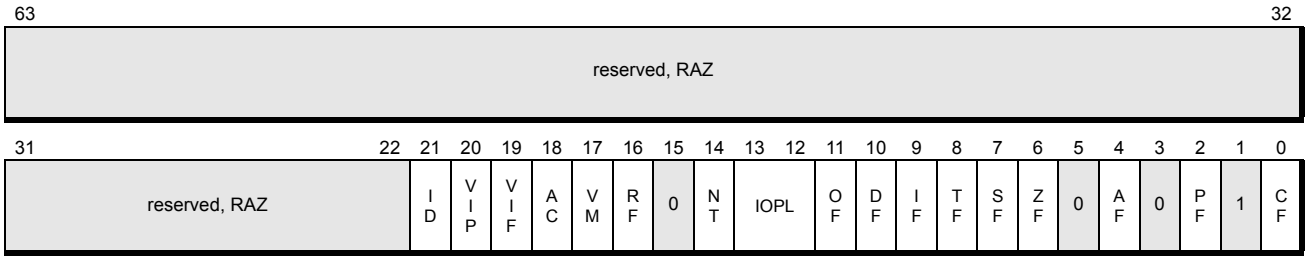
3.1.7 RFLAGS Register

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 3-8 on page 63 shows the format of the RFLAGS register. The legacy EFLAGS register is identical to the low 32 bits of the register shown in Figure 3-8 (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.



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

Figure 3-8. 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, a debug exception (#DB) occurs after each instruction completes execution. Single stepping begins with the instruction *following* the instruction that sets TF. Single stepping is disabled (TF=0) when the #DB exception occurs or when any exception or interrupt occurs.

See “Single Stepping” on page 398 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 “Masking External Interrupts” on page 243 for information on interrupt masking. See “Accessing the RFLAGs Register” on page 185 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 “Virtual Interrupts” on page 293 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 "Task Switches Using Task Gates" on page 375 for information on switching tasks using task gates, and "Nesting Tasks" on page 377 for information on task nesting.

Resume Flag (RF) Bit. Bit 16. The RF bit allows an instruction to be restarted following an instruction breakpoint resulting in a debug exception (#DB). This bit prevents multiple debug exceptions from occurring on the same instruction.

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 “Virtual Interrupts” on page 293 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

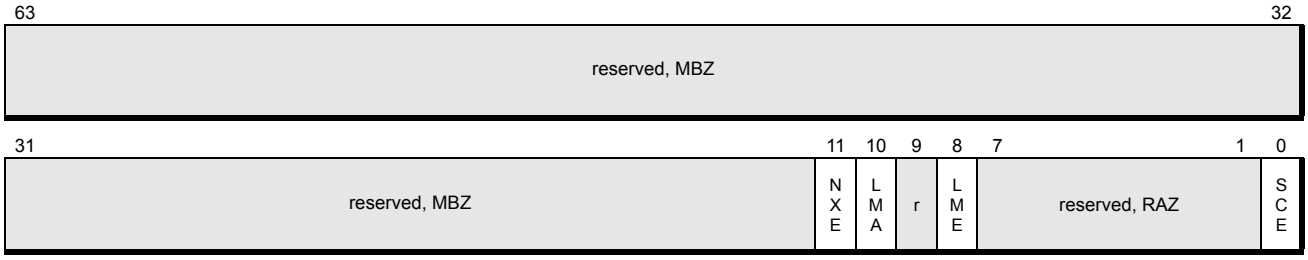
an external, maskable interrupt. Before returning to the virtual-8086 mode application, the service routine sets VIP to 1 if EFLAGS.VIF=1. When the virtual-8086 mode application attempts to enable interrupts by clearing EFLAGS.VIF to 0 while VIP=1, a general-protection exception (#GP) occurs. The #GP 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 "Virtual Interrupts" on page 293 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 "Processor Feature Identification" on page 75 for more information on the CPUID instruction.

3.1.8 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 "Model-Specific Registers (MSRs)" on page 69 for more information on MSRs). It can be read and written only by privileged software. Figure 3-9 on page 68 shows the format of the EFER register.



Bits	Mnemonic	Description	R/W
63–12	reserved, MBZ	Reserved, Must be Zero	
11	NXE	No-Execute Enable	R/W
10	LMA	Long Mode Active	R
9	reserved, MBZ	Reserved, Must be Zero	
8	LME	Long Mode Enable	R/W
7-1	reserved, RAZ	Reserved, Read as Zero	
0	SCE	System Call Extensions	R/W

Figure 3-9. Extended Feature Enable Register (EFER)

The function of the EFER bits are (unless otherwise noted, all bits are read/write):

System-Call Extension (SCE) Bit. Bit 0. 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 “Fast System Call and Return” on page 179 for additional information.

Long Mode Enable (LME) Bit. Bit 8. 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-only. 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 14.

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.

No-Execute Enable (NXE) Bit. Bit 11. Setting this bit to 1 enables the no-execute page-protection feature. The feature is disabled when this bit is cleared to 0. See "No Execute (NX) Bit" on page 173 for more information.

Before setting NXE, system software should verify the processor supports the feature by examining the extended-feature flags returned by the CPUID instruction. For more information, see "Function 8000_0001h: Processor Signature and AMD Features" in Volume 3.

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 3-10. The EFER register—described in "Extended Feature Enable Register (EFER)" on page 67—is also an MSR.

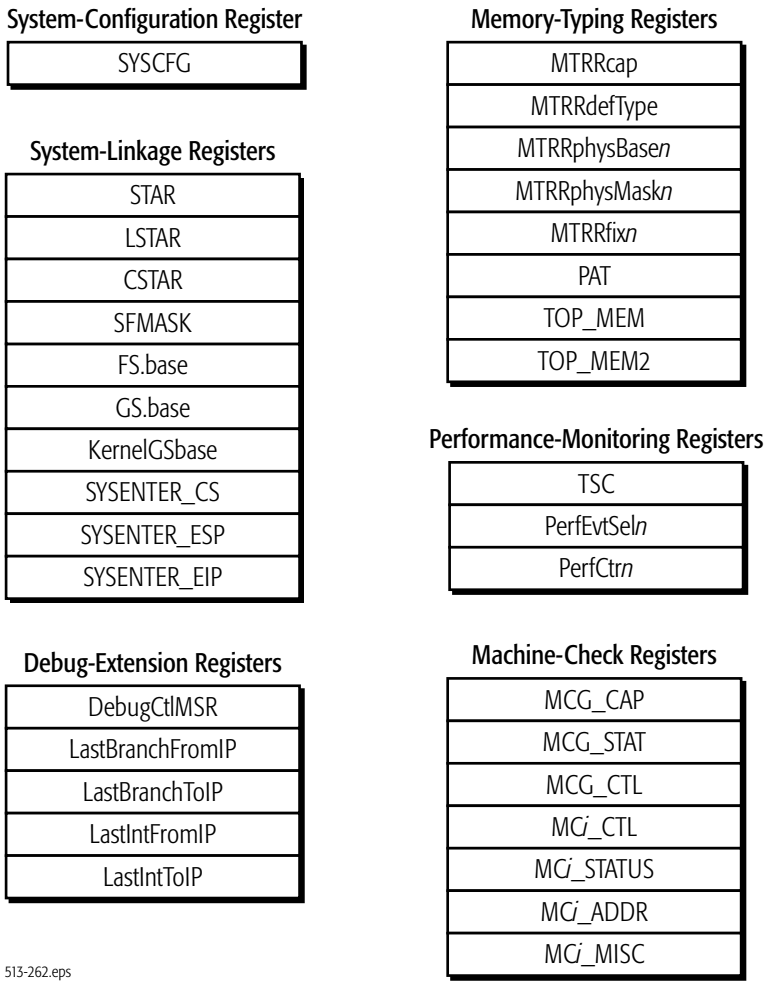


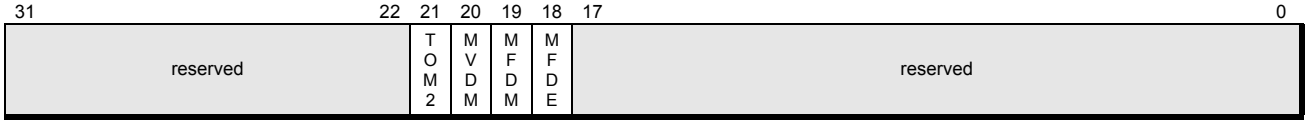
Figure 3-10. AMD64 Architecture Model-Specific Registers

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 3-11 shows the format of the SYSCFG

register. Some features are implementation specific, and are described in the BIOS writer's guide for the implementation. Implementation-specific features are not shown in Figure 3-11.



Bits	Mnemonic	Description	R/W
31-22	reserved		
21	TOM2	MtrrTom2En	R/W
20	MVDM	MtrrVarDramEn	R/W
19	MFDM	MtrrFixDramModEn	R/W
18	MFDE	MtrrFixDramEn	R/W
17-0	reserved		

Figure 3-11. System-Configuration Register (SYSCFG)

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 "Extended Fixed-Range MTRR Type-Field Encodings" on page 233 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 "Extended Fixed-Range MTRR Type-Field Encodings" on page 233 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 "IORRs" on page 235 and "Top of Memory" on page 237 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 "Top of Memory" on page 237 for information on using this feature.

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 "FS and GS Registers in 64-Bit Mode" on page 86 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.

SYSENTER*x* 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 "Fast System Call and Return" on page 179.

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 Mbytes 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 "Memory-Type Range Registers" on page 215 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:

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

LastBranch*x* and LastException*x* Registers. The four registers, LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP, are all used to record the source and target of control transfers when branch recording is enabled.

Refer to "Control-Transfer Breakpoint Features" on page 399 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. There can be any number of performance counters, each numbered from 0 to *n*. 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 *read time-stamp counter* instruction, RDTSC. System software can make RDTSC available for use by non-privileged software by clearing the time-stamp disable bit (CR4.TSD) to 0.

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

PerfCtr*n* Registers. These registers are performance counters that hold a count of processor events or the duration of events, under the control of the corresponding PerfEvtSel*n* register. Each PerfCtr*n* 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 "Using Performance Counters" on page 406 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 3-10 on page 70.

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 "Using Machine Check Features" on page 314 for more information on using these registers.

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. After the information is collected, software can be tuned to optimize performance and benefit to users. For example, game software can identify and enable the media capabilities of a particular processor implementation.

The CPUID instruction supports multiple functions, each providing different 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.

Function codes are loaded into the EAX register before executing the CPUID instruction. CPUID functions are divided into two types:

- *Standard functions* include the earliest features offered in the x86 architecture.
- *Extended functions* include newer features of the x86 and AMD64 architectures, such as SSE, SSE2, and 3DNow! instructions, and long mode.

See “CPUID” in Volume 3 for details on the operation of this instruction, and the *AMD AMD64 Processor Recognition Application Note Addendum* (order no. 25481) for information returned by each processor implementation.

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 11, 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 selectors 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 selector 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 272.

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 selector 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, 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 7 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 8 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 86 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 81 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 274, and in "Long-Mode Interrupt Control Transfers" on page 285.

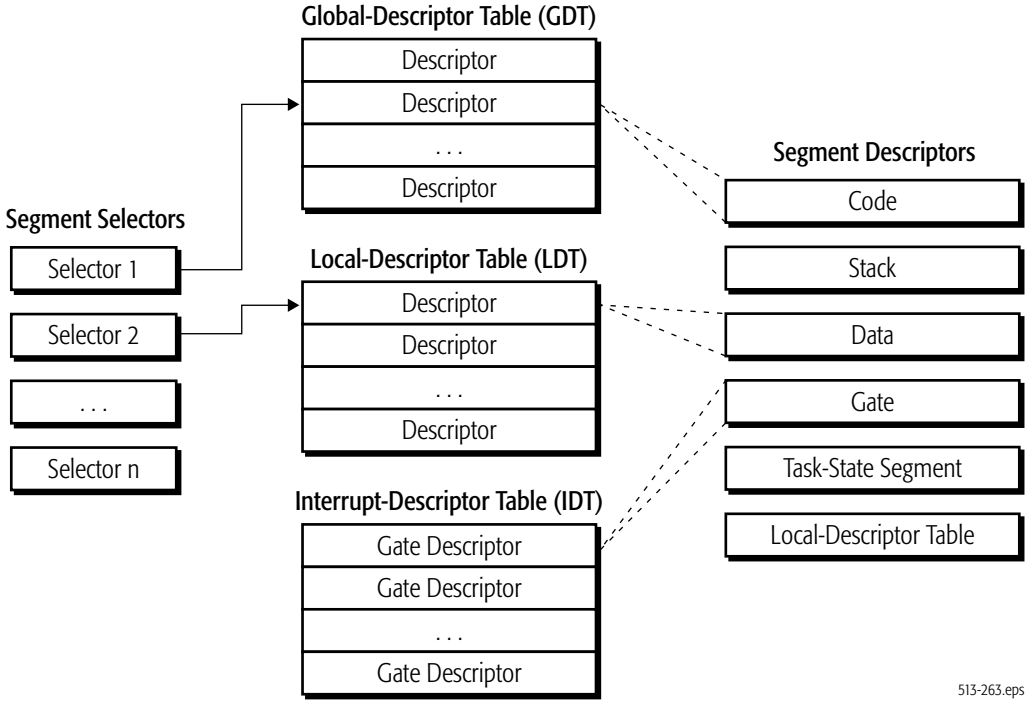


Figure 4-1. Segmentation Data Structures

Figure 4-2 on page 82 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).

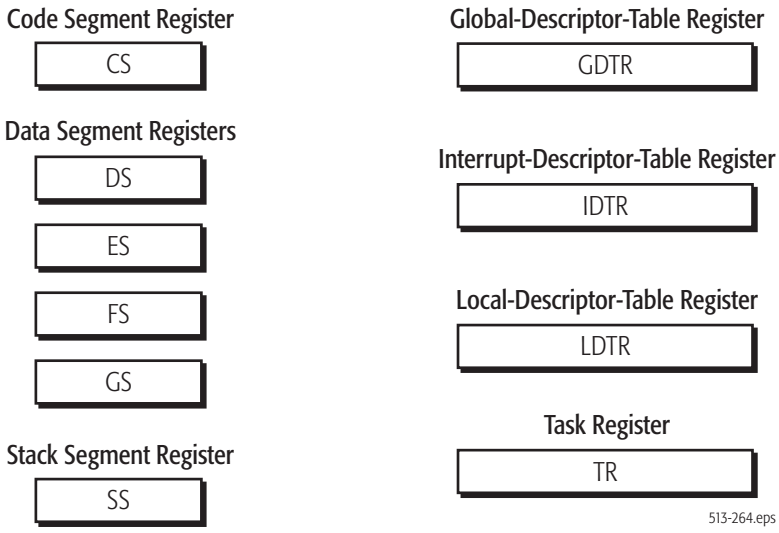


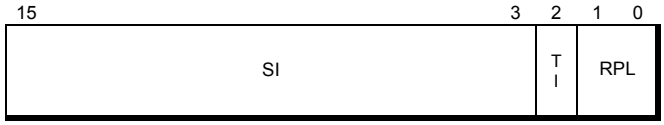
Figure 4-2. Segment and Descriptor-Table Registers

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 358.

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 on page 83 shows the segment selector format.



Bits	Mnemonic	Description	R/W
15-3	SI	Selector Index	R/W
2	TI	Table Indicator	R/W
1-0	RPL	Requestor Privilege Level	R/W

Figure 4-3. Segment Selector

The selector format consists of the following fields:

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 95 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 119 and “Control-Transfer Privilege Checks” on page 122 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. 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 or SS register with a null selector.

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 lists the supported segment registers and their functions.

Table 4-1. Segment Registers

Segment Register	Segment Register Function
CS	References code-segment descriptor entry
DS	References default data-segment descriptor entry
ES	References optional data-segment descriptor entry
FS	References optional data-segment descriptor entry
GS	References optional data-segment descriptor entry
SS	References stack segment descriptor entry

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).

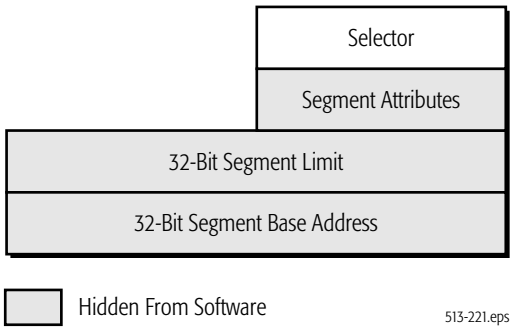


Figure 4-4. Segment-Register Format

CS Register. The CS register is loaded with 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 is loaded with the segment selector referencing the default data-segment descriptor entry. The SS register is loaded with 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.

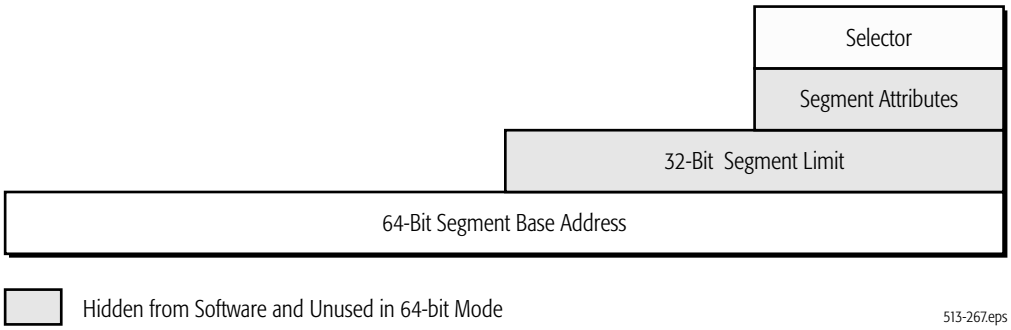


Figure 4-5. FS and GS Segment-Register Format—64-Bit Mode

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.

To allow loading all 64 bits of the base address, the FS.base and GS.base hidden descriptor-register fields are mapped to MSRs. Privileged software (CPL=0) can load the 64-bit base address into FS.base or GS.base using a single WRMSR instruction. The addresses written into the expanded FS.base and GS.base registers must be in canonical form. A WRMSR instruction that attempts to write a non-canonical address to these registers causes a general-protection exception (#GP) to occur.

The FS.base MSR address is C000_0100h while the GS.base MSR address is C000_0101h.

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 89 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 84 for information on using the null selector). The first usable GDT entry is referenced with a selector index of 1.

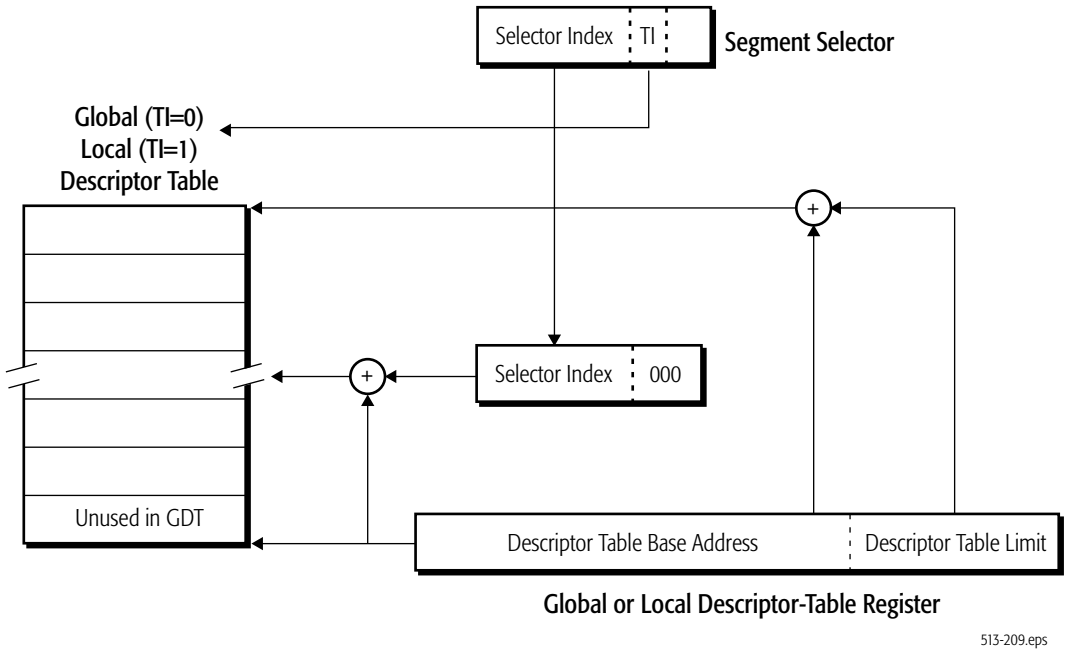


Figure 4-6. Global and Local Descriptor-Table Access

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 187). Figure 4-7 shows the format of the GDTR in legacy mode and compatibility mode.

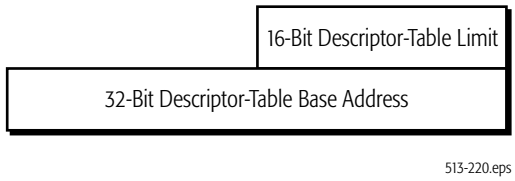


Figure 4-7. GDTR and IDTR Format—Legacy Modes

Figure 4-8 on page 90 shows the format of the GDTR in 64-bit mode.

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

The GDTR contains 2 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 doubleword 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 89). 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 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.

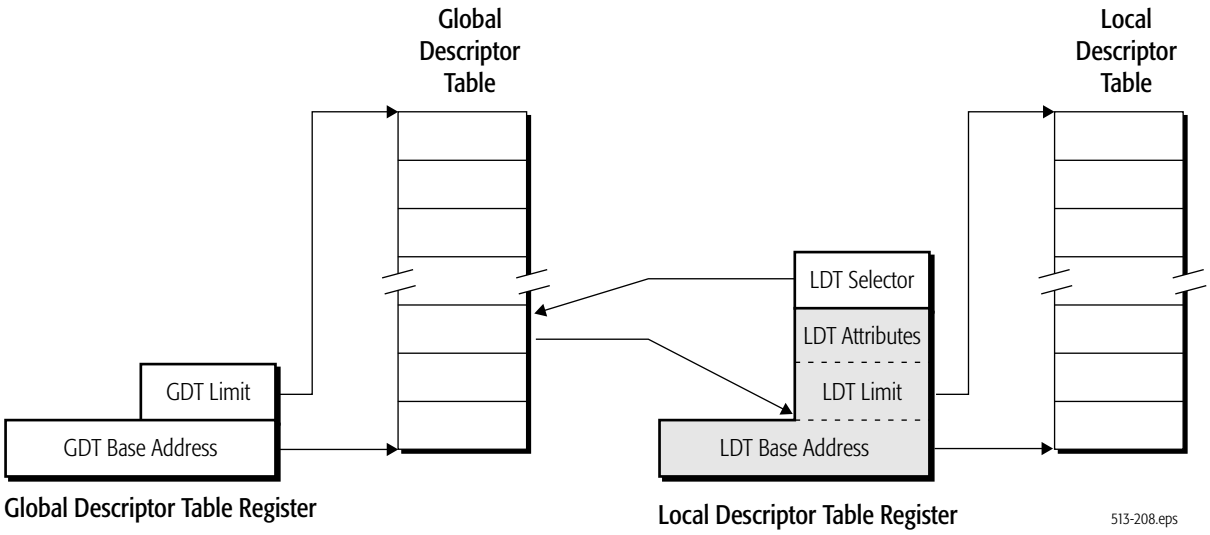


Figure 4-9. Relationship between the LDT and GDT

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 187).
- Performing a task switch (see "Switching Tasks" on page 373).

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

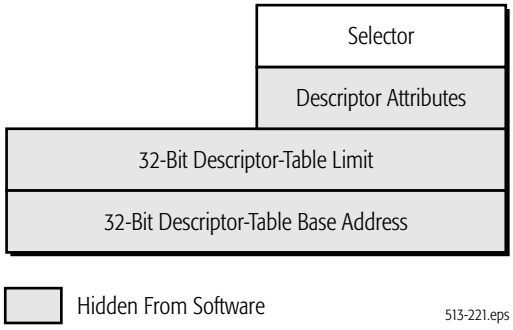


Figure 4-10. LDTR Format—Legacy Mode

Figure 4-11 shows the format of the LDTR in long mode (both compatibility mode and 64-bit mode).

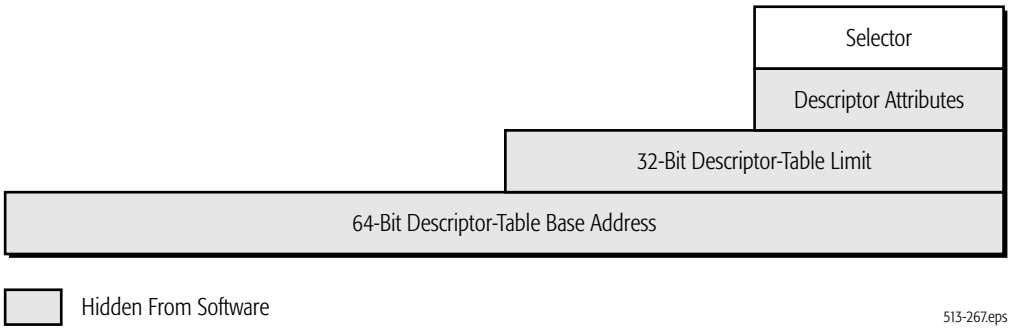


Figure 4-11. LDTR Format—Long Mode

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 Figures 4-10 and 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 doubleword 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 114 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 187 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 98.

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 on page 95 shows how the interrupt-vector number indexes the IDT.

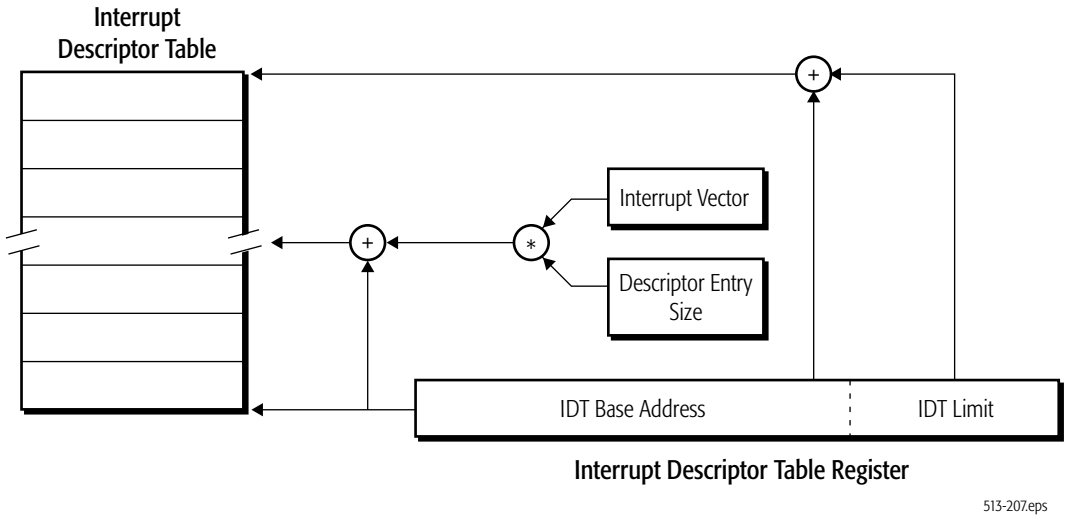


Figure 4-12. Indexing an IDT

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 187). The format of the IDTR is identical to that of the GDTR in all modes. Figure 4-7 on page 89 shows the format of the IDTR in legacy mode. Figure 4-8 on page 90 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 104.

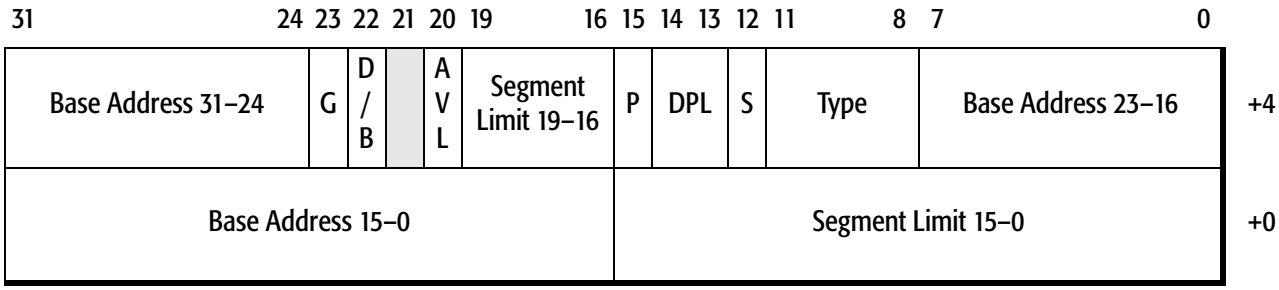


Figure 4-13. Generic Segment Descriptor—Legacy Mode

Figure 4-13 shows the fields in a generic, legacy-mode, 8-byte segment descriptor. In this figure, +0 indicates the address of the descriptor's first byte, and +4 indicates the address of the descriptor's fifth byte. The fields are defined as follows, from least-significant to most-significant bit positions:

Segment Limit. The 20-bit segment limit is formed by concatenating bits 19–16 of byte +4 with bits 15–0 of byte +0. 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 98). 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 101).

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 byte +4 with bits 7–0 of byte +4, and with bits 15–0 of byte +0. The segment-base address field locates the start of a segment in virtual-address space.

S Bit and Type Field. Bit 12 of byte +4, and bits 11–8 of byte +4. 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 4-2. Descriptor Types

S Field	Descriptor Type	Type-Field Encoding
0 (System)	LDT	See Table 4-5 on page 103
	TSS	
	Gate	
1 (User)	Code	See Table 4-3 on page 100
	Data	See Table 4-4 on page 102

Descriptor Privilege-Level (DPL) Field. Bits 14–13 of byte +4. 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 119 and “Control-Transfer Privilege Checks” on page 122 for more information on how the DPL is used during segment privilege-checks.

Present (P) Bit. Bit 15 of byte +4. 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 byte +4. 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 byte +4. 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 100 for a description of the D bit in code-segment descriptors. "Data-Segment Default Operand Size (D/B) Bit" on page 102 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 byte +4. 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 on page 99 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.

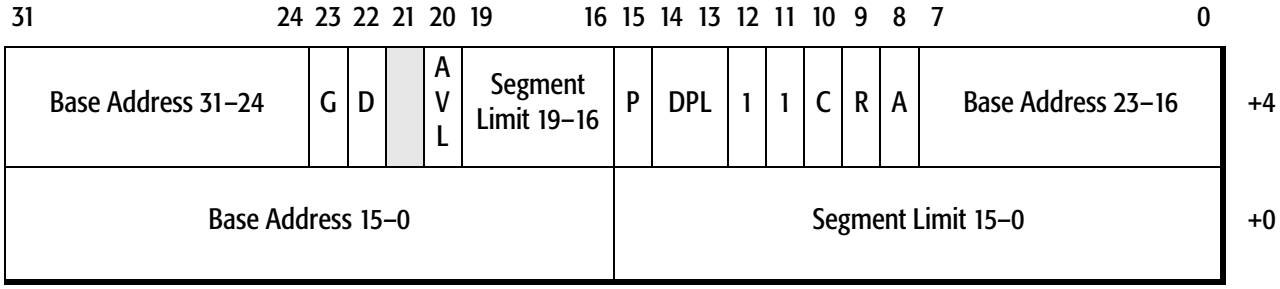


Figure 4-14. Code-Segment Descriptor—Legacy Mode

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 byte +4. 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 122 for more information on conforming and non-conforming code-segments.

Readable (R) Bit. Bit 9 of byte +4. 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 byte +4. 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 100 summarizes the code-segment type-field encodings.

Table 4-3. Code-Segment Descriptor Types

Hex Value	Type Field				Description
	Bit 11 (Code/Data)	Bit 10	Bit 9	Bit 8	
		Conforming (C)	Readable (R)	Accessed (A)	
8	1	0	0	0	Execute-Only
9		0	0	1	Execute-Only – Accessed
A		0	1	0	Execute/Readable
B		0	1	1	Execute/Readable – Accessed
C		1	0	0	Conforming, Execute-Only
D		1	0	1	Conforming, Execute-Only – Accessed
E		1	1	0	Conforming, Execute/Readable
F		1	1	1	Conforming, Execute/Readable – Accessed

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 on page 101 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.

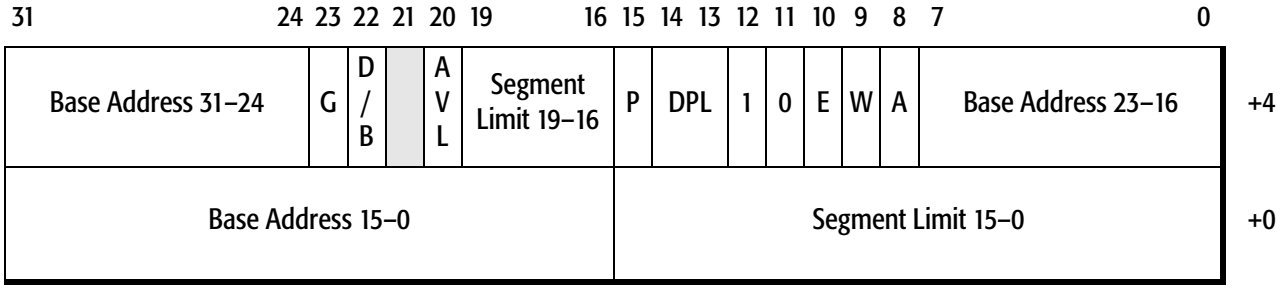


Figure 4-15. Data-Segment Descriptor—Legacy Mode

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 byte +4. 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 byte +4. 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 byte +4. 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 4-4. Data-Segment Descriptor Types

Hex Value	Type Field				Description
	Bit 11 (Code/Data)	Bit 10	Bit 9	Bit 8	
		Expand-Down (E)	Writable (W)	Accessed (A)	
0	0	0	0	0	Read-Only
1		0	0	1	Read-Only – Accessed
2		0	1	0	Read/Write
3		0	1	1	Read/Write – Accessed
4		1	0	0	Expand-down, Read-Only
5		1	0	1	Expand-down, Read-Only – Accessed
6		1	1	0	Expand-down, Read/Write
7		1	1	1	Expand-down, Read/Write – Accessed

Data-Segment Default Operand Size (D/B) Bit. Bit 22 of byte +4. 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 4-5. System-Segment Descriptor Types (S=0)—Legacy Mode

Hex Value	Type Field (Bits 11–8)	Description
0	0000	reserved (Illegal)
1	0001	Available 16-bit TSS
2	0010	LDT
3	0011	Busy 16-bit TSS
4	0100	16-bit Call Gate
5	0101	Task Gate
6	0110	16-bit Interrupt Gate
7	0111	16-bit Trap Gate
8	1000	reserved (Illegal)
9	1001	Available 32-bit TSS
A	1010	reserved (Illegal)
B	1011	Busy 32-bit TSS
C	1100	32-bit Call Gate
D	1101	reserved (Illegal)
E	1110	32-bit Interrupt Gate
F	1111	32-bit Trap Gate

Figure 4-16 on page 104 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 90).
- 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 363 for more information on the TSS.

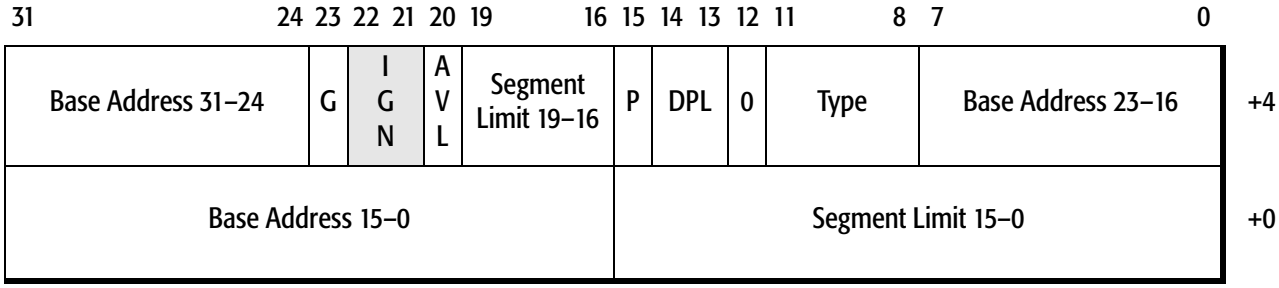


Figure 4-16. LDT and TSS Descriptor–Legacy/Compatibility Modes

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 105) 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 127 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.
- *Interrupt Gates* and *Trap Gates*—These gates (Figure 4-18 on page 105) are located in the IDT and are used to control access to interrupt-service routines. “Legacy Protected-Mode Interrupt Control Transfers” on page 274 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.
- *Task Gates*—These gates (Figure 4-19 on page 105) 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 358 for more information on task gates and their use.

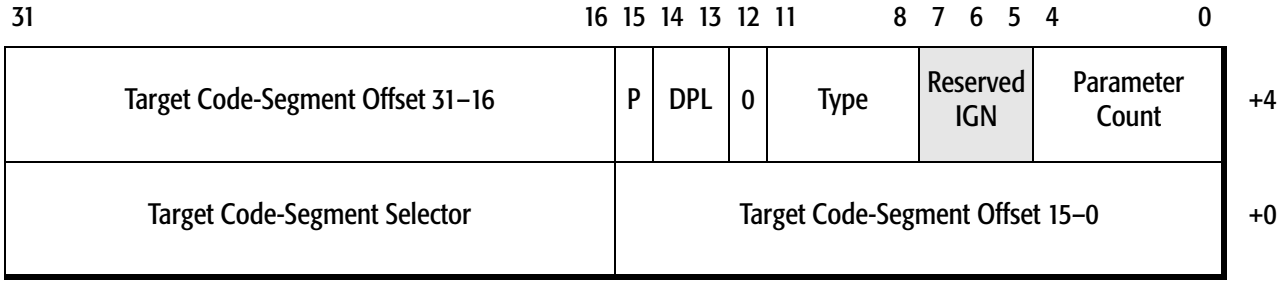


Figure 4-17. Call-Gate Descriptor—Legacy Mode

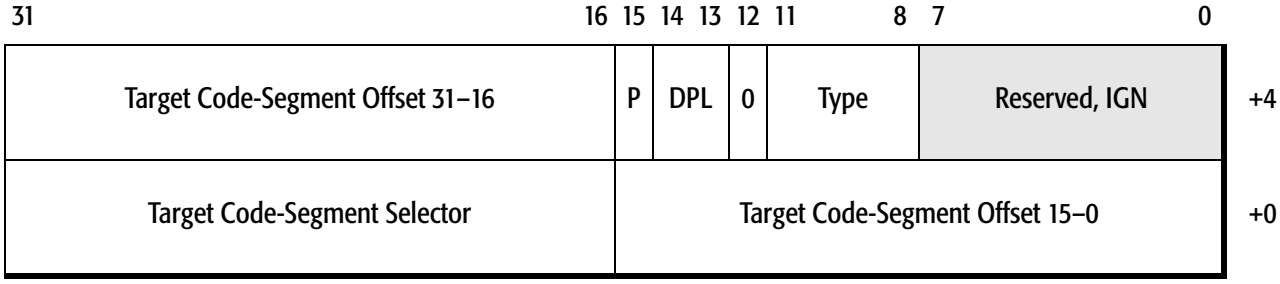


Figure 4-18. Interrupt-Gate and Trap-Gate Descriptors—Legacy Mode

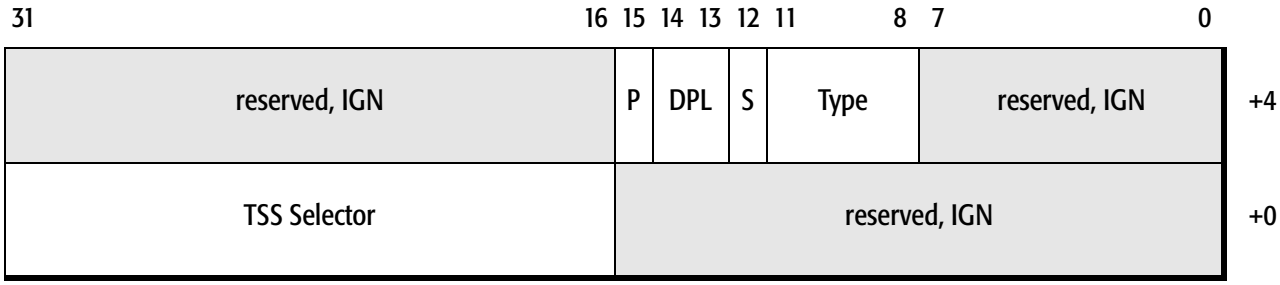


Figure 4-19. Task-Gate Descriptor—Legacy Mode

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 132 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 107). 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 98.

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.

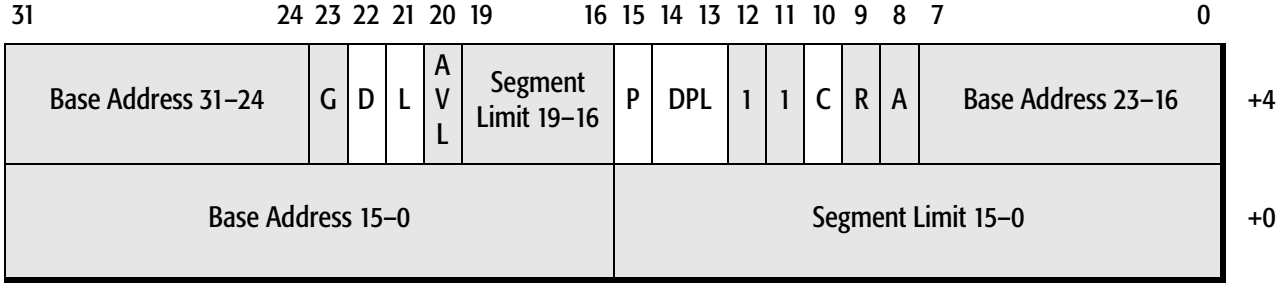


Figure 4-20. Code-Segment Descriptor—Long Mode

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 100)—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.

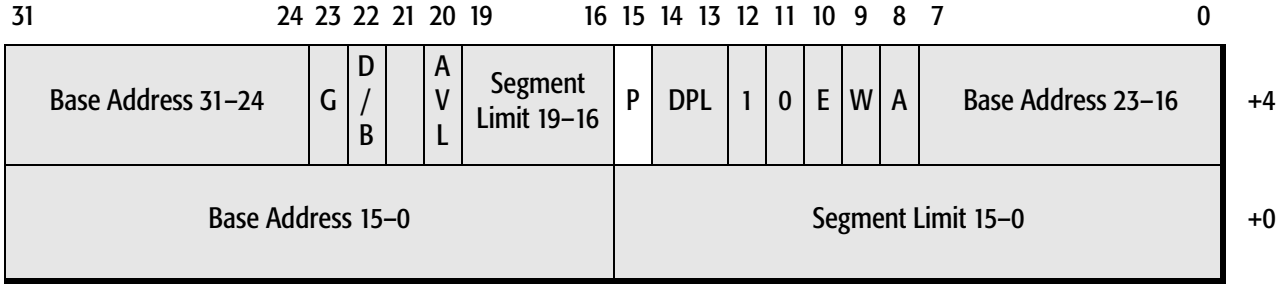


Figure 4-21. Data-Segment Descriptor–Long Mode

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 86 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 on page 110. An attempt to use an illegal descriptor type causes a general-protection exception (#GP).

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

Hex Value	Type Field				Description
	Bit 11	Bit 10	Bit 9	Bit 8	
0	0	0	0	0	reserved (Illegal)
1	0	0	0	1	
2	0	0	1	0	64-Bit LDT[1]
3	0	0	1	1	reserved (Illegal)
4	0	1	0	0	
5	0	1	0	1	
6	0	1	1	0	
7	0	1	1	1	
8	1	0	0	0	
9	1	0	0	1	Available 64-bit TSS
A	1	0	1	0	reserved (Illegal)
B	1	0	1	1	Busy 64-bit TSS
C	1	1	0	0	64-bit Call Gate
D	1	1	0	1	reserved (Illegal)
E	1	1	1	0	64-bit Interrupt Gate
F	1	1	1	1	64-bit Trap Gate

Note:
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 on page 111. 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 104, is used. See "LLDT and LTR Instructions" on page 187 for more information.

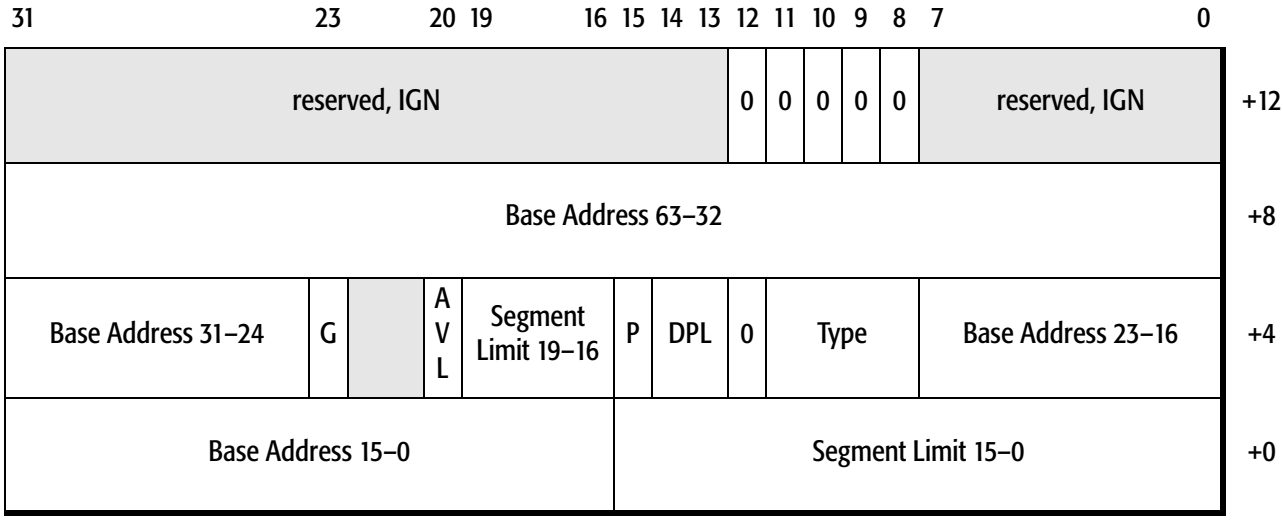


Figure 4-22. System-Segment Descriptor—64-Bit Mode

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 dword +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 110, 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 113. 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 114.

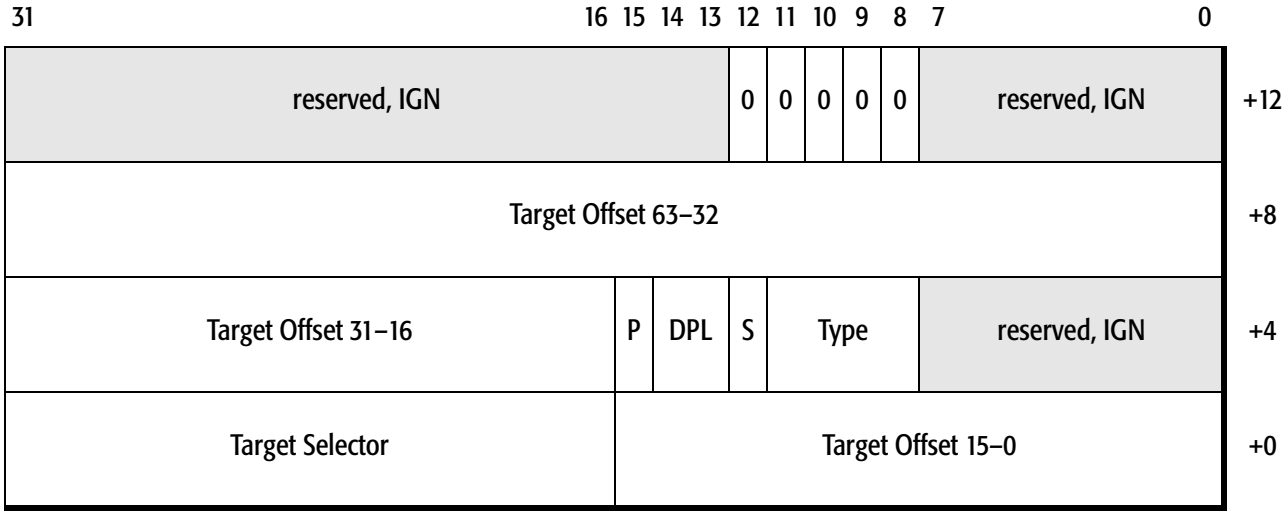


Figure 4-23. Call-Gate Descriptor—Long Mode

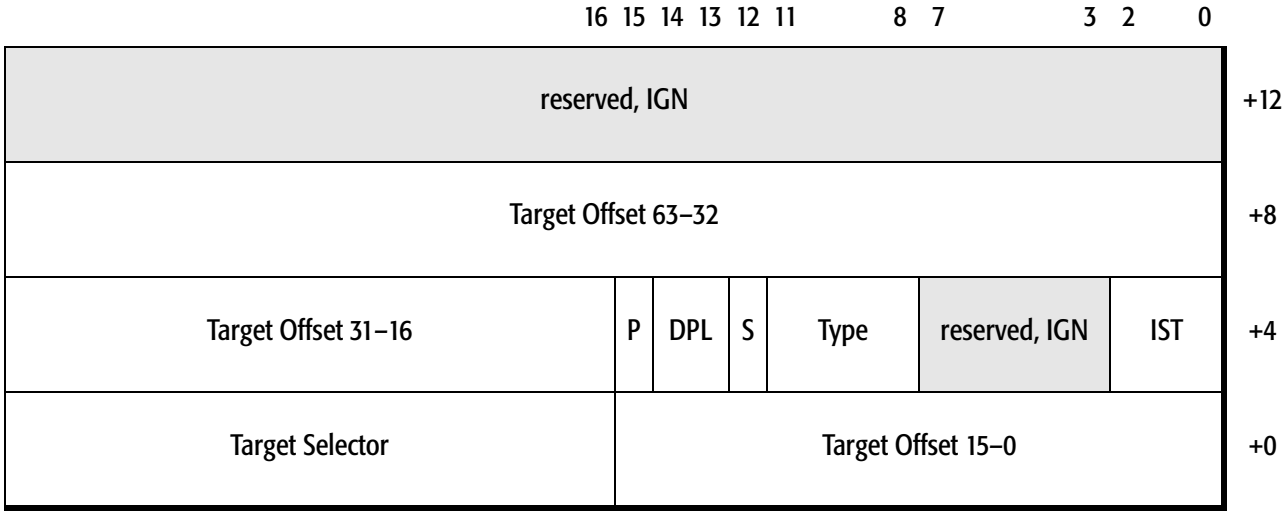


Figure 4-24. Interrupt-Gate and Trap-Gate Descriptors—Long Mode

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 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 290 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 on page 115 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 4-7. Descriptor-Entry Field Changes in Long Mode

Descriptor Field	Descriptor Type	Long Mode	
		Compatibility Mode	64-Bit Mode
Limit	Code	Same as legacy x86	Same as legacy x86
	Data		
	System		
Offset	Gate	Expanded to 64 bits	Expanded to 64 bits
Base	Code	Same as legacy x86	Same as legacy x86
	Data		
	System		
Selector	Gate	Same as legacy x86	
IST[1]	Gate	Interrupt and trap gates only. (New for long mode.)	
S and Type	Code	Same as legacy x86	Same as legacy x86
	Data		
	System	Types 02h, 09h, and 0Bh redefined Types 01h and 03h are illegal	
	Gate	Types 0Ch, 0Eh, and 0Fh redefined Types 04h–07h are illegal	
DPL	Code	Same as legacy x86	Same as legacy x86
	Data		
	System		
	Gate		
Present	Code	Same as legacy x86	Same as legacy x86
	Data		
	System		
	Gate		

Note:

1. *Not available (reserved) in legacy mode.*

Table 4-7. Descriptor-Entry Field Changes in Long Mode (continued)

Descriptor Field	Descriptor Type	Long Mode	
		Compatibility Mode	64-Bit Mode
Default Size	Code	Same as legacy x86	D=0 Indicates 64-bit address, 32-bit data D=1 Reserved
	Data		Same as legacy x86
Long[1]	Code	Specifies compatibility mode	Specifies 64-bit mode
Granularity	Code	Same as legacy x86	Same as legacy x86
	Data		
	System		
Available	Code	Same as legacy x86	Same as legacy x86
	Data		
	System		

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

4.9 Segment-Protection Overview

The AMD64 architecture fully supports 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 172. 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 BIOS, 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 on page 118 shows the relationship of the four privilege levels to each other.

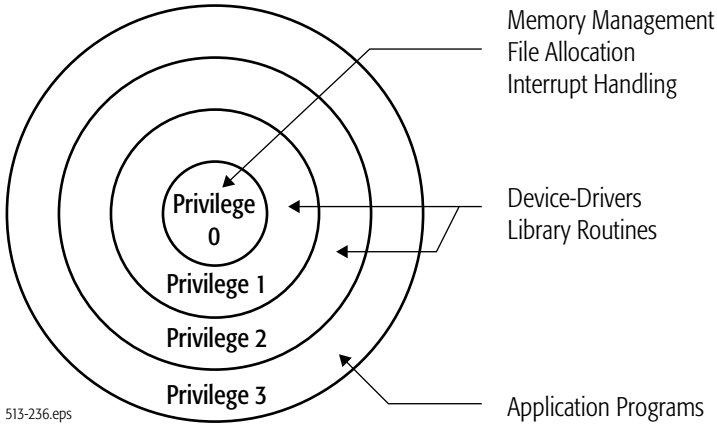


Figure 4-25. Privilege-Level Relationships

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 120 shows two examples of data-access privilege checks.

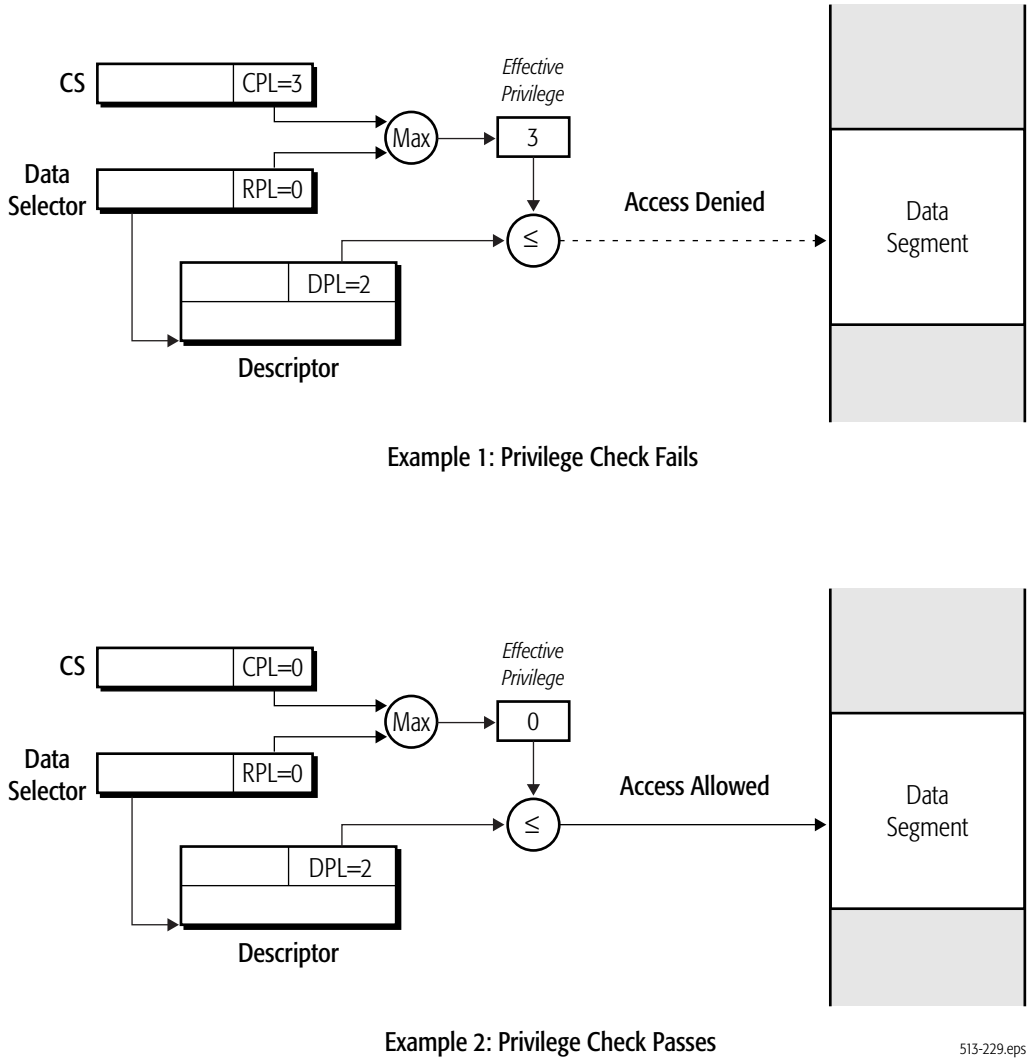


Figure 4-26. Data-Access Privilege-Check Examples

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 on page 122 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.

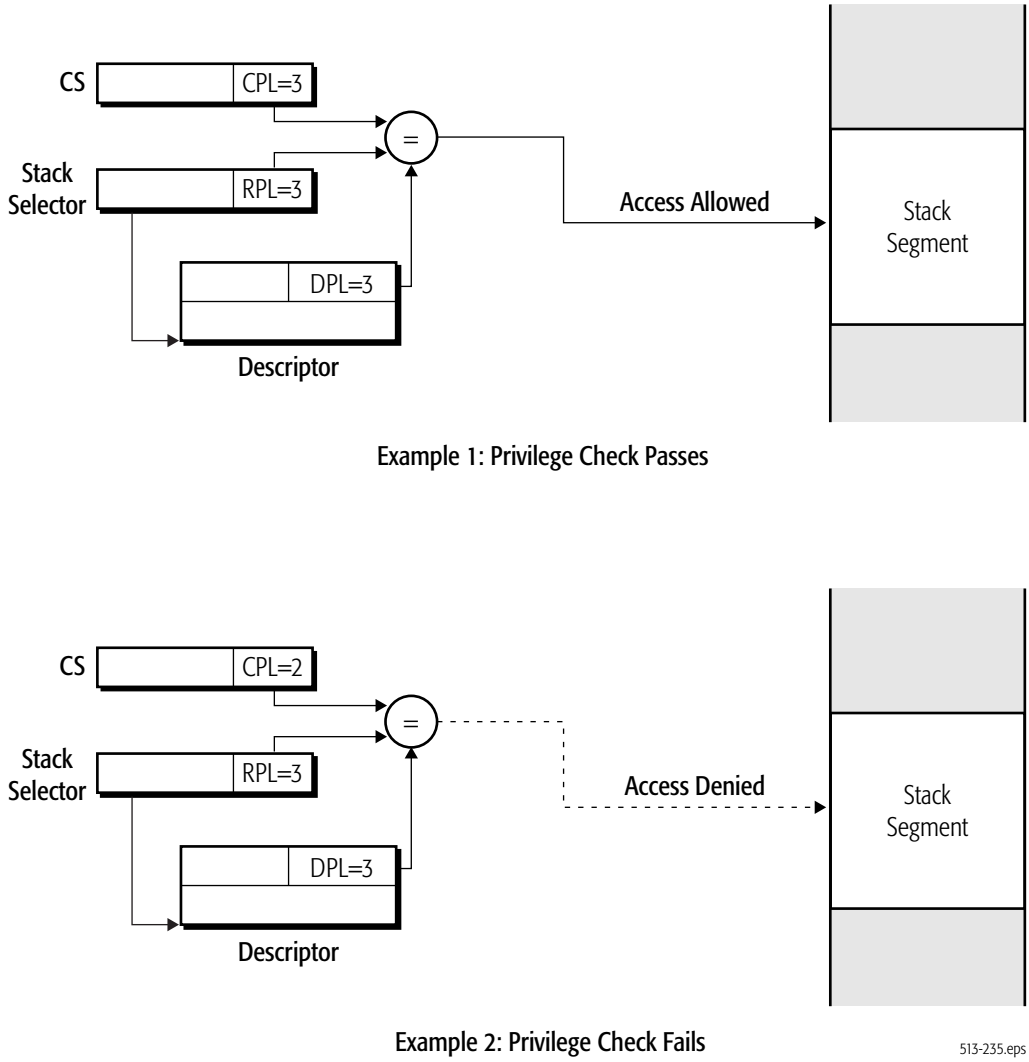


Figure 4-27. Stack-Access Privilege-Check Examples

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 180 and "SYSENTER and SYSEXIT (Legacy Mode Only)" on page 182 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 127.
- Return control transfers using the RET instruction. These are discussed in "Return Control Transfers" on page 135.
- Interrupts and exceptions, including the INT*n* 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 99 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 132 for more information on automatic stack switching during privilege-changing control transfers).

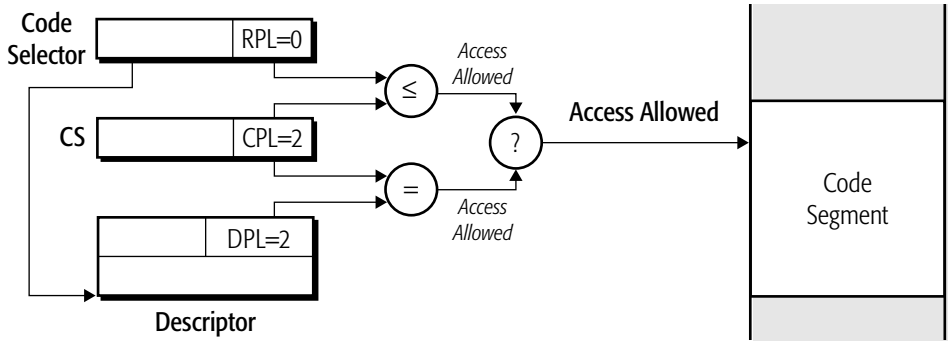
Nonconforming Code Segments. Software can perform a direct control transfer to a nonconforming code segment only if the target code-segment descriptor DPL and the CPL are equal and the RPL is less than or equal to the CPL. Software must use a call gate to transfer control to a more-privileged, nonconforming code segment (see "Control Transfers Through Call Gates" on page 127 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:

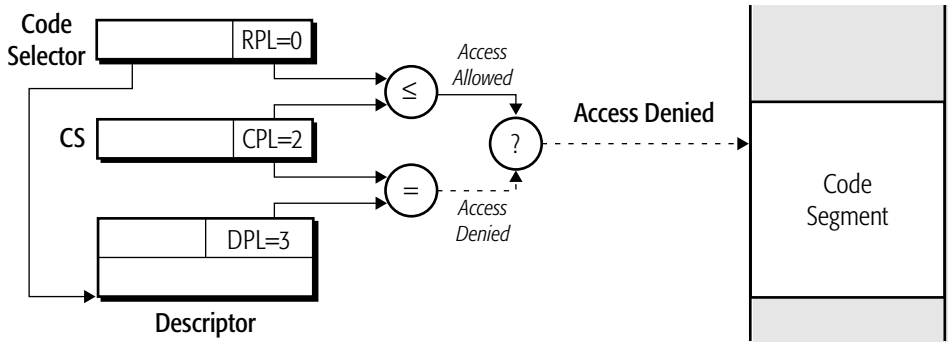
1. *DPL = CPL Check*—The processor compares the target code-segment descriptor DPL with the currently executing program CPL. If they are equal, the processor performs the next check. If they are not equal, a general-protection exception (#GP) occurs.
2. *RPL ≤ CPL Check*—The processor compares the target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

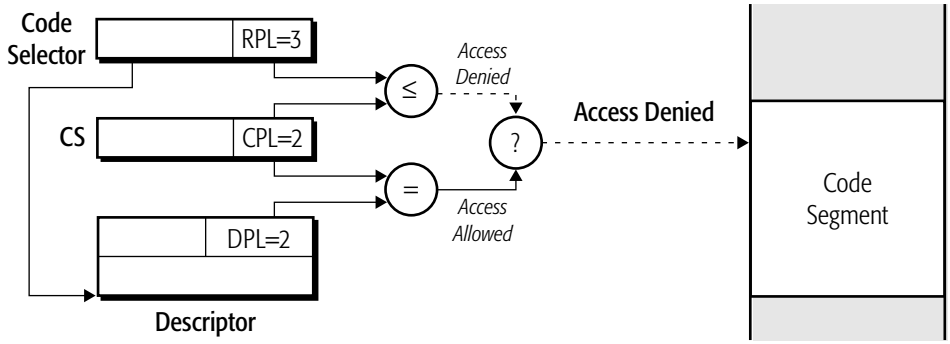
Figure 4-28 on page 125 shows three examples of privilege checks performed as a result of a far control transfer to a nonconforming code-segment. In Example 1, access is allowed because CPL = DPL and RPL ≤ CPL. In Example 2, access is denied because CPL ≠ DPL. In Example 3, access is denied because RPL > CPL.



Example 1: Privilege Check Passes



Example 2: Privilege Check Fails



Example 3: Privilege Check Fails

513-230.eps

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 on page 127 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.

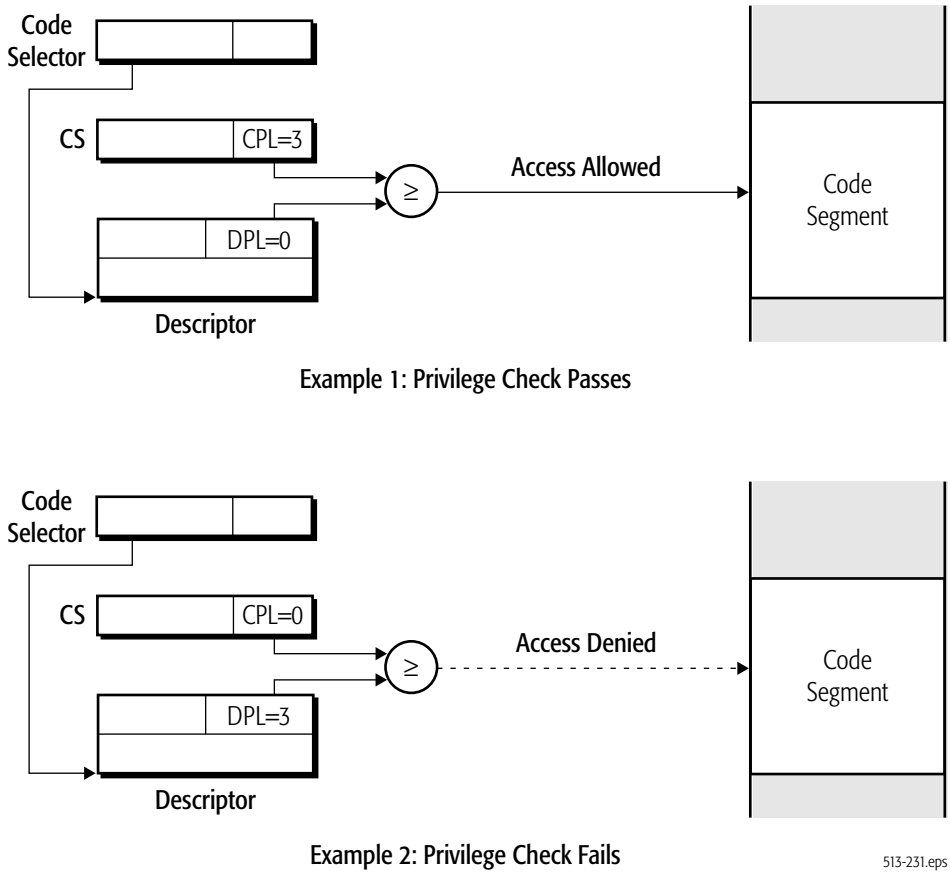


Figure 4-29. Conforming Code-Segment Privilege-Check Examples

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 104 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).

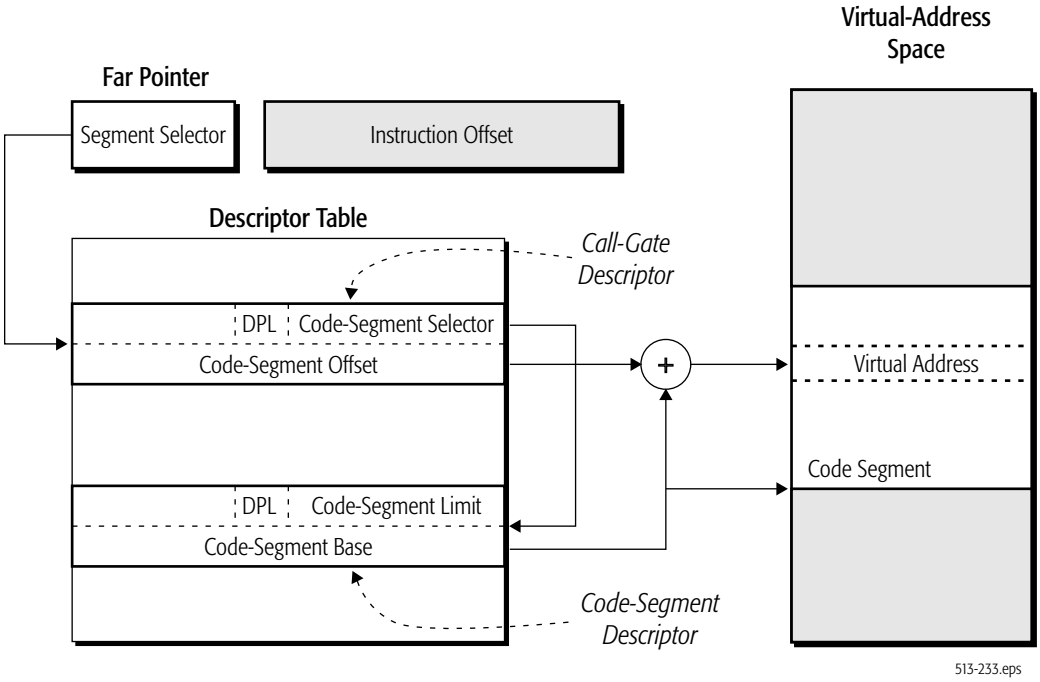


Figure 4-30. Legacy-Mode Call-Gate Transfer Mechanism

Figure 4-31 on page 129 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.

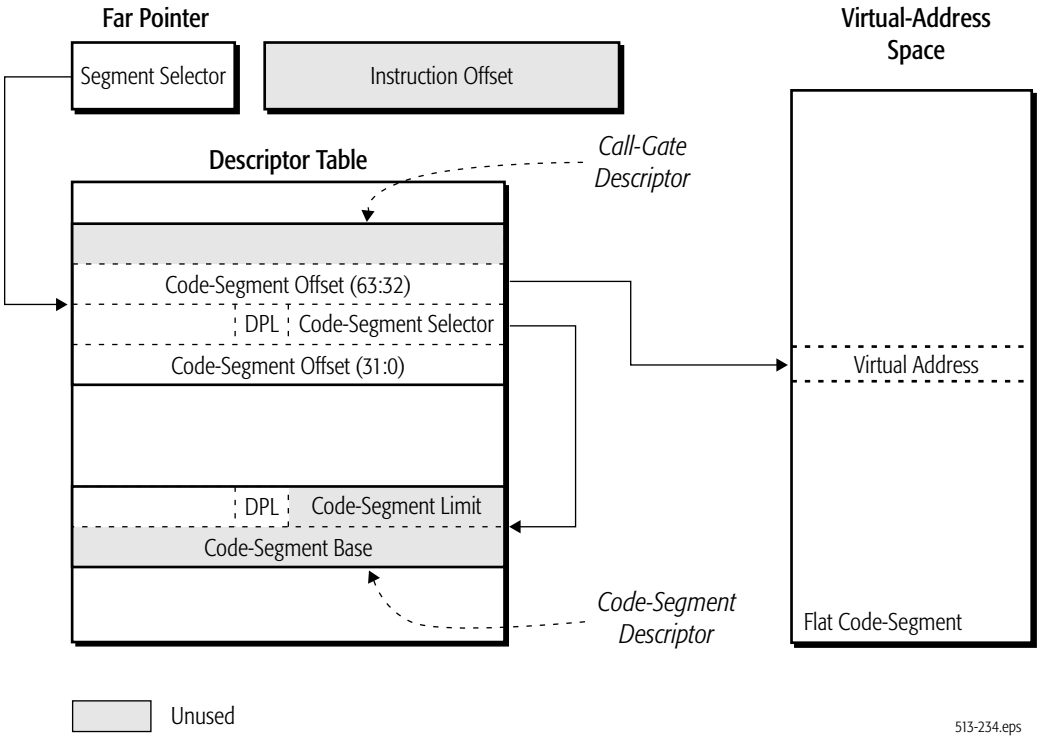


Figure 4-31. Long-Mode Call-Gate Access Mechanism

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 (DPL_G). The CPL must be numerically *less than or equal to* DPL_G for this check to pass. In other words, the following expression must be true: $CPL \leq DPL_G$.

2. The processor compares the RPL in the call-gate selector with DPL_G. The RPL must be numerically *less than or equal to* DPL_G for this check to pass. In other words, the following expression must be true: $RPL \leq DPL_G$.
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 *conforming* code segment—is used to transfer control through a call gate, the CPL must be numerically *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: $CPL \geq DPL_S$.
   - When a JMP instruction is used to transfer control through a call gate to a *nonconforming* code segment, the CPL must be numerically *equal to* DPL_S for this check to pass. (JMP instructions cannot change CPL.) In other words, the following expression must be true: $CPL = DPL_S$.

Figure 4-32 on page 131 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.

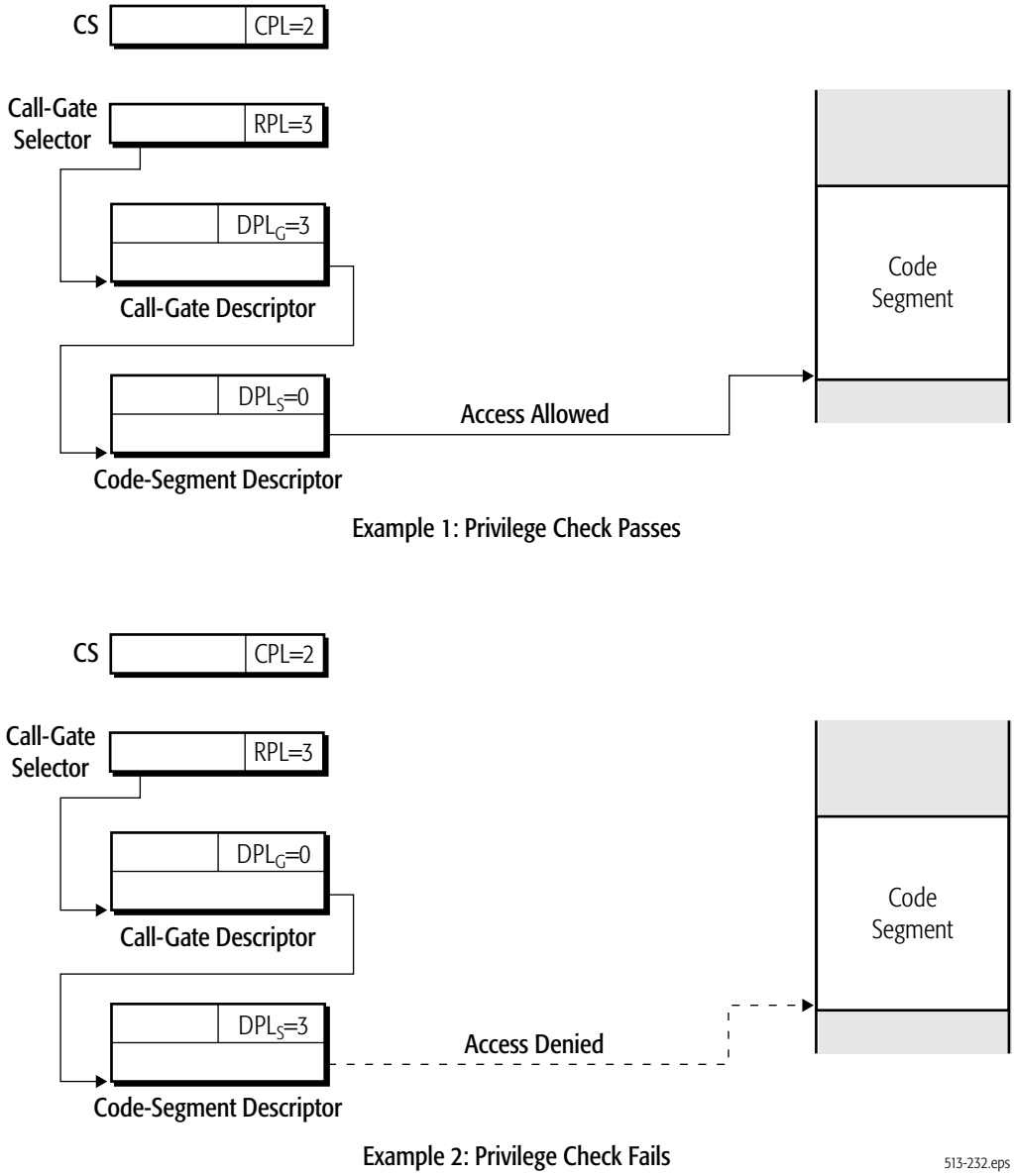


Figure 4-32. Privilege-Check Examples for Call Gates

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 ($DPL_S = 3$) 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 358 for more information on the legacy-mode and long-mode TSS formats.

Figure 4-33 on page 133 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.

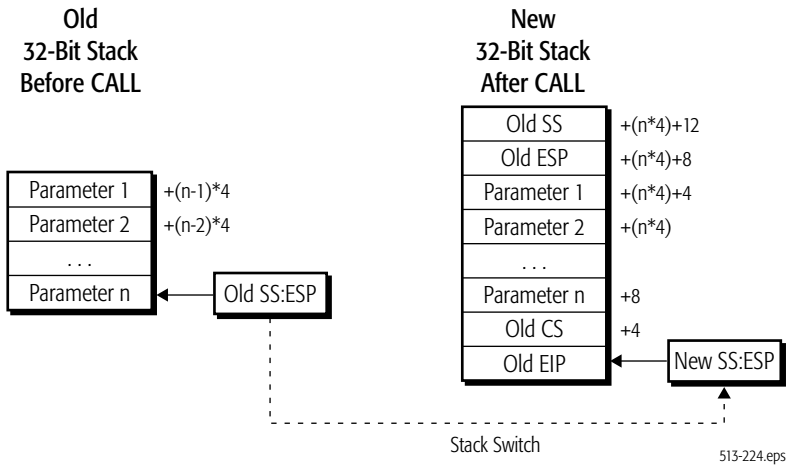


Figure 4-33. Legacy-Mode 32-Bit Stack Switch, with Parameters

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.

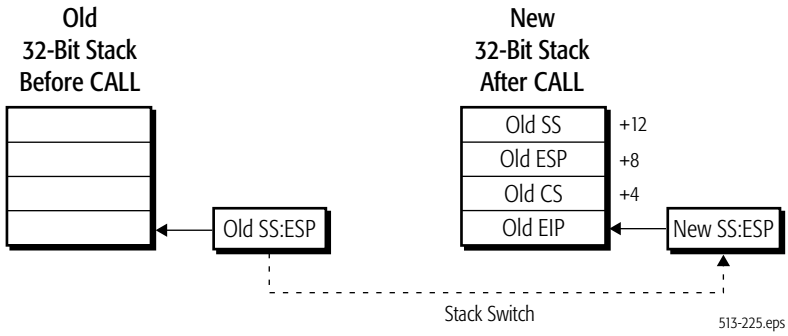


Figure 4-34. 32-Bit Stack Switch, No Parameters—Legacy Mode

Figure 4-35 on page 135 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 137 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.

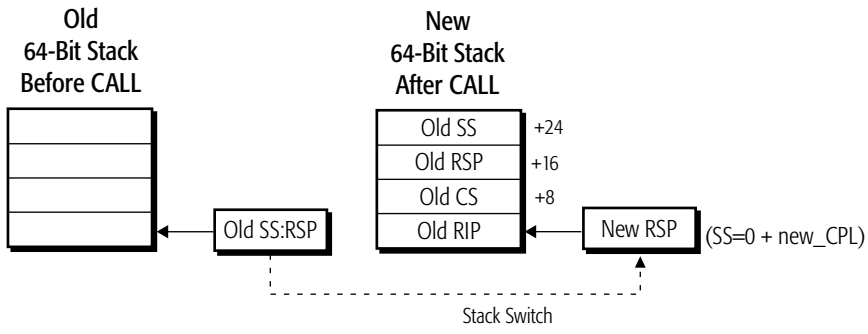


Figure 4-35. Stack Switch—Long Mode

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 137.

  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 95 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:

```
maximum segment-offset = (limit × 1000h) + 0FFFh
```

For example, if the segment-limit field is 0100h, then the maximum allowable segment-offset is

(0100h × 1000h) + 0FFFh = 10_1FFFh.

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:

  ```
  minimum segment-offset = (limit × 1000) + 0FFFh + 1
  ```

  For example, if the segment-limit field is 0100h, then the minimum allowable segment-offset is

  (0100h × 1000h) + 0FFFh + 1 = 10_1000h.

For expand-down segments, the maximum segment size is specified when the segment-limit value is 0.

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 95 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.

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 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 145 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.

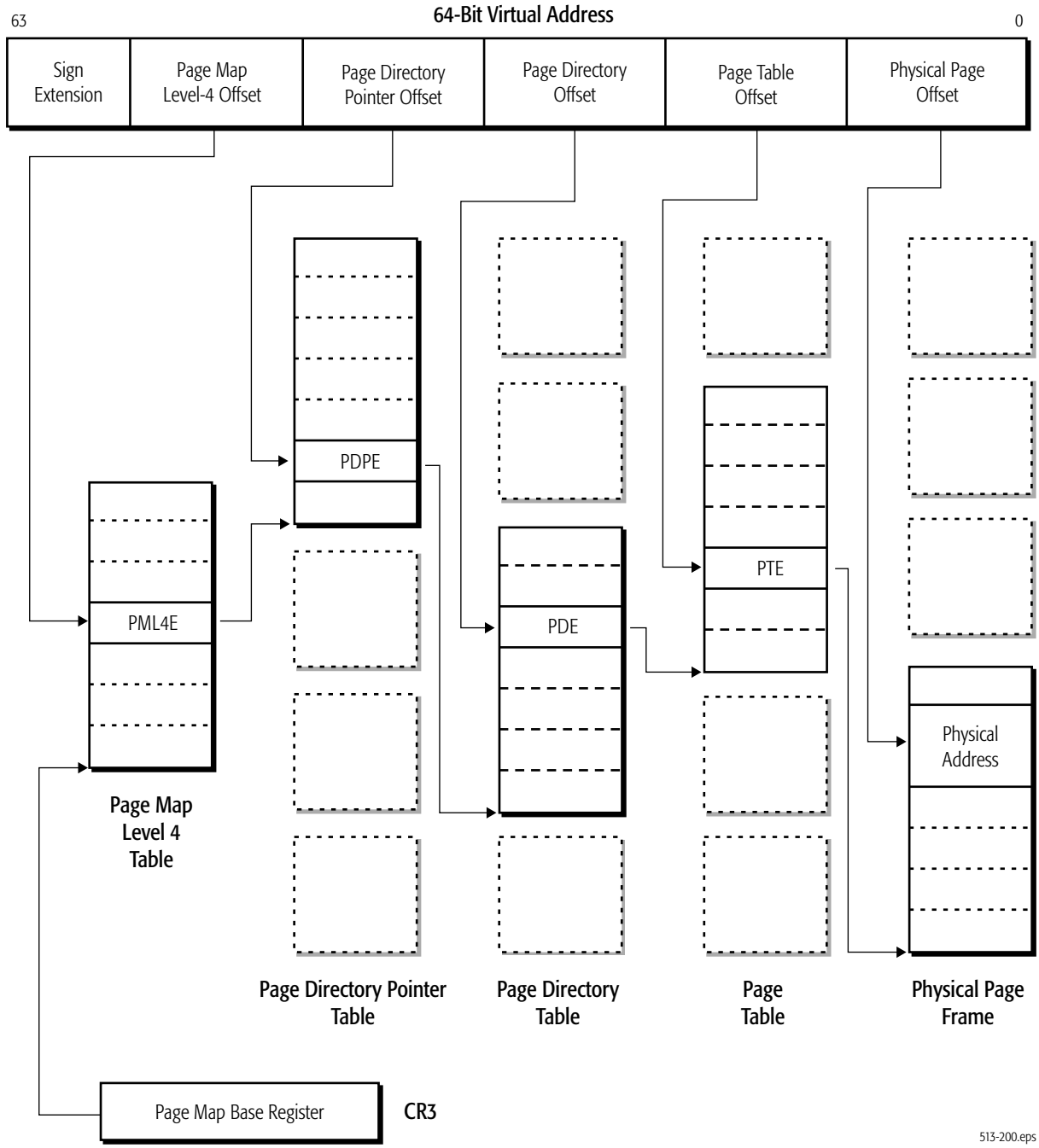


Figure 5-1. Virtual to Physical Address Translation—Long Mode

Legacy page translation offers a variety of alternatives in translating virtual addresses to physical addresses. Three

physical-page sizes of 4 Kbytes, 2 Mbytes, and 4 Mbytes 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 5-1. Supported Paging Alternatives (CR0.PG=1)

Mode		Physical-Address Extensions (CR4.PAE)	Page-Size Extensions (CR4.PSE)	Page-Directory Page Size	Resulting Physical-Page Size	Maximum Virtual Address	Maximum Physical Address
Long Mode	64-Bit Mode	Must be enabled	–	PDE.PS=0	4 Kbyte	64-bit	52-bit
	Compatibility Mode			PDE.PS=1	2 Mbyte		
Legacy Mode		Enabled	–	PDE.PS=0	4 Kbyte	32-bit	52-bit
				PDE.PS=1	2 Mbyte		52-bit
		Disabled	Disabled	–	4 Kbyte		32-bit
			Enabled	PDE.PS=0	4 Kbyte		32-bit
				PDE.PS=1	4 Mbyte		40-bit

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 422 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 422.

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 4Mbyte physical pages in the translation process. The 4Mbyte 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 168 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 152 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 is one of the data structures used in page translation (see Figure 5-1 on page 145). 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. 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 146. 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.

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 150.

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 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).

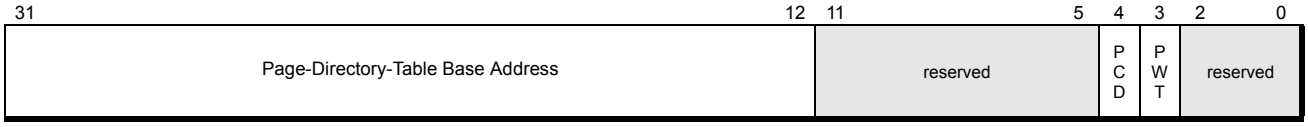


Figure 5-2. Control Register 3 (CR3)—Non-PAE Paging Legacy-Mode

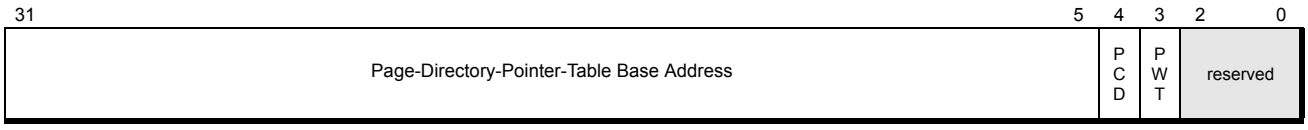


Figure 5-3. Control Register 3 (CR3)—PAE Paging Legacy-Mode

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 are 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 on page 151:

- 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.

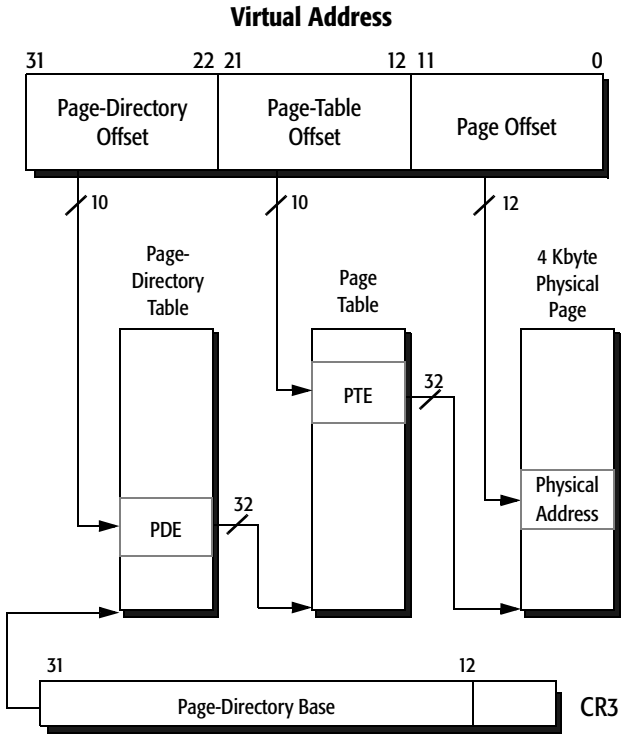


Figure 5-4. 4-Kbyte Non-PAE Page Translation—Legacy Mode

Figure 5-5 shows the format of the PDE (page-directory entry), and Figure 5-6 on page 152 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 166.

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.

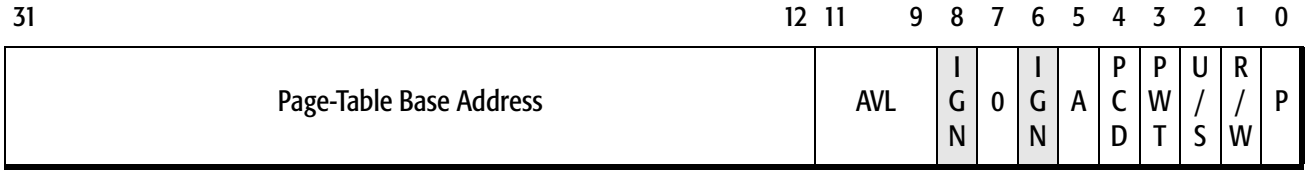


Figure 5-5. 4-Kbyte PDE—Non-PAE Paging Legacy-Mode

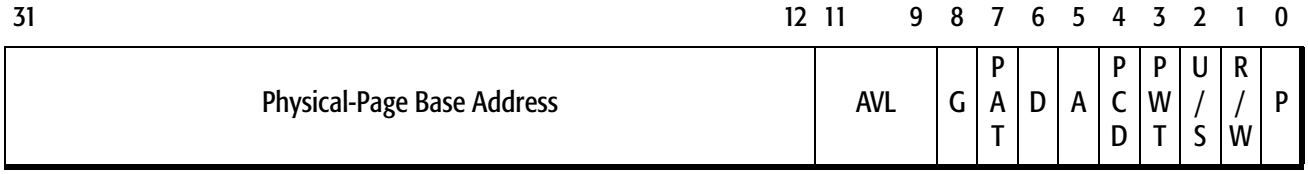


Figure 5-6. 4-Kbyte PTE—Non-PAE Paging Legacy-Mode

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 153:

- Bits 31–22 index into the 1024-entry page-directory table.
- Bits 21–0 provide the byte offset into the physical page.

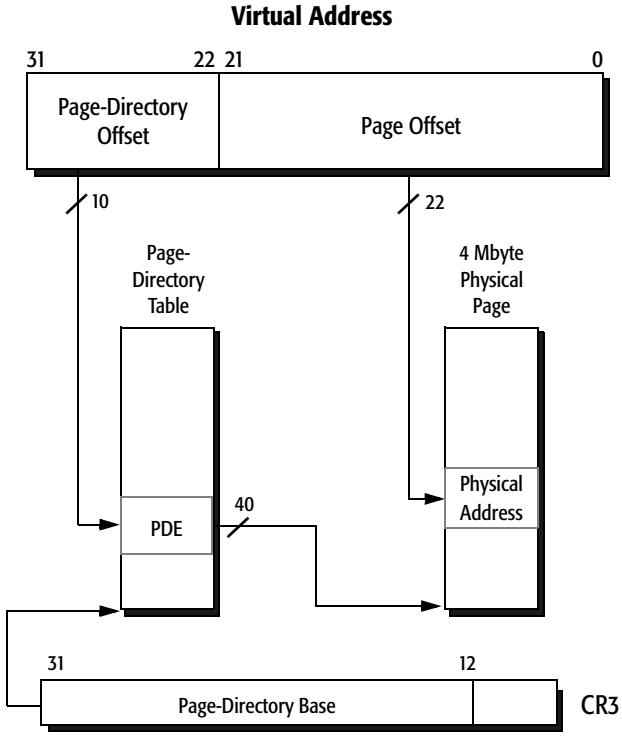


Figure 5-7. 4-Mbyte Page Translation–Non-PAE Paging Legacy-Mode

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].

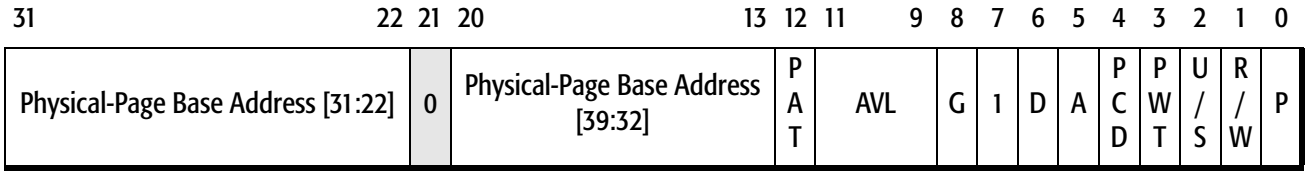


Figure 5-8. 4-Mbyte PDE–Non-PAE Paging Legacy-Mode

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 on page 155:

- 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.

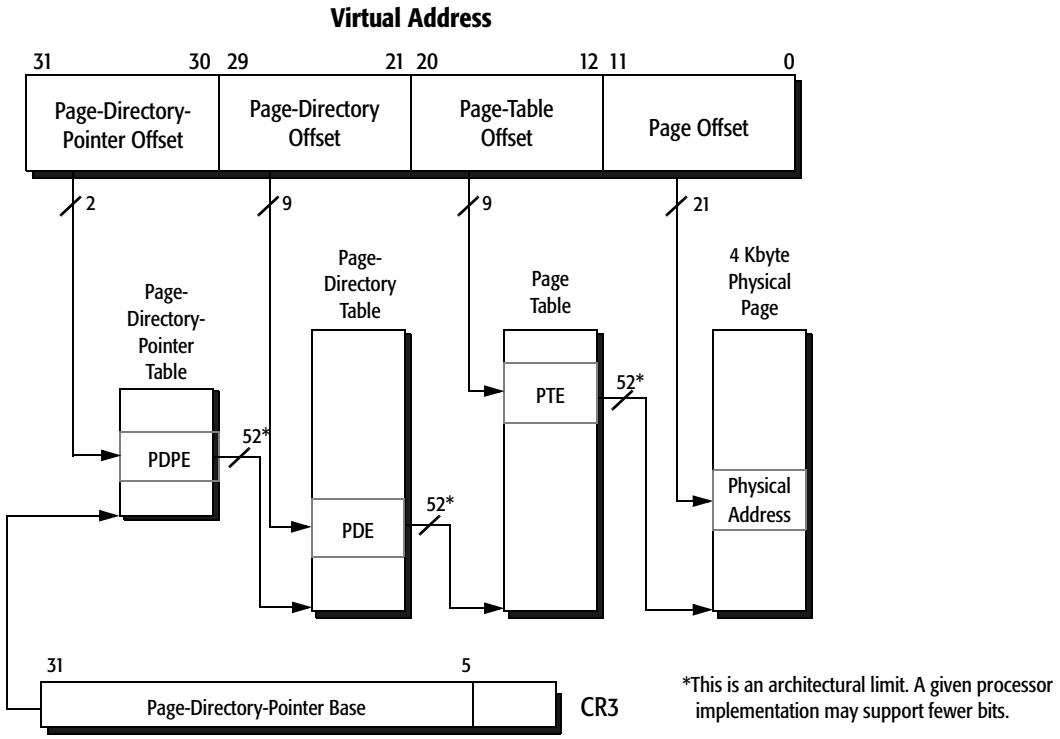


Figure 5-9. 4-Kbyte PAE Page Translation—Legacy Mode

Figures 5-10 through 5-12 on page 156 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 166.

Figure 5-11 shows the PDE.PS bit cleared to 0 (bit 7), specifying a 4-Kbyte physical-page translation.

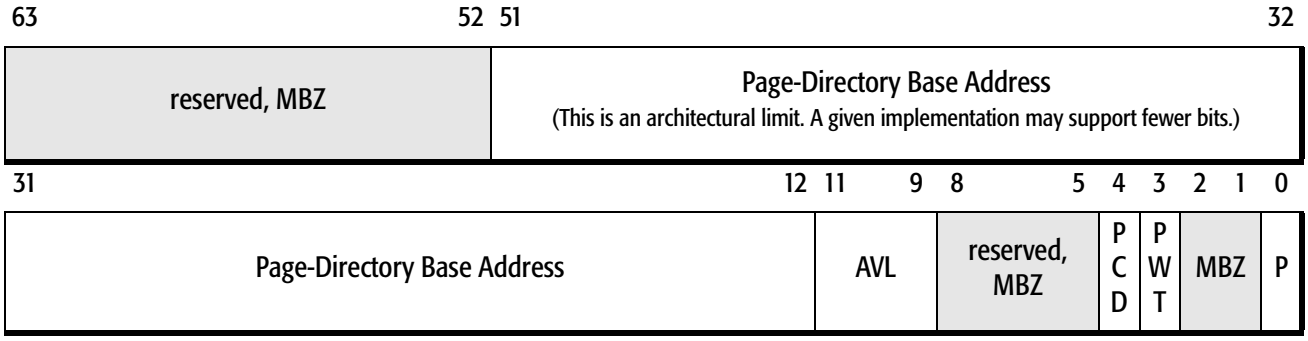


Figure 5-10. 4-Kbyte PDPE—PAE Paging Legacy-Mode

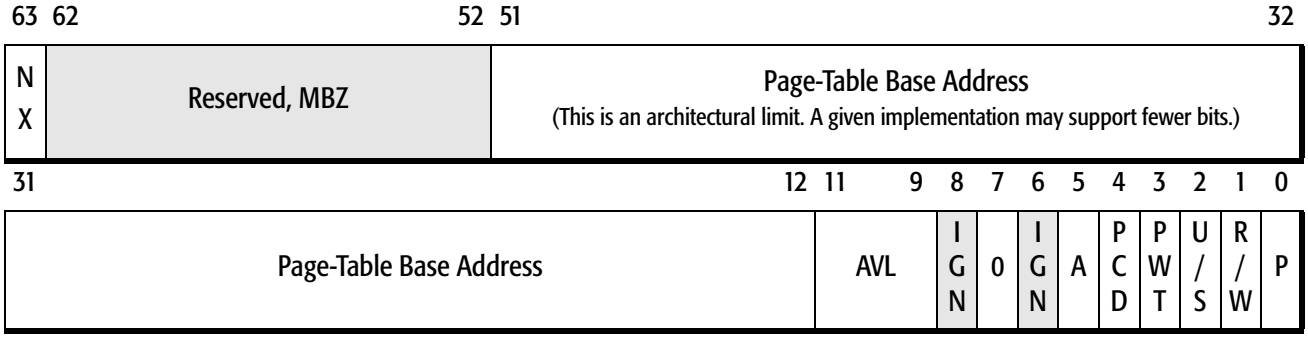


Figure 5-11. 4-Kbyte PDE—PAE Paging Legacy-Mode

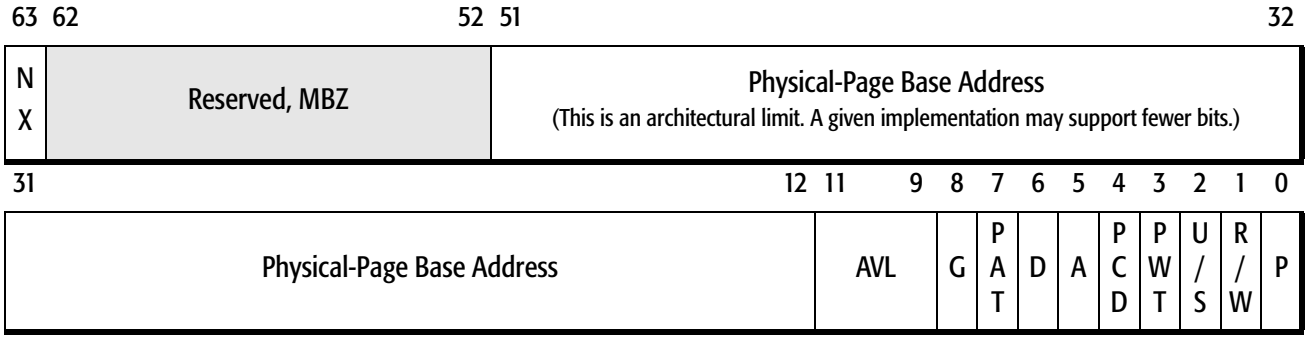


Figure 5-12. 4-Kbyte PTE—PAE Paging Legacy-Mode

2-Mbyte 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:

- 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.

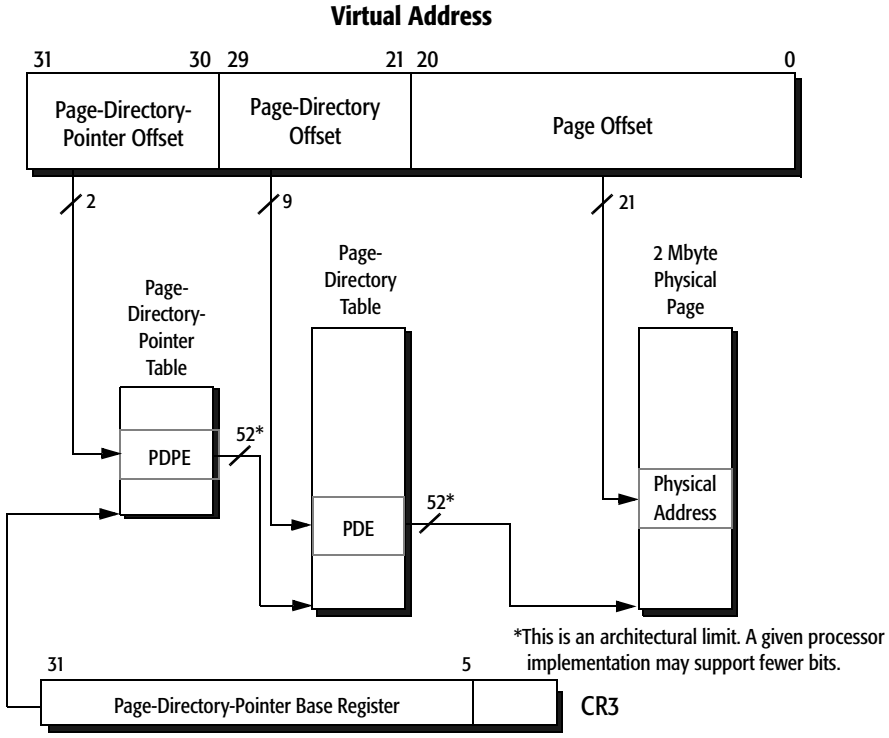


Figure 5-13. 2-Mbyte PAE Page Translation—Legacy Mode

Figure 5-14 on page 158 shows the format of the PDPE (page-directory-pointer entry) and Figure 5-15 on page 158 shows the format of the PDE (page-directory entry). PTEs are not used in 2-Mbyte page translations.

Figure 5-15 on page 158 shows the PDE.PS bit set to 1 (bit 7), specifying a 2-Mbyte physical-page translation.

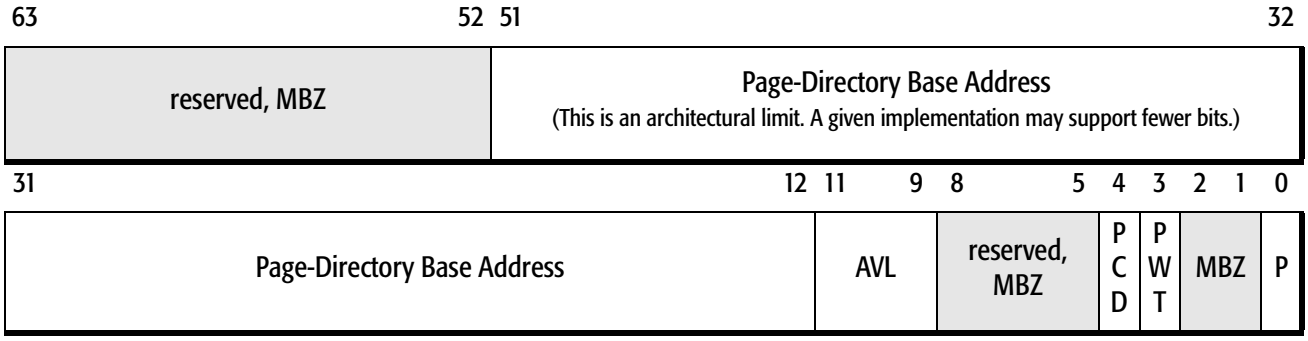


Figure 5-14. 2-Mbyte PDPE—PAE Paging Legacy-Mode

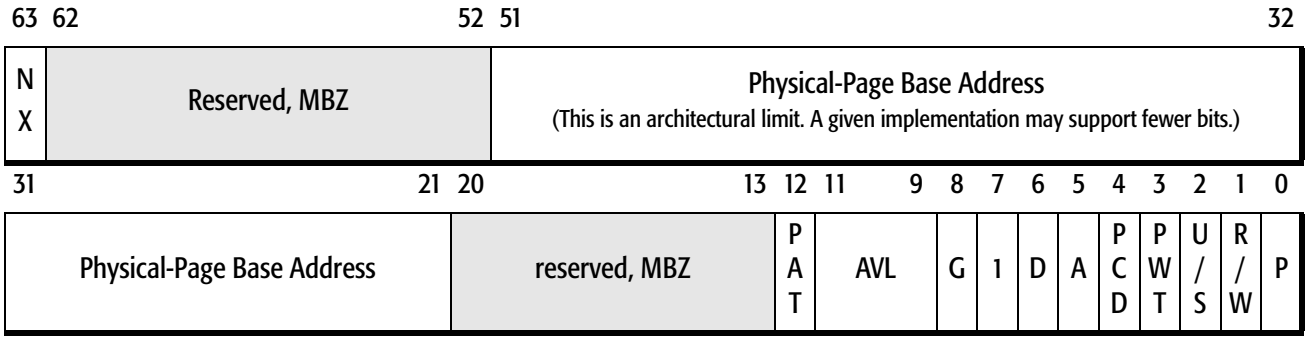


Figure 5-15. 2-Mbyte PDE—PAE Paging Legacy-Mode

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.

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 5-16 shows the long-mode CR3 format.

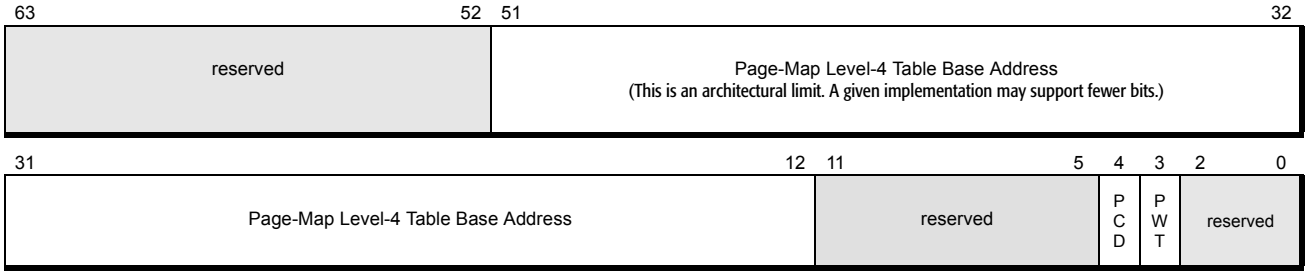


Figure 5-16. Control Register 3 (CR3)—Long Mode

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.

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 48-bit virtual address into six fields. Each field is used as an index into the four-level page-translation hierarchy. The virtual-address fields are described as follows, and are shown in Figure 5-17 on page 161:

- 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.

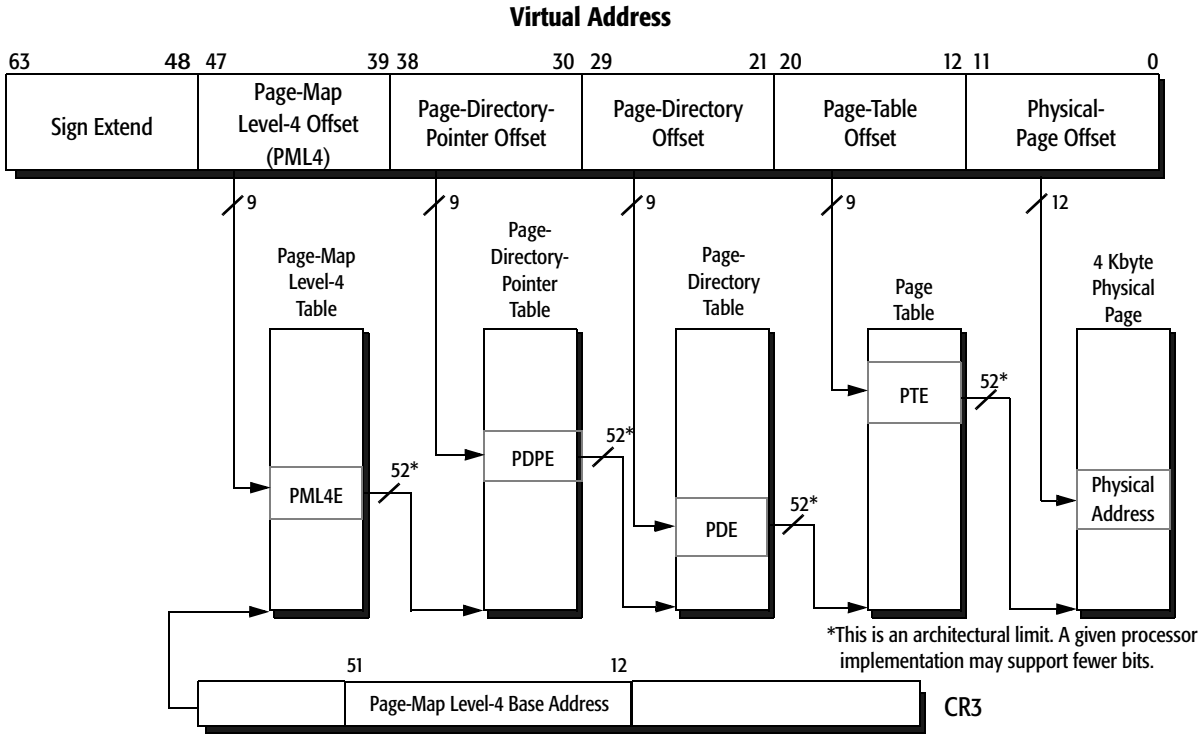


Figure 5-17. 4-Kbyte Page Translation—Long Mode

Figures 5-18 through 5-20 on page 162 and Figure 5-21 on page 163 show the long-mode 4-Kbyte translation-table formats:

- Figure 5-18 shows the PML4E (page-map level-4 entry) format.
- Figure 5-19 shows the PDPE (page-directory-pointer entry) format.
- Figure 5-20 shows the PDE (page-directory entry) format.
- Figure 5-21 shows the PTE (page-table entry) format.

The fields within these table entries are described in "Page-Translation-Table Entry Fields" on page 166.

Figure 5-20 shows the PDE.PS bit (bit 7) cleared to 0, indicating a 4-Kbyte physical-page translation.

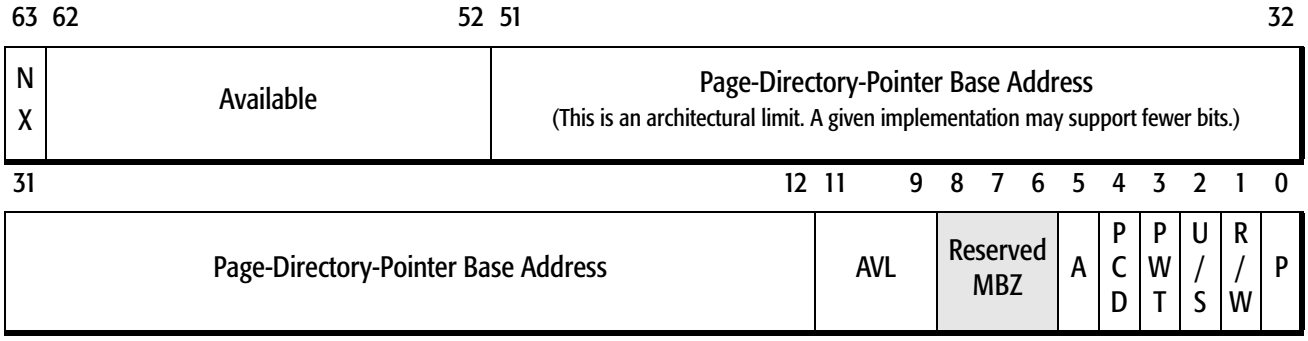


Figure 5-18. 4-Kbyte PML4E—Long Mode

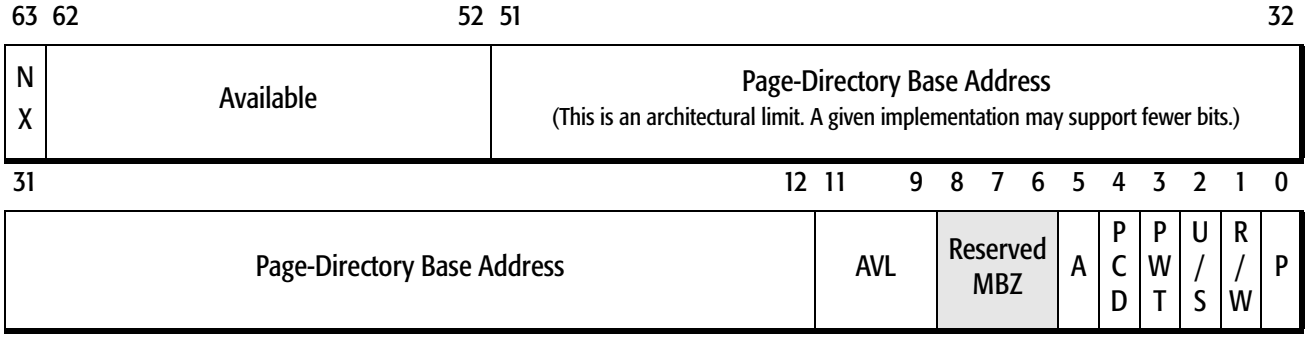


Figure 5-19. 4-Kbyte PDPE—Long Mode

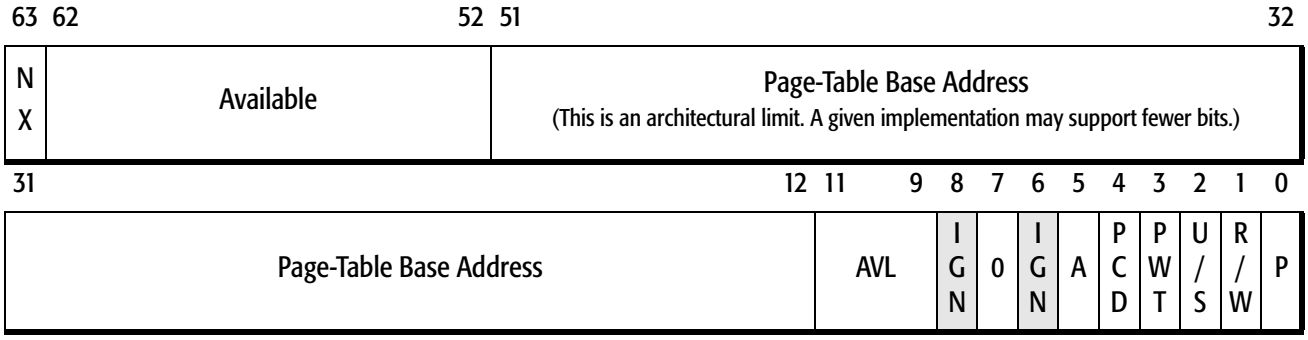


Figure 5-20. 4-Kbyte PDE—Long Mode

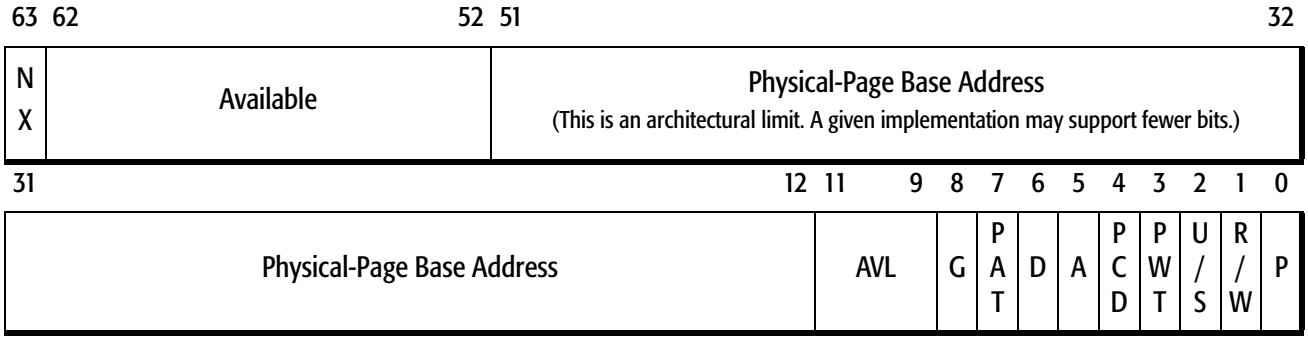


Figure 5-21. 4-Kbyte PTE—Long Mode

5.3.4 2-Mbyte Page Translation

In long mode, 2-Mbyte physical-page translation is performed by dividing the 48-bit virtual address into four fields. Each field is used as an index into a three-level page-translation hierarchy. The virtual-address fields are described as follows, and are shown in Figure 5-22 on page 164:

- Bits 47–39 index into the 512-entry page-map level-4 table.

  Bits 63–48 are a sign extension of bit 47 as required for canonical address forms.
- 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.

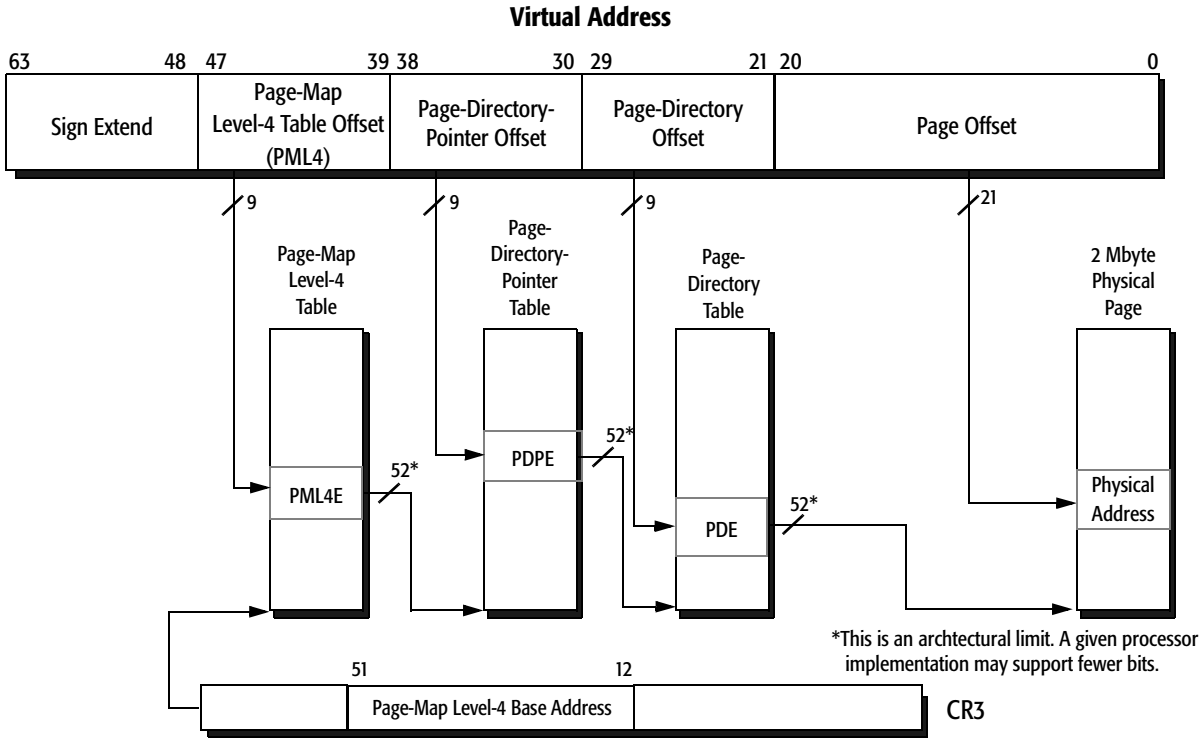


Figure 5-22. 2-Mbyte Page Translation—Long Mode

Figures 5-23 through 5-25 on page 165 show the long-mode 2-Mbyte translation-table formats (the formats are identical to those used for 4-Kbyte page translations and are repeated here for clarity):

- Figure 5-23 shows the PML4E (page-map level-4 entry) format.
- Figure 5-24 shows the PDPE (page-directory-pointer entry) format.
- Figure 5-25 shows the PDE (page-directory entry) format.

The fields within these table entries are described in "Page-Translation-Table Entry Fields" on page 166. 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.

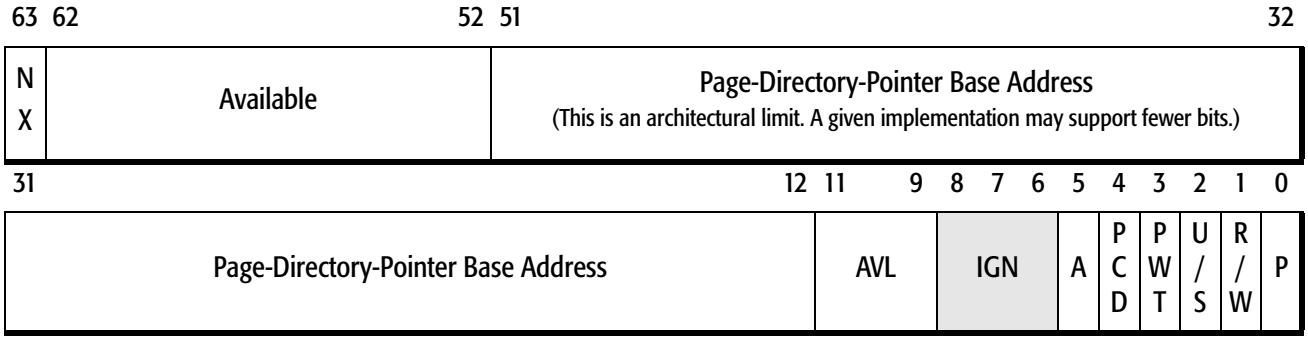


Figure 5-23. 2-Mbyte PML4E—Long Mode

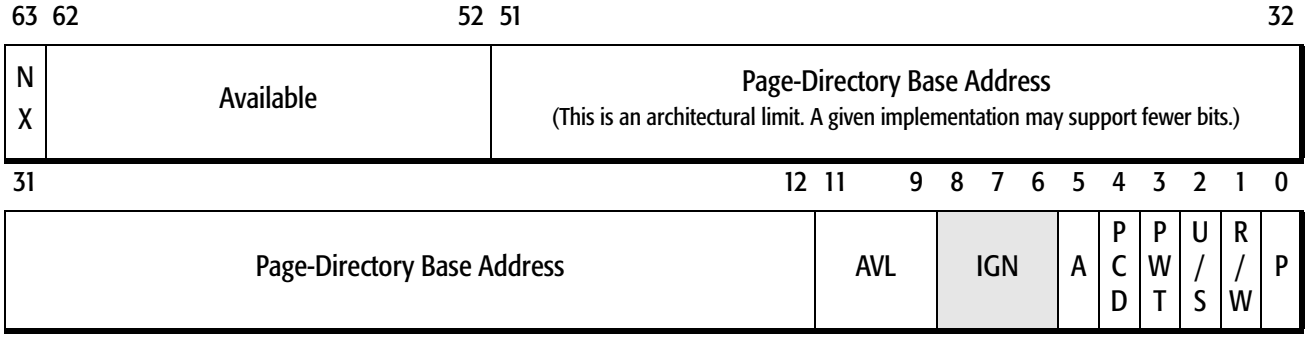


Figure 5-24. 2-Mbyte PDPE—Long Mode

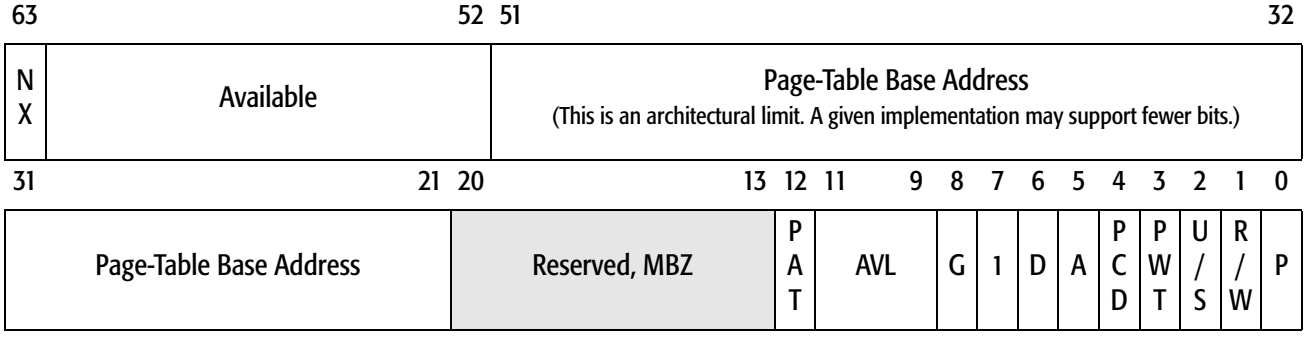


Figure 5-25. 2-Mbyte PDE—Long Mode

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, or 4 Mbytes, 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.

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.

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 172 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 172 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 206 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 206 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 page-translation table or 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 only present in page-directory entries. It specifies the lowest level of the page-translation hierarchy and the physical-page size. When the PS bit is cleared to 0, the lowest level of the page-translation hierarchy is the page-table entry (PTE), and the physical-page size is 4 Kbytes. When the PS bit is set to 1, the lowest level of the page-

translation hierarchy is the page-directory entry (PDE)s. The physical-page size is determined as follows:

- If CR4.PAE=0 and CR4.PSE=1, the physical-page size is 4 Mbytes.
- If CR4.PAE=1, the physical-page size is 2 Mbytes.

See Table 5-1 on page 146 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 171 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), 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 228). 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 228 for a description of the PAT mechanism and how it is used.

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 67). 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 173 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.

5.5 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 171. Only privileged software running at CPL=0 can manage the TLBs.

5.5.1 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 page-global extensions (CR4.PGE=0).

5.5.2 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. 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 are immediately 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. Two mechanisms are provided to explicitly invalidate the TLB:

- The *invalidate TLB entry* instruction (INVLPG) can be used to invalidate specific entries within the TLB. This instruction invalidates a page, regardless of whether it is marked as global or not.
- 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.

Implicit Invalidations. The following operations cause the entire TLB to be invalidated, including global pages:

- Modifying the CR0.PG bit (page enable) or the CR0.PE bit (protected-mode 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).

5.6 Page-Protection Checks

Two forms of page-level memory protection are provided by the legacy architecture. The first form of protection prevents non-privileged (user) code and data from accessing privileged (supervisor) code and data. The second form of protection prevents writes into read-only address spaces. The AMD64 architecture introduces a third form of protection that prevents software from attempting to execute data pages as instructions. All of these forms of protection are available at all levels of the page-translation hierarchy.

The processor checks a page for execute permission only when the page translation is loaded into the instruction TLB as a

result of a page-table walk. The remaining 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 bits involved in these checks are:

- *User/Supervisor (U/S)*—The U/S bit is introduced in "User/Supervisor (U/S) Bit" on page 167.
- *Read/Write (R/W)*—The R/W bit is introduced in "Read/Write (R/W) Bit" on page 167.
- *Write-Protect Enable (CR0.WP)*—The CR0.WP bit is introduced in "Write Protect (WP) Bit" on page 55.

5.6.1 No Execute (NX) Bit

The NX bit in the page-translation tables specifies whether instructions can be executed from the page. This bit is not checked during every instruction fetch. Instead, the NX bits in the page-translation-table entries are checked by the processor when the instruction TLB is loaded with a page translation. The processor attempts to load the translation into the instruction TLB when an instruction fetch misses the TLB. If a set NX bit is detected (indicating the page is not executable), a page-fault exception (#PF) occurs.

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. It is enabled by setting the NXE bit in the EFER register to 1 (see "Extended Feature Enable Register (EFER)" on page 67). Before setting this bit, system software must verify the processor supports the NX feature by checking the CPUID extended-feature flags (see "Function 8000_0001h: Processor Signature and AMD Features" in Volume 3).

5.6.2 User/Supervisor (U/S) Bit

The U/S bit in the page-translation tables determines the privilege level required to access the 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 "Segment-Protection Overview" on page 116 for more information on the protection-ring concept and CPL.

5.6.3 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).

5.6.4 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 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.

5.7 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 further qualifies the protection effect on the physical page.

Table 5-2 on page 175 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

Page-Map Level-4 Entry		Page-Directory-Pointer Entry		Page-Directory Entry		Page-Table Entry		Effective Result on Physical Page	
U/S	R/W	U/S	R/W	U/S	R/W	U/S	R/W	U/S	R/W
S	–	–	–	–	–	–	–	S	R/W
–	–	S	–	–	–	–	–		
–	–	–	–	S	–	–	–		
–	–	–	–	–	–	S	–		
U	R	U	–	U	–	U	–	U	R[1]
U	–	U	R	U	–	U	–		
U	–	U	–	U	R	U	–		
U	–	U	–	U	–	U	R		
U	R/W	U	R/W	U	R/W	U	R/W	U	R/W

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.

Note:
1. *Supervisor-level programs can access these pages as R/W.*

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 on page 176 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

Page-Map Level-4 Entry	Page Directory-Pointer Entry	Page Directory Entry	Page Table Entry	Physical Page
R/W	R/W	R/W	R/W	R/W
R	–	–	–	R
–	R	–	–	
–	–	R	–	
–	–	–	R	
W	W	W	W	W

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 175, 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-Management Instructions

System-management 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

Mnemonic	Name	Privilege			Reference
		CPL=0	O/S[1]	Any	
ARPL	Adjust Requestor Privilege Level			X	"Adjusting Access Rights" on page 188
CLI	Clear Interrupt Flag		X		"CLI and STI Instructions" on page 185
CLTS	Clear Task-Switched Flag in CR0	X			"CLTS Instruction" on page 185
HLT	Halt	X			"Processor Halt" on page 188
INT3	Interrupt to Debug Vector			X	"Breakpoint Instruction (INT3)" on page 399
INVD	Invalidate Caches	X			"Cache Management" on page 189
INVLPG	Invalidate TLB Entry	X			"TLB Invalidation" on page 189
IRET*x*	Interrupt Return (all forms)			X	"Returning From Interrupt Procedures" on page 281

Note:

1. *The operating system controls the privilege required to use the instruction.*

Table 6-1. System-Management Instructions (continued)

Mnemonic	Name	Privilege			Reference
		CPL=0	O/S[1]	Any	
LAR	Load Access-Rights Byte			X	"Checking Access Rights" on page 187
LGDT	Load Global-Descriptor-Table Register	X			"LGDT and LIDT Instructions" on page 187
LIDT	Load Interrupt-Descriptor-Table Register	X			
LLDT	Load Local-Descriptor-Table Register	X			"LLDT and LTR Instructions" on page 187
LMSW	Load Machine-Status Word	X			"LMSW and SMSW Instructions" on page 184
LSL	Load Segment Limit			X	"Checking Segment Limits" on page 188
LTR	Load Task Register	X			"LLDT and LTR Instructions" on page 187
MOV CR*n*	Move to/from Control Registers	X			"MOV CRn Instructions" on page 184
MOV DR*n*	Move to/from Debug Registers	X			"Accessing Debug Registers" on page 185
RDMSR	Read Model-Specific Register	X			"RDMSR and WRMSR Instructions" on page 186
RDPMC	Read Performance-Monitor Counter		X		"RDPMC Instruction" on page 186
RDTSC	Read Time-Stamp Counter		X		"RDTSC Instruction" on page 186
RSM	Return from System-Management Mode			X	"Leaving SMM" on page 335
SGDT	Store Global-Descriptor-Table Register			X	"SGDT and SIDT Instructions" on page 187
SIDT	Store Interrupt-Descriptor-Table Register			X	
SLDT	Store Local-Descriptor-Table Register			X	"SLDT and STR Instructions" on page 187
SMSW	Store Machine-Status Word			X	"LMSW and SMSW Instructions" on page 184
STI	Set Interrupt Flag		X		"CLI and STI Instructions" on page 185
STR	Store Task Register			X	"SLDT and STR Instructions" on page 187
SWAPGS	Swap GS and KernelGSbase Registers	X			"SWAPGS Instruction" on page 183
SYSCALL	Fast System Call			X	"SYSCALL and SYSRET" on page 180

Note:

1. *The operating system controls the privilege required to use the instruction.*

Table 6-1. System-Management Instructions (continued)

Mnemonic	Name	Privilege			Reference
		CPL=0	O/S[1]	Any	
SYSENTER	System Call			X	"SYSENTER and SYSEXIT (Legacy Mode Only)" on page 182
SYSEXIT	System Return	X			
SYSRET	Fast System Return	X			"SYSCALL and SYSRET" on page 180
UD2	Undefined Operation			X	"System Instruction Reference" in Volume 3
VERR	Verify Segment for Reads			X	"Checking Read/Write Rights" on page 188
VERW	Verify Segment for Writes			X	
WBINVD	Writeback and Invalidate Caches	X			"Cache Management" on page 189
WRMSR	Write Model-Specific Register	X			"RDMSR and WRMSR Instructions" on page 186

Note:

1. *The operating system controls the privilege required to use the instruction.*

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 "Processor Feature Identification" on page 75.
- 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 185.
- 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 186.

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 CPL) are consistent for all application and system processes. Only the CPL is allowed to vary. The processor assumes (but does not check) that the SYSCALL target CS has CPL=0 and the SYSRET target CS has CPL=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.

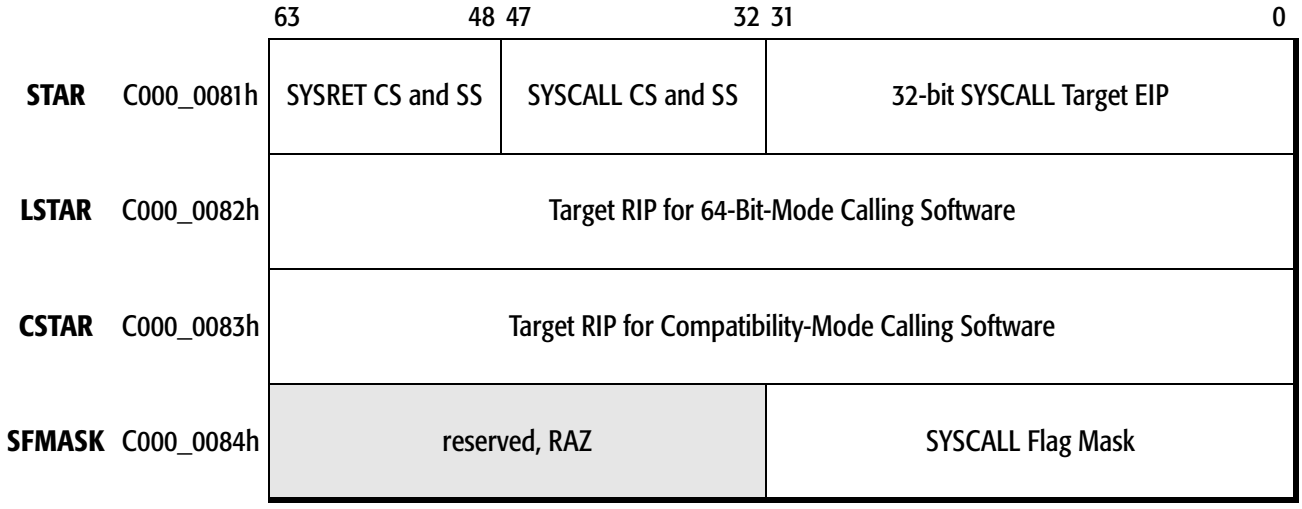


Figure 6-1. STAR, LSTAR, CSTAR, and MASK MSRs

- *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.**
- *SYSENTER Target EIP*—**Holds the offset into the CS of the called procedure.**

Figure 6-2 shows the register formats and their corresponding MSR IDs.

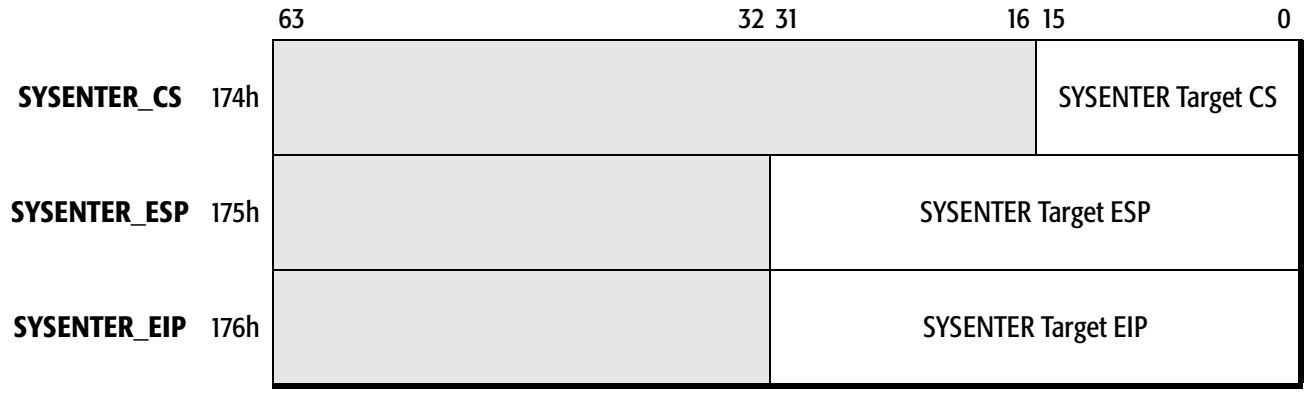


Figure 6-2. SYSENTER_CS, SYSENTER_ESP, SYSENTER_EIP MSRs

6.1.3 SWAPGS Instruction

The SWAPGS instruction provides a fast method for system software to load a pointer to system data structures. SWAPGS can be used upon entering system-software routines as a result of a SYSCALL instruction or as a result of an interrupt or exception. Before returning to application software, SWAPGS can restore an application data-structure pointer that was replaced by the system data-structure pointer.

SWAPGS exchanges the base-address value located in the KernelGSbase model-specific register (MSR address C000_0102h) with the base-address value located in the hidden portion of the GS selector register (GS.base). This exchange allows the system-kernel software to quickly access kernel data structures by using the GS segment-override prefix during memory references.

The need for SwapGS arises from the requirement that, upon entry to the OS kernel, the kernel needs to obtain a 64-bit pointer to its essential data structures. When using SYSCALL to implement system calls, no kernel stack exists at the OS entry point. Neither is there a straightforward method to obtain a pointer to kernel structures, from which the kernel stack pointer could be read. Thus, the kernel cannot save GPRs or reference memory. SwapGS does not require any GPR or memory operands, so no registers need to be saved before using it. Similarly, when the OS kernel is entered via an interrupt or exception (where the kernel stack is already set up), SwapGS can be used to quickly get a pointer to the kernel data structures.

See "FS and GS Registers in 64-Bit Mode" on page 86 for more information on using the GS.base register in 64-bit mode.

6.2 System Status and Control

System-status and system-control instructions are used to determine the features supported by a processor, gather information about the current execution state, and control the processor operating modes.

6.2.1 Processor Feature Identification (CPUID)

CPUID Instruction. The CPUID instruction provides complete information about the processor implementation and its capabilities. Software operating at any privilege level can execute the CPUID instruction to collect this information. System software normally uses the CPUID instruction to determine which optional features are available so the system can be configured appropriately. The optional features identified by the CPUID instruction are described in "CPUID" in Volume 3.

6.2.2 Accessing Control Registers

MOV CR*n* Instructions. The MOV CRn instructions can be used to copy data between the control registers and the general-purpose registers. These instructions are privileged and cause a general-protection exception (#GP) if non-privileged software attempts to execute them.

LMSW and SMSW Instructions. The machine status word is located in CR0 register bits 15–0. The *load machine status word* (LMSW) instruction writes only the least-significant four status-word bits (CR0[3:0]). All remaining status-word bits (CR0[15:4]) are

left unmodified by the instruction. The instruction is privileged and causes a #GP to occur if non-privileged software attempts to execute it.

The *store machine status word* (SMSW) instruction stores all 16 status-word bits (CR0[15:0]) into the target GPR or memory location. The instruction is not privileged and can be executed by all software.

CLTS Instruction. The *clear task-switched bit* instruction (CLTS) clears CR0.TS to 0. The CR0.TS bit is set to 1 by the processor every time a task switch takes place. The bit is useful to system software in determining when the x87 and multimedia register state should be saved or restored. See "Task Switched (TS) Bit" on page 54 for more information on using CR0.TS to manage x87-instruction state. The CLTS instruction is privileged and causes a #GP to occur if non-privileged software attempts to execute it.

6.2.3 Accessing the RFLAGs Register

The RFLAGS register contains both application and system bits. This section describes the instructions used to read and write system bits. Descriptions of instruction effects on application flags can be found in "Flags Register" in Volume 1 and "Instruction Effects on RFLAGS" in Volume 3.

POPF and PUSHF Instructions. The *pop and push RFLAGS* instructions are used for moving data between the rFLAGS register and the stack. They are not system-management instructions, but their behavior is mode-dependent.

CLI and STI Instructions. The *clear interrupt* (CLI) and *set interrupt* (STI) instructions modify only the RFLAGS.IF bit or RFLAGS.VIF bit. Clearing rFLAGS.IF to 0 causes the processor to ignore maskable interrupts. Setting RFLAGS.IF to 1 causes the processor to allow maskable interrupts.

See "Virtual Interrupts" on page 293 for more information on the operation of these instructions when virtual-8086 mode extensions are enabled (CR4.VME=1).

6.2.4 Accessing Debug Registers

The MOV DR*n* instructions are used to copy data between the debug registers and the general-purpose registers. These instructions are privileged and cause a general-protection exception (#GP) if non-privileged software attempts to execute

them. See "Debug Registers" on page 383 for a detailed description of the debug registers.

6.2.5 Accessing Model-Specific Registers

RDMSR and WRMSR Instructions. The *read/write model-specific register* instructions (RDMSR and WRMSR) can be used by privileged software to access the 64-bit MSRs. See "Model-Specific Registers (MSRs)" on page 69 for details about the MSRs.

RDPMC Instruction. The *read performance-monitoring counter* instruction, RDPMC, is used to read the model-specific performance-monitor registers, PerfCTR[3:0].

RDTSC Instruction. The *read time-stamp counter* instruction, RDTSC, is used to read the model-specific time-stamp counter (TSC) register.

6.3 Segment Register and Descriptor Register Access

The AMD64 architecture supports the legacy instructions that load and store segment registers and descriptor registers. In some cases the instruction capabilities are expanded to support long mode.

6.3.1 Accessing Segment Registers

MOV, POP, and PUSH Instructions. The MOV and POP instructions can be used to load a selector into a segment register from a general-purpose register or memory (MOV) or from the stack (POP). Any segment register, except the CS register, can be loaded with the MOV and POP instructions. The CS register must be loaded with a far-transfer instruction.

All segment register selectors can be stored in a general-purpose register or memory using the MOV instruction or pushed onto the stack using the PUSH instruction.

When a selector is loaded into a segment register, the processor automatically loads the corresponding descriptor-table entry into the hidden portion of the selector register. The hidden portion contains the base address, limit, and segment attributes.

Segment-load and segment-store instructions work normally in 64-bit mode. The appropriate entry is read from the system descriptor table (GDT or LDT) and is loaded into the hidden portion of the segment descriptor register. However, the

contents of data-segment and stack-segment descriptor registers are ignored, except in the case of the FS and GS segment-register base fields—see "FS and GS Registers in 64-Bit Mode" on page 86 for more information.

The ability to use segment-load instructions allows a 64-bit operating system to set up segment registers for a compatibility-mode application before switching to compatibility mode.

6.3.2 Accessing Descriptor-Table Registers

LGDT and LIDT Instructions. The *load GDTR* (LGDT) and *load IDTR* (LIDT) instructions load a *pseudo-descriptor* from memory into the GDTR or IDTR registers, respectively.

LLDT and LTR Instructions. The *load LDTR* (LLDT) and *load TR* (LTR) instructions load a system-segment descriptor from the GDT into the LDTR and TR segment-descriptor registers (hidden portion), respectively.

SGDT and SIDT Instructions. The *store GDTR* (SGDT) and *store IDTR* (SIDT) instructions reverse the operation of the LGDT and LIDT instructions. SGDT and SIDT store a pseudo-descriptor from the GDTR or IDTR register into memory.

SLDT and STR Instructions. In all modes, the *store LDTR* (SLDT) and *store TR* (STR) instructions store the LDT or task selector from the visible portion of the LDTR or TR register into a general-purpose register or memory, respectively. The hidden portion of the LDTR or TR register is not stored.

6.4 Protection Checking

Several instructions are provided to allow software to determine the outcome of a protection check before performing a memory access that could result in a protection violation. By performing the checks before a memory access, software can avoid violations that result in a general-protection exception (#GP).

6.4.1 Checking Access Rights

LAR Instruction. The *load access-rights* (LAR) instruction can be used to determine if access to a segment is allowed, based on privilege checks and type checks. The LAR instruction uses a segment-selector in the source operand to reference a descriptor in the GDT or LDT. LAR performs a set of access-rights checks and, if successful, loads the segment-descriptor

access rights into the destination register. Software can further examine the access-rights bits to determine if access into the segment is allowed.

6.4.2 Checking Segment Limits

LSL Instruction. The *load segment-limit* (LSL) instruction uses a segment-selector in the source operand to reference a descriptor in the GDT or LDT. LSL performs a set of preliminary access-rights checks and, if successful, loads the segment-descriptor limit field into the destination register. Software can use the limit value in comparisons with pointer offsets to prevent segment limit violations.

6.4.3 Checking Read/Write Rights

VERR and VERW Instructions. The *verify read-rights* (VERR) and *verify write-rights* (VERW) can be used to determine if a target code or data segment (not a system segment) can be read or written from the current privilege level (CPL). The source operand for these instructions is a pointer to the segment selector to be tested. If the tested segment (code or data) is readable from the current CPL, the VERR instruction sets RFLAGS.ZF to 1; otherwise, it is cleared to zero. Likewise, if the tested data segment is writable, the VERW instruction sets the RFLAGS.ZF to 1. A code segment cannot be tested for writability.

6.4.4 Adjusting Access Rights

ARPL Instruction. The *adjust RPL-field* (ARPL) instruction can be used by system software to prevent access into privileged-data segments by lower-privileged software. This can happen if an application passes a selector to system software and the selector RPL is less than (has greater privilege than) the calling-application CPL. To prevent this surrogate access, system software executes ARPL with the following operands:

- The destination operand is the data-segment selector passed to system software by the application.
- The source operand is the application code-segment selector (available on the system-software stack as a result of the CALL into system software by the application).

ARPL is not supported in 64-bit mode.

6.5 Processor Halt

The *processor halt* instruction (HLT) halts instruction execution, leaving the processor in the halt state. No registers or machine state are modified as a result of executing the HLT instruction.

The processor remains in the halt state until one of the following occurs:

- A non-maskable interrupt (NMI).
- An enabled, maskable interrupt (INTR).
- Processor reset (RESET).
- Processor initialization (INIT).
- System-management interrupt (SMI).

6.6 Cache and TLB Management

Cache-management instructions are used by system software to maintain coherency within the memory hierarchy. Memory coherency and caches are discussed in Chapter 7, "Memory System." Similarly, TLB-management instructions are used to maintain coherency between page translations cached in the TLB and the translation tables maintained by system software in memory. See "Translation-Lookaside Buffer (TLB)" on page 170 for more information.

6.6.1 Cache Management

WBINVD Instruction. The *writeback and invalidate* (WBINVD) instruction is used to write all modified cache lines to memory so that memory contains the most recent copy of data. After the writes are complete, the instruction invalidates all cache lines. This instruction operates on all caches in the memory hierarchy, including caches that are external to the processor.

INVD Instruction. The *invalidate* (INVD) instruction is used to invalidate all cache lines in all caches in the memory hierarchy. Unlike the WBINVD instruction, no modified cache lines are written to memory. The INVD instruction should only be used in situations where memory coherency is not required.

6.6.2 TLB Invalidation

INVLPG Instruction. The *invalidate TLB entry* (INVLPG) instruction can be used to invalidate specific entries within the TLB. The source operand is a virtual-memory address that specifies the TLB entry to be invalidated. Invalidating a TLB entry does not remove the associated page-table entry from the data cache. See "Translation-Lookaside Buffer (TLB)" on page 170 for more information.

7 Memory System

This chapter describes:

- Cache coherency mechanisms
- Cache control mechanisms
- Memory typing
- Memory mapped I/O
- Memory ordering rules
- Serializing instructions

Figure 7-1 on page 192 shows a conceptual picture of a processor and memory system, and how data and instructions flow between the various components. This diagram is not intended to represent a specific microarchitectural implementation but instead is used to illustrate the major memory-system components covered by this chapter.

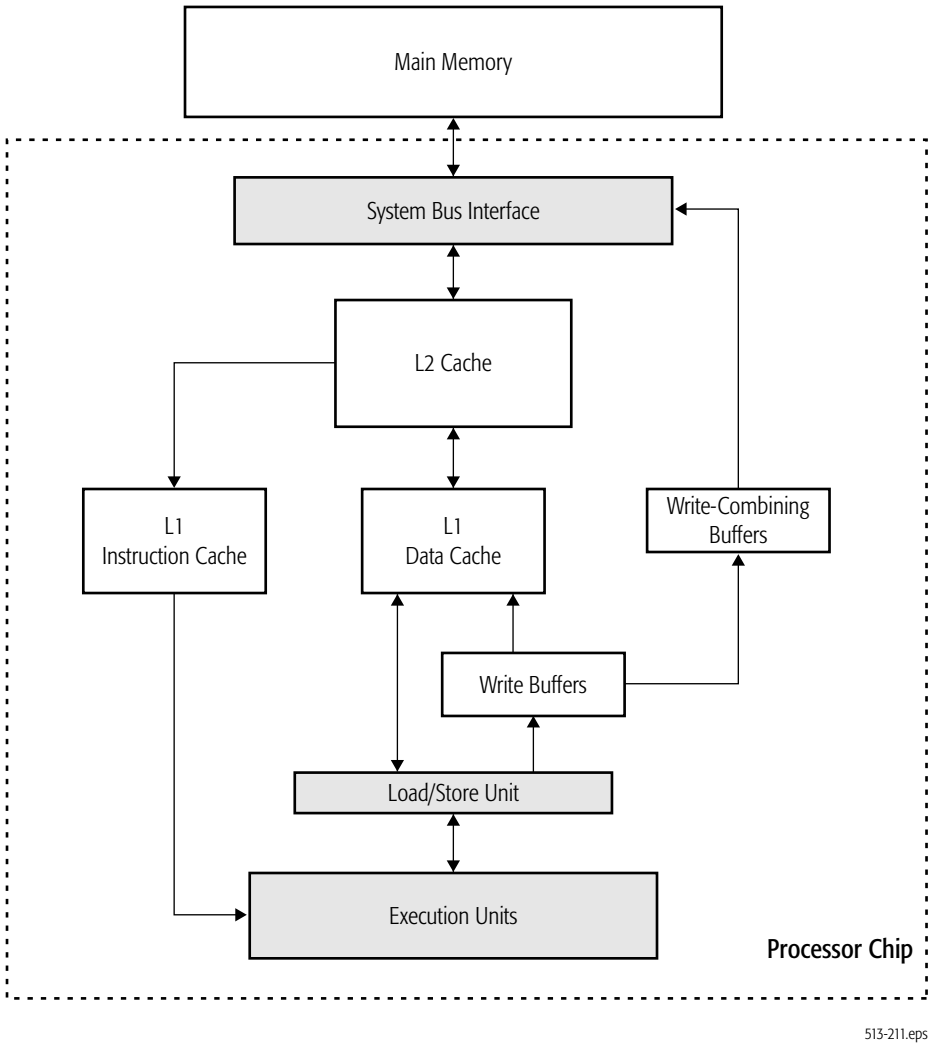


Figure 7-1. Processor and Memory System

The memory-system components described in this chapter are shown as *unshaded* boxes in Figure 7-1. Those items are summarized in the following paragraphs.

Main memory is external to the processor chip and is the memory-hierarchy level farthest from the processor execution units.

Caches are the memory-hierarchy levels closest to the processor execution units. They are much smaller and much faster than main memory, and can be either internal or external to the

processor chip. Caches contain copies of the most frequently used instructions and data. By allowing fast access to frequently used data, software can run much faster than if it had to access that data from main memory. Figure 7-1 shows three caches, all internal to the processor:

- *L1 Data Cache*—The L1 (level-1) data cache holds the data most recently read or written by the software running on the processor.
- *L1 Instruction Cache*—The L1 instruction cache is similar to the L1 data cache except that it holds only the instructions executed most frequently. In some processor implementations, the L1 instruction cache can be combined with the L1 data cache to form a unified L1 cache.
- *L2 Cache*—The L2 (level-2) cache is usually several times larger than the L1 caches, but it is also slower. It is common for L2 caches to be implemented as a unified cache containing both instructions and data. Recently used instructions and data that do not fit within the L1 caches can reside in the L2 cache. The L2 cache can be exclusive, meaning it does not cache information contained in the L1 cache. Conversely, inclusive L2 caches contain a copy of the L1-cached information.

Memory-read operations from cacheable memory first check the cache to see if the requested information is available. A *read hit* occurs if the information is available in the cache, and a *read miss* occurs if the information is not available. Likewise, a *write hit* occurs if the memory write can be stored in the cache, and a *write miss* occurs if it cannot be stored in the cache.

Caches are divided into fixed-size blocks called *cache lines*. The cache allocates lines to correspond to regions in memory of the same size as the cache line, aligned on an address boundary equal to the cache-line size. For example, in a cache with 32-byte lines, the cache lines are aligned on 32-byte boundaries and byte addresses 0007h and 001Eh are both located in the same cache line. The size of a cache line is implementation dependent. Most implementations have either 32-byte or 64-byte cache lines.

The process of loading data into a cache is a *cache-line fill*. Even if only a single byte is requested, all bytes in a cache line are loaded from memory. Typically, a cache-line fill must remove (evict) an existing cache line to make room for the new line

loaded from memory. This process is called *cache-line replacement*. If the existing cache line was modified before the replacement, the processor performs a cache-line *writeback* to main memory when it performs the cache-line fill.

Cache-line writebacks help maintain *coherency* (consistency) between the caches and main memory. Internally, the processor can also maintain cache coherency by *internally probing* (checking) the other caches and write buffers for a more recent version of the requested data. External devices can also check processor caches for more recent versions of data by *externally probing* the processor. Throughout this document, the term *probe* is used to refer to external probes, while internal probes are always qualified with the word *internal*.

Write buffers temporarily hold data writes when main memory or the caches are busy with other memory accesses. The existence of write buffers is implementation dependent.

Implementations of the architecture can use *write-combining buffers* if the order and size of non-cacheable writes to main memory is not important to the operation of software. These buffers can combine multiple, individual writes to main memory and transfer the data in fewer bus transactions.

7.1 Memory-Access Ordering

The flexibility in which memory accesses can be ordered is closely related to the flexibility in which a processor implementation can *execute* and *retire* instructions. Instruction execution *creates* results and status and determines whether or not the instruction causes an exception. Instruction retirement *commits* the results of instruction execution, in program order, to software-visible resources such as memory, caches, write-combining buffers, and registers, or it causes an exception to occur if instruction execution created one.

Implementations of the AMD64 architecture retire instructions in program order, but implementations can execute instructions *in any order*. Implementations can also *speculatively execute* instructions—executing instructions before knowing they are needed. Internally, implementations manage data reads and writes so that instructions complete in order. However, because implementations can execute instructions out of order and speculatively, the sequence of memory accesses can also be out

of program order (weakly ordered). Processor implementations adhere to the following rules governing memory accesses, which can be further restricted depending on the memory type being accessed:

7.1.1 Read Ordering

Generally, reads do not affect program order because they do not affect the state of software-visible resources. However, some system devices might be sensitive to reads. In such a situation software can map a read-sensitive device to a memory type that enforces strong read-ordering, or use read/write barrier instructions to force strong read-ordering.

For cacheable memory types, the following rules govern read ordering:

- Out-of-order reads are allowed. Out-of-order reads can occur as a result of out-of-order instruction execution or speculative execution. The processor can read memory out-of-order to allow out-of-order execution to proceed.
- Speculative reads are allowed. A speculative read occurs when the processor begins executing a memory-read instruction before it knows the instruction will actually complete. For example, the processor can predict a branch will occur and begin executing instructions following the predicted branch before it knows whether the prediction is valid. When one of the speculative instructions reads data from memory, the read itself is speculative.
- Reads can be reordered ahead of writes. Reads are generally given a higher priority by the processor than writes because instruction execution stalls if the read data required by an instruction is not immediately available. Allowing reads ahead of writes usually maximizes software performance.
- A read *cannot* be reordered ahead of a prior write if the read is from the same location as the prior write. In this case, the read instruction stalls until the write instruction completes execution. The read instruction requires the result of the write instruction for proper software operation. For cacheable memory types, the write data can be forwarded to the read instruction before it is actually written to memory.

7.1.2 Write Ordering

Writes affect program order because they affect the state of software-visible resources. The following rules govern write ordering:

- Generally, out-of-order writes are *not* allowed. Write instructions executed out of order cannot commit (write) their result to memory until all previous instructions have completed in program order. The processor can, however, hold the result of an out-of-order write instruction in a private buffer (not visible to software) until that result can be committed to memory.
- It is possible for writes to *write-combining* memory types to appear to complete out of order, relative to writes into other memory types. See "Memory Types" on page 200 and "Write Combining" on page 205 for additional information.
- Speculative writes are *not* allowed. As with out-of-order writes, speculative write instructions cannot commit their result to memory until all previous instructions have completed in program order. Processors can hold the result in a private buffer (not visible to software) until the result can be committed.
- Write buffering is allowed. When a write instruction completes and commits its result, that result can be buffered before actually writing the result into a memory location in program order. Although the write buffer itself is not directly accessible by software, the results in the buffer are accessible during memory accesses to the locations that are buffered. For cacheable memory types, the write buffer can be read out-of-order and speculatively read, just like memory.
- Write combining is allowed. In some situations software can relax the write-ordering rules and allow several writes to be combined into fewer writes to memory. When write-combining is used, it is possible for writes to other memory types to proceed ahead of (out-of-order) memory-combining writes, unless the writes are to the same address. Write-combining should be used only when the order of writes does not affect program order (for example, writes to a graphics frame buffer).

7.1.3 Read/Write Barriers

When the order of memory accesses must be strictly enforced, software can use read/write barrier instructions to force reads and writes to proceed in program order. Read/write barrier instructions force all prior reads or writes to complete before subsequent reads or writes are executed. The LFENCE, SFENCE, and MFENCE instructions are provided as dedicated read, write, and read/write barrier instructions (respectively).

Serializing instructions, I/O instructions, and locked instructions can also be used as read/write barriers.

Table 7-1 on page 203 shows the memory-access ordering possible for each memory type supported by the AMD64 architecture.

7.2 Memory Coherency and Protocol

Implementations that support caching support a cache-coherency protocol for maintaining coherency between main memory and the caches. The cache-coherency protocol is also used to maintain coherency between all processors in a multiprocessor system. The cache-coherency protocol supported by the AMD64 architecture is the *MOESI* (modified, owned, exclusive, shared, invalid) protocol. The states of the MOESI protocol are:

- *Invalid*—A cache line in the invalid state does not hold a valid copy of the data. Valid copies of the data can be either in main memory or another processor cache.
- *Exclusive*—A cache line in the exclusive state holds the most recent, correct copy of the data. The copy in main memory is also the most recent, correct copy of the data. No other processor holds a copy of the data.
- *Shared*—A cache line in the shared state holds the most recent, correct copy of the data. Other processors in the system may hold copies of the data in the shared state, as well. The copy in main memory is also the most recent, correct copy of the data (since no other processor holds it in *owned* state).
- *Modified*—A cache line in the modified state holds the most recent, correct copy of the data. The copy in main memory is stale (incorrect), and no other processor holds a copy.
- *Owned*—A cache line in the owned state holds the most recent, correct copy of the data. The owned state is similar to the shared state in that other processors can hold a copy of the most recent, correct data. Unlike the shared state, however, the copy in main memory can be stale (incorrect). Only one processor can hold the data in the owned state—all other processors must hold the data in the shared state.

Figure 7-2 on page 198 shows the general MOESI state transitions possible with various types of memory accesses. This

is a logical software view, not a hardware view, of how cache-line state transitions. Instruction-execution activity and external-bus transactions can both be used to modify the cache MOESI state in multiprocessing or multi-mastering systems.

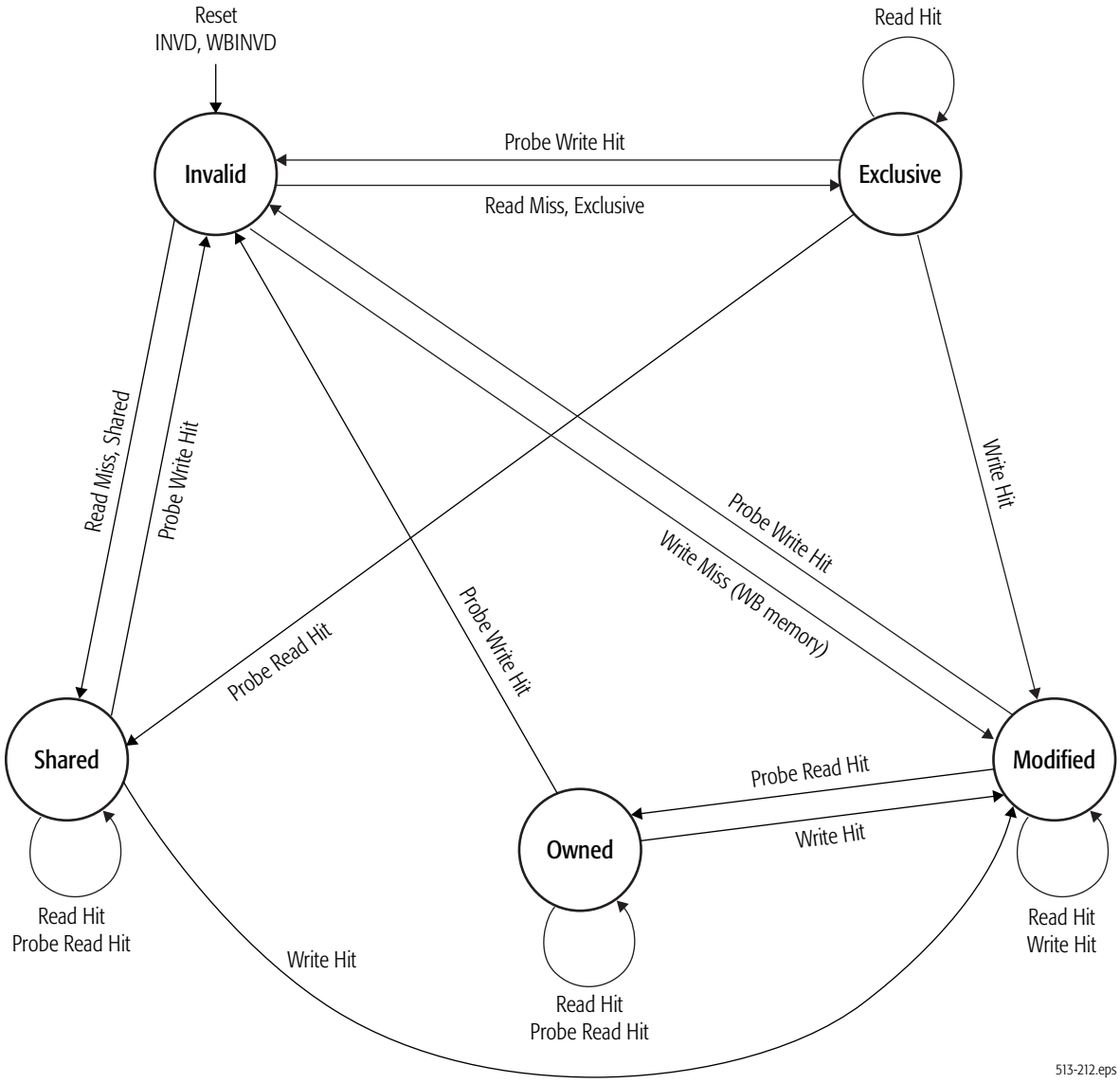


Figure 7-2. MOESI State Transitions

To maintain memory coherency, external bus masters (typically other processors with their own internal caches) need to acquire the most recent copy of data before caching it

internally. That copy can be in main memory or in the internal caches of other bus-mastering devices. When an external master has a cache read-miss or write-miss, it *probes* the other mastering devices to determine whether the most recent copy of data is held in any of their caches. If one of the other mastering devices holds the most recent copy, it provides it to the requesting device. Otherwise, the most recent copy is provided by main memory.

There are two general types of bus-master probes:

- Read probes indicate the external master is requesting the data for read purposes.
- Write probes indicate the external master is requesting the data for the purpose of modifying it.

Referring back to Figure 7-2, the state transitions involving probes are initiated by other processors and external bus masters into the processor. Some read probes are initiated by devices that intend to cache the data. Others, such as those initiated by I/O devices, do not intend to cache the data. Some processor implementations do not change the data MOESI state if the read probe is initiated by a device that does not intend to cache the data.

State transitions involving read misses and write misses can cause the processor to generate probes into external bus masters and to read main memory.

Read hits do not cause a MOESI-state change. Write hits generally cause a MOESI-state change into the modified state. If the cache line is already in the modified state, a write hit does not change its state.

The specific operation of external-bus signals and transactions and how they influence a cache MOESI state are implementation dependent. For example, an implementation could convert a write miss to a WB memory type into two separate MOESI-state changes. The first would be a read-miss placing the cache line in the exclusive state. This would be followed by a write hit into the exclusive cache line, changing the cache-line state to modified.

7.2.1 Special Coherency Considerations

In some cases, data can be modified in a manner that is impossible for the memory-coherency protocol to handle due to the effects of instruction prefetching. In such situations

software must use serializing instructions and/or cache-invalidation instructions to guarantee subsequent data accesses are coherent.

An example of this type of a situation is a page-table update followed by accesses to the physical pages referenced by the updated page tables. The following sequence of events shows what can happen when software changes the translation of virtual-page *A* from physical-page *M* to physical-page *N*:

1. Software invalidates the TLB entry. The tables that translate virtual-page *A* to physical-page *M* are now held only in main memory. They are not cached by the TLB.
2. Software changes the page-table entry for virtual-page A in main memory to point to physical-page *N* rather than physical-page *M*.
3. Software accesses data in virtual-page A.

During Step 3, software expects the processor to access the data from physical-page *N*. However, it is possible for the processor to prefetch the data from physical-page *M* before the page table for virtual-page *A* is updated in Step 2. This is because the physical-memory references for the *page tables* are different than the physical-memory references for the *data*. Because the physical-memory references are different, the processor does not recognize them as requiring coherency checking and believes it is safe to prefetch the data from virtual-page A, which is translated into a read from physical page M.

To prevent this problem, software should insert a read/write barrier instruction (typically a serializing instruction) immediately after the page-table update to force subsequent instructions to access data using the correct virtual-page-to-physical-page translation. The serializing instructions INVLPG and MOV CR3, used to flush the TLB, can be used for this purpose.

7.3 Memory Types

The AMD64 architecture defines the following memory types:

- *Uncacheable (UC)*—Reads from, and writes to, UC memory are not cacheable. Reads from UC memory cannot be speculative. Write-combining to UC memory is not allowed.

Reads from UC memory cause the write buffers to be written to memory and invalidated.

The UC memory type is useful for memory-mapped I/O devices where strict ordering of reads and writes is important.

- *Cache Disable (CD)*—The CD memory type is a form of uncacheable memory type that occurs when caches are disabled (CR0.CD=1 and CR0.NW=0). With CD memory, it is possible for the address to be cached due to an earlier cacheable access, or due to two virtual-addresses aliasing to a single physical address.

  For the L1 data cache and the L2 cache, reads from, and writes to, CD memory that hit the cache cause the cache line to be invalidated before accessing main memory. If the cache line is in the modified state, the line is written to main memory and then invalidated.

  For the L1 instruction cache, reads from CD memory that hit the cache read the cached instructions rather than access main memory. Reads that miss the cache access main memory and do not cause cache-line replacement.

- *Write-Combining (WC)*—Reads from, and writes to, WC memory are not cacheable. Reads from WC memory can be speculative.

  Writes to this memory type can be combined internally by the processor and written to memory as a single write operation to reduce memory accesses. For example, four word writes to consecutive addresses can be combined by the processor into a single quadword write, resulting in one memory access instead of four.

  The WC memory type is useful for graphics-display memory buffers where the order of writes is not important.

- *Write-Protect (WP)*—Reads from WP memory are cacheable and allocate cache lines on a read miss. Reads from WP memory can be speculative.

  Writes to WP memory that hit in the cache do not update the cache. Instead, all writes update memory (write to memory), and writes that hit in the cache invalidate the cache line. Write buffering of WP memory is allowed.

  The WP memory type is useful for shadowed-ROM memory where updates must be immediately visible to all devices that read the shadow locations.

- *Writethrough (WT)*—Reads from WT memory are cacheable and allocate cache lines on a read miss. Reads from WT memory can be speculative.

  All writes to WT memory update main memory, and writes that hit in the cache update the cache line (cache lines remain in the same state after a write that hits a cache line). Writes that miss the cache do not allocate a cache line. Write buffering of WT memory is allowed.
- *Writeback (WB)*—Reads from WB memory are cacheable and allocate cache lines on a read miss. Cache lines can be allocated in the shared, exclusive, or modified states. Reads from WB memory can be speculative.

  All writes that hit in the cache update the cache line and place the cache line in the modified state. Writes that miss the cache allocate a new cache line and place the cache line in the modified state. Writes to main memory only take place during writeback operations. Write buffering of WB memory is allowed.

  The WB memory type provides the highest-possible performance and is useful for most software and data stored in system memory (DRAM).

Table 7-1 on page 203 shows the memory access ordering possible for each memory type supported by the AMD64 architecture. Table 7-2 on page 203 shows the caching policy for the same memory types.

Table 7-1. Memory Access by Memory Type

Memory Access Allowed		Memory Type				
		UC/CD	WC	WP	WT	WB
Read	Out-of-Order	no	yes	yes	yes	yes
	Speculative	no	yes	yes	yes	yes
	Reorder Before Write	no	yes	yes	yes	yes
Write	Out-of-Order[1]	no	yes	no	no	no
	Speculative	no	no	no	no	no
	Buffering	no	yes	yes	yes	yes
	Combining[2]	no	yes	no	yes	yes

Note:

1. *Out-of-order relative to writes into other memory types.*
2. *Write-combining buffers are separate from write buffers.*

Table 7-2. Caching Policy by Memory Type

Caching Policy	Memory Type					
	UC	CD	WC	WP	WT	WB
Read Cacheable	no	no	no	yes	yes	yes
Write Cacheable	no	no	no	no	yes	yes
Read Allocate	no	no	no	yes	yes	yes
Write Allocate	no	no	no	no	no	yes
Write Hits Update Memory	yes	yes[1]	yes[2]	yes[3]	yes	no

Note:

1. *For the L1 data cache and the L2 cache, if an access hits the cache, the cache line is invalidated. If the cache line is in the modified state, the line is written to main memory and then invalidated. For the L1 instruction cache, read hits access the cache rather than main memory.*
2. *The data is not cached, so a cache write hit cannot occur. However, memory is updated.*
3. *Write hits update memory and invalidate the cache line.*

7.4 Buffering and Combining Memory Writes

7.4.1 Write Buffering

Writes to memory (main memory and caches) can be stored internally by the processor in *write buffers* before actually writing the data into a memory location. System performance can be improved by buffering writes, as shown in the following examples:

- When higher-priority memory transactions, such as reads, compete for memory access with writes, writes can be delayed in favor of reads, which minimizes or eliminates an instruction-execution stall due to a memory-operand read.
- When the memory is busy, buffering writes while the memory is busy removes the writes from the instruction-execution pipeline, which frees instruction-execution resources.

The processor manages the write buffer so that it is transparent to software. Memory accesses check the write buffer, and the processor completes writes into memory from the buffer in program order. Also, the processor completely empties the write buffer by writing the contents to memory as a result of performing any of the following operations:

- *SFENCE Instruction*—Executing a store-fence (SFENCE) instruction forces all memory writes before the SFENCE (in program order) to be written into memory before memory writes that follow the SFENCE instruction. The memory-fence (MFENCE) instruction has a similar effect, but it forces the ordering of loads in addition to stores.
- *Serializing Instructions*—Executing a serializing instruction forces the processor to retire the serializing instruction (complete both instruction execution and result writeback) before the next instruction is fetched from memory.
- *I/O instructions*—Before completing an I/O instruction, all previous reads and writes must be written to memory, and the I/O instruction must complete before completing subsequent reads or writes. Writes to I/O-address space (OUT instruction) are never buffered.
- *Locked Instructions*—Before completing a locked instruction (an instruction executed using the LOCK prefix), all previous reads and writes must be written to memory, and the locked instruction must complete before completing

subsequent writes. Locked writes are never buffered, although locked reads and writes are cacheable.

- *Interrupts and Exceptions*—Interrupts and exceptions are serializing events that force the processor to write all results from the write buffer to memory before fetching the first instruction from the interrupt or exception service routine.
- *UC-Memory Reads*—UC-memory reads are not reordered ahead of writes.

Write buffers can behave similarly to *write-combining buffers* because multiple writes are collected internally before transferring the data to caches or main memory. See the following section for a description of write combining.

7.4.2 Write Combining

Write-combining memory uses a different buffering scheme than write buffering described above. Writes to write-combining (WC) memory can be combined internally by the processor in a buffer for more efficient transfer to main memory at a later time. For example, 16 doubleword writes to consecutive memory addresses can be combined in the WC buffers and transferred to main memory as a single burst operation rather than as individual memory writes.

The following instructions perform writes to WC memory:

- MASKMOVDQU
- MASKMOVQ
- MOVNTDQ
- MOVNTI
- MOVNTPD
- MOVNTPS
- MOVNTQ

WC memory is not cacheable. A WC buffer writes its contents only to main memory.

The size and number of WC buffers available is implementation dependent. The processor assigns an address range to an empty WC buffer when a WC-memory write occurs. The size and alignment of this address range is equal to the buffer size. All subsequent writes to WC memory that fall within this address range can be stored by the processor in the WC-buffer entry until an event occurs that causes the processor to write the WC buffer to main memory. After the WC buffer is written to main

memory, the processor can assign a new address range on a subsequent WC-memory write.

Writes to consecutive addresses in WC memory are not required for the processor to combine them. The processor combines any WC memory write that falls within the active-address range for a buffer. Multiple writes to the same address overwrite each other (in program order) until the WC buffer is written to main memory.

It is possible for writes to proceed out of program order when WC memory is used. For example, a write to cacheable memory that follows a write to WC memory can be written into the cache before the WC buffer is written to main memory. For this reason, and the reasons listed in the previous paragraph, software that is sensitive to the order of memory writes should avoid using WC memory.

WC buffers are written to main memory under the same conditions as the write buffers, namely when:

- Executing a store-fence (SFENCE) instruction.
- Executing a serializing instruction.
- Executing an I/O instruction.
- Executing a locked instruction (an instruction executed using the LOCK prefix).
- An interrupt or exception occurs.

WC buffers are also written to main memory when:

- A subsequent non-write-combining operation has a write address that matches the WC-buffer active-address range.
- A write to WC memory falls outside the WC-buffer active-address range. The existing buffer contents are written to main memory, and a new address range is established for the latest WC write.

7.5 Memory Caches

The AMD64 architecture supports the use of internal and external caches. The size, organization, coherency mechanism, and replacement algorithm for each cache is implementation dependent. Generally, the existence of the caches is transparent to both application and system software. In some cases, however, software can use cache-structure information to

optimize memory accesses or manage memory coherency. Such software can use the extended-feature functions of the CPUID instruction to gather information on the caching subsystem supported by the processor. For more information on using CPUID in this manner, see "Functions 8000_0005h and 8000_0006h: Cache Information" in Volume 3.

7.5.1 Cache Organization and Operation

Although the detailed organization of a processor cache depends on the implementation, the general constructs are similar. L1 caches—data and instruction, or unified—and L2 caches usually are implemented as n-way set-associative caches. Figure 7-3 on page 208 shows a typical *logical* organization of an n-way set-associative cache. The physical implementation of the cache can be quite different.

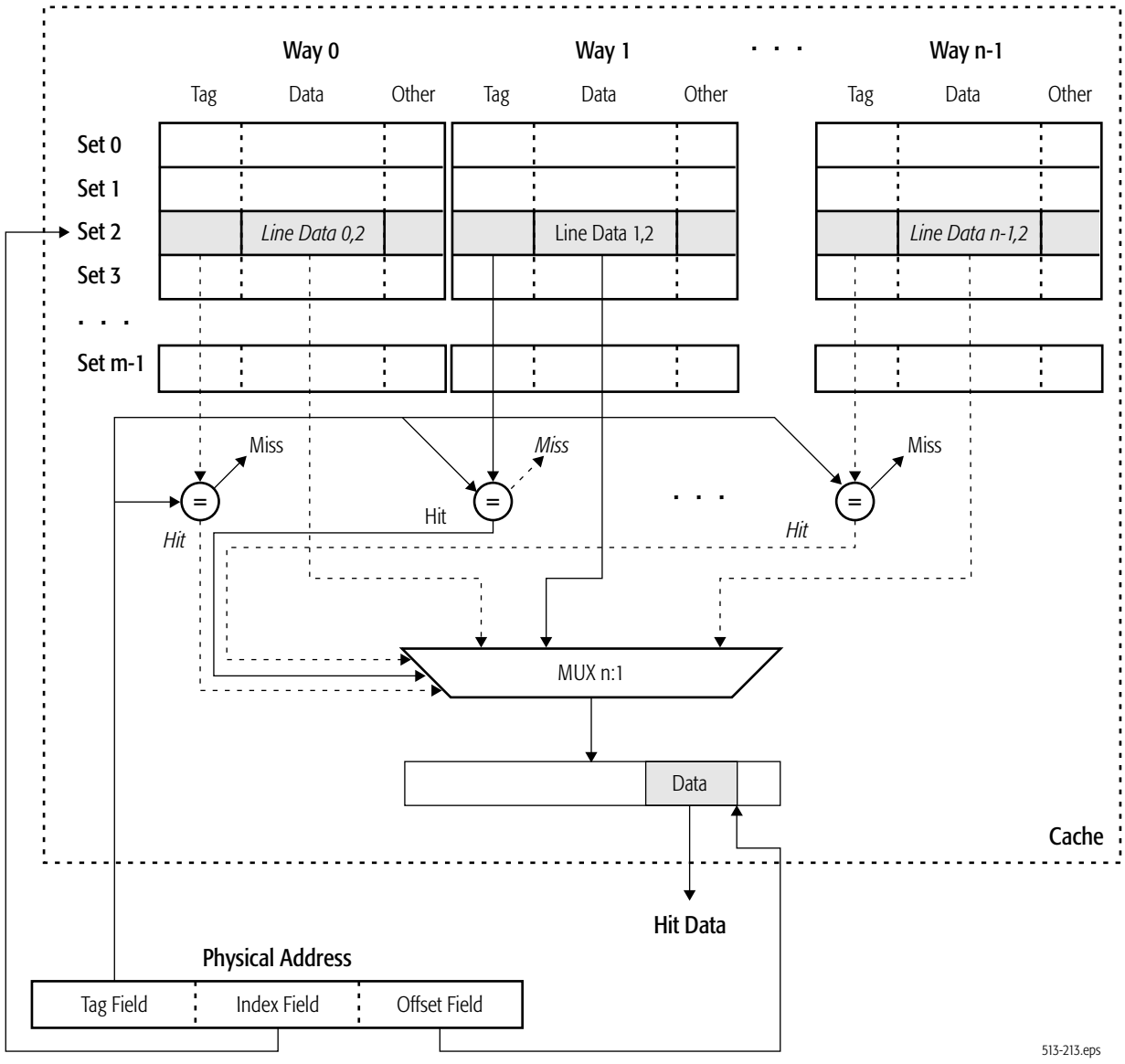


Figure 7-3. Cache Organization Example

As shown in Figure 7-3, the cache is organized as an array of cache lines. Each cache line consists of three parts: a cache-data line (a fixed-size copy of a memory block), a tag, and other information. Rows of cache lines in the cache array are *sets*, and columns of cache lines are *ways*. In an n-way set-associative cache, each set is a collection of n lines. For example, in a four-

way set-associative cache, each set is a collection of four cache lines, one from each way.

The cache is accessed using the physical address of the data or instruction being referenced. To access data within a cache line, the physical address is used to select the set, way, and byte from the cache. This is accomplished by dividing the physical address into the following three fields:

- *Index*—The *index field* selects the cache set (row) to be examined for a hit. All cache lines within the set (one from each way) are selected by the index field.
- *Tag*—The *tag field* is used to select a specific cache line from the cache set. The physical-address tag field is compared with each cache-line tag in the set. If a match is found, a cache hit is signalled, and the appropriate cache line is selected from the set. If a match is not found, a cache miss is signalled.
- *Offset*—The *offset field* points to the first byte in the cache line corresponding to the memory reference. The referenced data or instruction value is read from (or written to, in the case of memory writes) the selected cache line starting at the location selected by the offset field.

In Figure 7-3 on page 208, the physical-address index field is shown selecting Set 2 from the cache. The tag entry for each cache line in the set is compared with the physical-address tag field. The tag entry for Way 1 matches the physical-address tag field, so the cache-line data for Set 2, Way 1 is selected using the n:1 multiplexor. Finally, the physical-address offset field is used to point to the first byte of the referenced data (or instruction) in the selected cache line.

Cache lines can contain other information in addition to the data and tags, as shown in Figure 7-3. MOESI state and the state bits associated with the cache-replacement algorithm are typical pieces of information kept with the cache line. Instruction caches can also contain pre-decode or branch-prediction information. The type of information stored with the cache line is implementation dependent.

Self-Modifying Code. Software that writes into a code segment is classified as self-modifying code. To avoid cache-coherency problems due to self-modifying code, a check is made during data writes to see whether the data-memory location

corresponds to a code-segment memory location. If it does, implementations of the AMD64 architecture invalidate the corresponding *instruction-cache* line(s) during the data-memory write. Entries in the data cache are not invalidated, and it is possible for the modified instruction to be cached by the data cache following the memory write. A subsequent fetch of the modified instruction goes to main memory to get the coherent version of the instruction. If the data cache holds the most recent copy of the instruction rather than main memory, it provides that copy.

The processor determines whether a write is in a code segment by internally probing the instruction cache and prefetched instructions. If the internal probe returns a hit, the instruction-cache line and prefetched instructions are invalidated. The internal probes into the instruction cache and prefetch hardware are always performed using the *physical address* of an instruction in order to avoid potential aliasing problems associated with using virtual (linear) addresses.

7.5.2 Cache Control Mechanisms

The AMD64 architecture provides a number of mechanisms for controlling the cacheability of memory. These are described in the following sections.

Cache Disable. Bit 30 of the CR0 register is the cache-disable bit, CR0.CD. Caching is enabled when CR0.CD is cleared to 0, and caching is disabled when CR0.CD is set to 1. When caching is disabled, reads and writes access main memory.

Software can disable the cache while the cache still holds valid data (or instructions). If a read or write hits the L1 data cache or the L2 cache when CR0.CD=1, the processor does the following:

1. Writes the cache line back if it is in the modified or owned state.
2. Invalidates the cache line.
3. Performs a non-cacheable main-memory access to read or write the data.

If an instruction fetch hits the L1 instruction cache when CR0.CD=1, the processor reads the cached instructions rather than access main memory.

The processor also responds to cache probes when CR0.CD=1. Probes that hit the cache cause the processor to perform Step 1. Step 2 (cache-line invalidation) is performed only if the probe is performed on behalf of a memory write or an exclusive read.

Writethrough Disable. Bit 29 of the CR0 register is the *not writethrough* disable bit, CR0.NW. In early x86 processors, CR0.NW is used to control cache writethrough behavior, and the combination of CR0.NW and CR0.CD determines the cache operating mode.

In early x86 processors, clearing CR0.NW to 0 enables writeback caching for main memory, effectively disabling writethrough caching for main memory. When CR0.NW=0, software can disable writeback caching for specific memory pages or regions by using other cache control mechanisms. When software sets CR0.NW to 1, writeback caching is disabled for main memory, while writethrough caching is enabled.

In implementations of the AMD64 architecture, CR0.NW is not used to qualify the cache operating mode established by CR0.CD. Table 7-3 shows the effects of CR0.NW and CR0.CD on the AMD64 architecture cache-operating modes.

Table 7-3. AMD64 Architecture Cache-Operating Modes

CR0.CD	CR0.NW	Cache Operating Mode
0	0	Cache enabled with a writeback-caching policy.
0	1	Invalid setting—causes a general-protection exception (#GP).
1	0	Cache disabled. See "Cache Disable" on page 210.
1	1	

Page-Level Cache Disable. Bit 4 of all paging data-structure entries controls page-level cache disable (PCD). When a data-structure-entry PCD bit is cleared to 0, the page table or physical page pointed to by that entry is cacheable, as determined by the CR0.CD bit. When the PCD bit is set to 1, the page table or physical page is not cacheable. The PCD bit in the paging data-structure base-register (bit 4 in CR3) controls the cacheability of the highest-level page table in the page-translation hierarchy.

Page-Level Writethrough Enable. Bit 3 of all paging data-structure entries is the page-level writethrough enable control (PWT). When a data-structure-entry PWT bit is cleared to 0, the page table or physical page pointed to by that entry has a writeback caching policy. When the PWT bit is set to 1, the page table or physical page has a writethrough caching policy. The PWT bit in the paging data-structure base-register (bit 3 in CR3) controls the caching policy of the highest-level page table in the page-translation hierarchy.

The corresponding PCD bit must be cleared to 0 (page caching enabled) for the PWT bit to have an effect.

Memory Typing. Two mechanisms are provided for software to control access to and cacheability of specific memory regions:

- The memory-type range registers (MTRRs) control cacheability based on physical addresses. See “MTRRs” on page 217 for more information on the use of MTRRs.
- The page-attribute table (PAT) mechanism controls cacheability based on virtual addresses. PAT extends the capabilities provided by the PCD and PWT page-level cache controls. See “Page-Attribute Table Mechanism” on page 228 for more information on the use of the PAT mechanism.

System software can combine the use of both the MTRRs and PAT mechanisms to maximize control over memory cacheability.

If the MTRRs are disabled in implementations that support the MTRR mechanism, the default memory type is set to uncacheable (UC). Memory accesses are not cached even if the caches are enabled by clearing CR0.CD to 0. Cacheable memory types must be established using the MTRRs in order for memory accesses to be cached.

Cache Control Precedence. The cache-control mechanisms are used to define the memory type and cacheability of main memory and regions of main memory. Taken together, the most restrictive memory type takes precedence in defining the caching policy of memory. The order of precedence is:

1. Uncacheable (UC)
2. Write-combining (WC)

3. Write-protected (WP)
4. Writethrough (WT)
5. Writeback (WB)

For example, assume a large memory region is designated a writethrough type using the MTRRs. Individual pages within that region can have caching disabled by setting the appropriate page-table PCD bits. However, no pages within that region can have a writeback caching policy, regardless of the page-table PWT values.

7.5.3 Cache and Memory Management Instructions

Data Prefetch. The prefetch instructions are used by software as a hint to the processor that the referenced data is likely to be used in the near future. The processor can preload the cache line containing the data in anticipation of its use. PREFETCH provides a hint that the data is to be read. PREFETCHW provides a hint that the data is to be written. The processor can mark the line as modified if it is preloaded using PREFETCHW.

Memory Ordering. Instructions are provided for software to enforce memory ordering (serialization) in weakly-ordered memory types. These instructions are:

- *SFENCE (store fence)*—forces all memory writes (stores) preceding the SFENCE (in program order) to be written into memory before memory writes following the SFENCE.
- *LFENCE (load fence)*—forces all memory reads (loads) preceding the LFENCE (in program order) to be read from memory before memory reads following the LFENCE.
- *MFENCE (memory fence)*—forces all memory accesses (reads and writes) preceding the MFENCE (in program order) to be written into or read from memory before memory accesses following the MFENCE.

Cache Line Flush. The CLFLUSH instruction (writeback, if modified, and invalidate) takes the byte memory-address operand (a linear address), and checks to see if the address is cached. If the address is cached, the entire cache line containing the address is invalidated. If any portion of the cache line is dirty (in the modified or owned state), the entire line is written to main memory before it is invalidated. CLFLUSH affects *all caches* in the memory hierarchy—internal and external to the processor. The checking and invalidation

process continues until the address has been invalidated in all caches.

In most cases, the underlying memory type assigned to the address has no effect on the behavior of this instruction. However, when the underlying memory type for the address is UC or WC (as defined by the MTRRs), the processor does not proceed with checking all caches to see if the address is cached. In both cases, the address is uncacheable, and invalidation is unnecessary. Write-combining buffers are written back to memory if the corresponding physical address falls within the buffer active-address range.

Cache Writeback and Invalidate. Unlike the CLFLUSH instruction, the WBINVD instruction operates on the entire cache, rather than a single cache line. The WBINVD instruction first writes back all cache lines that are dirty (in the modified or owned state) to main memory. After writeback is complete, the instruction invalidates all cache lines. The checking and invalidation process continues until all internal caches are invalidated. A special bus cycle is transmitted to higher-level external caches directing them to perform a writeback-and-invalidate operation.

Cache Invalidate. The INVD instruction is used to invalidate all cache lines. Unlike the WBINVD instruction, dirty cache lines are not written to main memory. The process continues until all internal caches have been invalidated. A special bus cycle is transmitted to higher-level external caches directing them to perform an invalidation.

The INVD instruction should only be used in situations where memory coherency is not required.

7.5.4 Serializing Instructions

Serializing instructions force the processor to retire the serializing instruction and all previous instructions before the next instruction is fetched. A serializing instruction is retired when the following operations are complete:

- The instruction has executed.
- All registers modified by the instruction are updated.
- All memory updates performed by the instruction are complete.
- All data held in the write buffers have been written to memory.

Serializing instructions can be used as a barrier between memory accesses to force strong ordering of memory operations. Care should be exercised in using serializing instructions because they modify processor state and affect program flow. The instructions also force execution serialization, which can significantly degrade performance. When strongly-ordered memory accesses are required, but execution serialization is not, it is recommended that software use the memory-ordering instructions described on page 213.

The following are serializing instructions:

- *Non-Privileged Instructions*
  - CPUID
  - IRET
  - RSM
- *Privileged Instructions*
  - MOV CRn
  - MOV DRn
  - LGDT, LIDT, LLDT, LTR
  - SWAPGS
  - WRMSR
  - WBINVD, INVD
  - INVLPG

7.6 Memory-Type Range Registers

The AMD64 architecture supports three mechanisms for software access-control and cacheability-control over memory regions. These mechanisms can be used in place of similar capabilities provided by external chipsets used with early x86 processors.

This section describes a control mechanism that uses a set of programmable model-specific registers (MSRs) called the *memory-type-range registers* (MTRRs). The MTRR mechanism provides system software with the ability to manage hardware-device memory mapping. System software can characterize physical-memory regions by type (e.g. ROM, flash, memory-mapped I/O) and assign hardware devices to the appropriate physical-memory type.

Another control mechanism is implemented as an extension to the page-translation capability and is called the *page attribute table* (PAT). It is described in "Page-Attribute Table Mechanism" on page 228. Like the MTRRs, PAT provides system software with the ability to manage hardware-device memory mapping. With PAT, however, system software can characterize physical pages and assign virtually-mapped devices to those physical pages using the page-translation mechanism. PAT may be used in conjunction with the MTTR mechanism to maximize flexibility in memory control.

Finally control mechanisms are provided for managing memory-mapped I/O. These mechanisms employ extensions to the MTRRs and a separate feature called the *top-of-memory registers*. The MTRR extensions include additional MTRR type-field encodings for fixed-range MTRRs and variable-range I/O range registers (IORRs). These mechanisms are described in "Memory-Mapped I/O" on page 232.

7.6.1 MTRR Type Fields

The MTRR mechanism provides a means for characterizing physical-address ranges with a memory type (see "Memory Types" on page 200). The MTRRs contain a type field used to specify the memory type in effect for a given physical-address range.

There are two variants of the memory type-field encodings: standard and extended. Both the standard and extended encodings use type-field bits 2–0 to specify the memory type. For the standard encodings, bits 7–3 are reserved and must be zero. For the extended encodings, bits 7–5 are reserved, but bits 4–3 are defined as the RdMem and WrMem bits. "Extended Fixed-Range MTRR Type-Field Encodings" on page 233 describes the function of these extended bits and how software enables them. Only the fixed-range MTRRs support the extended type-field encodings. Variable-range MTRRs use the standard encodings.

Table 7-4 on page 217 shows the memory types supported by the MTRR mechanism and their encoding in the MTRR type fields referenced throughout this section. Unless the extended type-field encodings are explicitly enabled, the processor uses the type values shown in Table 7-4.

Table 7-4. MTRR Type Field Encodings

Type Value	Type Name	Type Description
00h	UC–Uncacheable	All accesses are uncacheable. Write combining is not allowed. Speculative accesses are not allowed
01h	WC–Write-Combining	All accesses are uncacheable. Write combining is allowed. Speculative reads are allowed
04h	WT–Writethrough	Reads allocate cache lines on a cache miss. Cache lines are not allocated on a write miss. Write hits update the cache and main memory.
05h	WP–Write-Protect	Reads allocate cache lines on a cache miss. All writes update main memory. Cache lines are not allocated on a write miss. Write hits invalidate the cache line and update main memory.
06h	WB–Writeback	Reads allocate cache lines on a cache miss, and can allocate to either the shared, exclusive, or modified state. Writes allocate to the modified state on a cache miss.

If the MTRRs are disabled in implementations that support the MTRR mechanism, the default memory type is set to uncacheable (UC). *Memory accesses are not cached even if the caches are enabled by clearing CR0.CD to 0.* Cacheable memory types must be established using the MTRRs to enable memory accesses to be cached.

7.6.2 MTRRs

Both fixed-size and variable-size address ranges are supported by the MTRR mechanism. The fixed-size ranges are restricted to the lower 1 Mbyte of physical-address space, while the variable-size ranges can be located anywhere in the physical-address space.

Figure 7-4 on page 218 shows an example mapping of physical memory using the fixed-size and variable-size MTRRs. The areas shaded gray are not mapped by the MTRRs. Unmapped areas are set to the software selected default memory type.

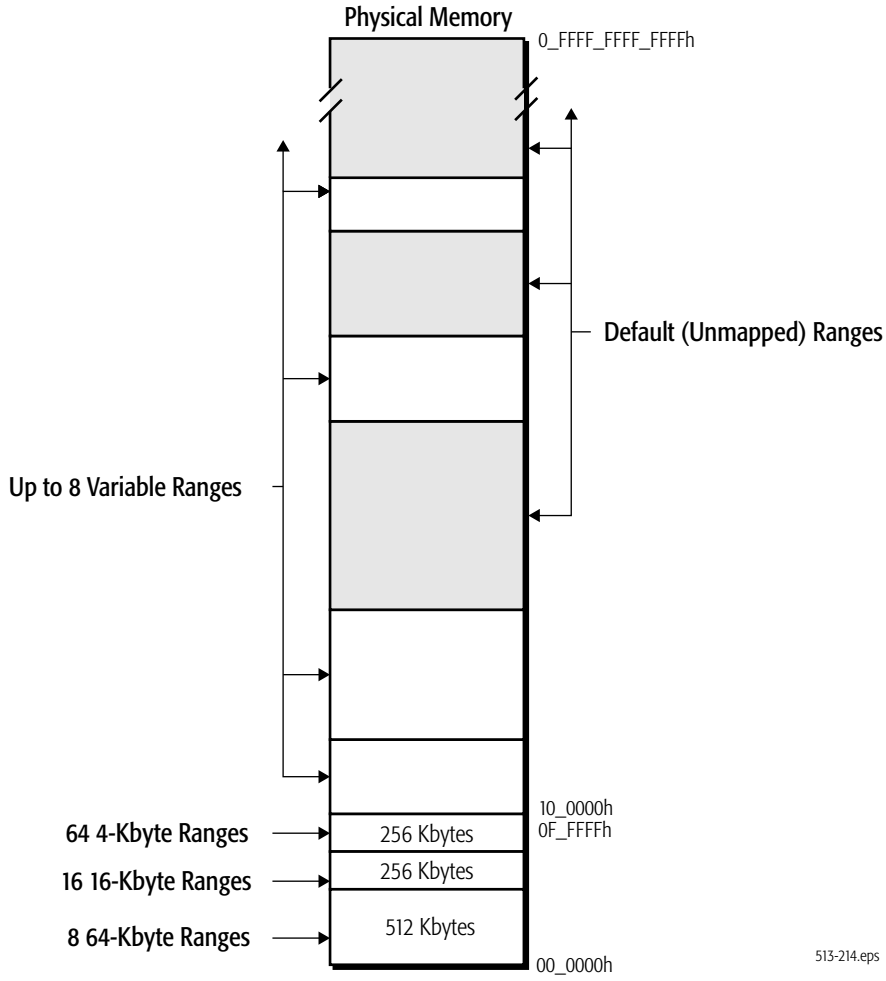


Figure 7-4. MTRR Mapping of Physical Memory

MTRRs are 64-bit model-specific registers (MSRs). They are read using the RDMSR instruction and written using the WRMSR instruction. See "Memory-Typing MSRs" on page 438 for a listing of the MTRR MSR numbers. The following sections describe the types of MTRRs and their function.

Fixed-Range MTRRs. The fixed-range MTRRs are used to characterize the first 1 Mbyte of physical memory. Each fixed-range MTRR contains eight type fields for characterizing a total of eight memory ranges. Fixed-range MTRRs support extended type-field encodings as described in "Extended Fixed-Range MTRR Type-Field Encodings" on page 233. The extended type

field allows a fixed-range MTRR to be used as a fixed-range IORR. Figure 7-5 shows the format of a fixed-range MTRR.

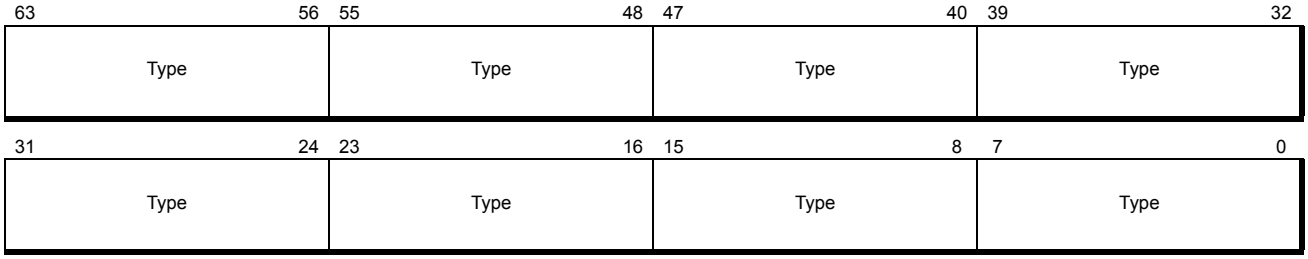


Figure 7-5. Fixed-Range MTRR

For the purposes of memory characterization, the first 1 Mbyte of physical memory is segmented into a total of 88 non-overlapping memory ranges, as follows:

- The 512 Kbytes of memory spanning addresses 00_0000h to 07_FFFFh are segmented into eight 64-Kbyte ranges. A single MTRR is used to characterize this address space.
- The 256 Kbytes of memory spanning addresses 08_0000h to 0B_FFFFh are segmented into 16 16-Kbyte ranges. Two MTRRs are used to characterize this address space.
- The 256 Kbytes of memory spanning addresses 0C_0000h to 0F_FFFFh are segmented into 64 4-Kbyte ranges. Eight MTRRs are used to characterize this address space.

Table 7-5 on page 220 shows the address ranges corresponding to the type fields within each fixed-range MTRR. The gray-shaded heading boxes represent the bit ranges for each type field in a fixed-range MTTR. See Table 7-4 on page 217 for the type-field encodings.

Table 7-5. Fixed-Range MTRR Address Ranges

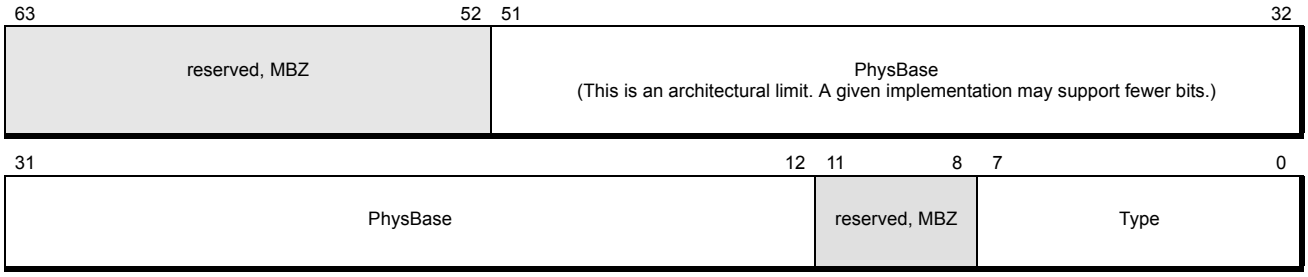
Physical Address Range (in hexadecimal)								Register Name
63–56	55–48	47–40	39–32	31–24	23–16	15–8	7–0	
70000–7FFFF	60000–6FFFF	50000–5FFFF	40000–4FFFF	30000–3FFFF	20000–2FFFF	10000–1FFFF	00000–0FFFF	MTRRfix64K_00000
9C000–9FFFF	98000–9BFFF	94000–97FFF	90000–93FFF	8C000–8FFFF	88000–8BFFF	84000–87FFF	80000–83FFF	MTRRfix16K_80000
BC000–BFFFF	B8000–BBFFF	B4000–B7FFF	B0000–B3FFF	AC000–AFFFF	A8000–ABFFF	A4000–A7FFF	A0000–A3FFF	MTRRfix16K_A0000
C7000–C7FFF	C6000–C6FFF	C5000–C5FFF	C4000–C4FFF	C3000–C3FFF	C2000–C2FFF	C1000–C1FFF	C0000–C0FFF	MTRRfix4K_C0000
CF000–CFFFF	CE000–CEFFF	CD000–CDFFF	CC000–CCFFF	CB000–CBFFF	CA000–CAFFF	C9000–C9FFF	C8000–C8FFF	MTRRfix4K_C8000
D7000–D7FFF	D6000–D6FFF	D5000–D5FFF	D4000–D4FFF	D3000–D3FFF	D2000–D2FFF	D1000–D1FFF	D0000–D0FFF	MTRRfix4K_D0000
DF000–DFFFF	DE000–DEFFF	DD000–DDFFF	DC000–DCFFF	DB000–DBFFF	DA000–DAFFF	D9000–D9FFF	D8000–D8FFF	MTRRfix4K_D8000
E7000–E7FFF	E6000–E6FFF	E5000–E5FFF	E4000–E4FFF	E3000–E3FFF	E2000–E2FFF	E1000–E1FFF	E0000–E0FFF	MTRRfix4K_E0000
EF000–EFFFF	EE000–EEFFF	ED000–EDFFF	EC000–ECFFF	EB000–EBFFF	EA000–EAFFF	E9000–E9FFF	E8000–E8FFF	MTRRfix4K_E8000
F7000–F7FFF	F6000–F6FFF	F5000–F5FFF	F4000–F4FFF	F3000–F3FFF	F2000–F2FFF	F1000–F1FFF	F0000–F0FFF	MTRRfix4K_F0000
FF000–FFFFF	FE000–FEFFF	FD000–FDFFF	FC000–FCFFF	FB000–FBFFF	FA000–FAFFF	F9000–F9FFF	F8000–F8FFF	MTRRfix4K_F8000

Variable-Range MTRRs. The variable-range MTRRs 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 the MTRR. Two variable-range MTRRs are used to characterize each address range: MTRRphysBase*n* and MTRRphysMask*n* (*n* is the address-range number from 0 to 7). For example, address-range 3 is characterized using the MTRRphysBase3 and MTRRphysMask3 register pair.

Figure 7-6 shows the format of the MTRRphysBase*n* register and Figure 7-7 on page 222 shows the format of the MTRRphysMask*n* register. The fields within the register pair are read/write.

MTRRphysBase*n* Registers. The fields in these variable-range MTRRs, shown in Figure 7-6, are:

- *Type*—Bits 7–0. The memory type used to characterize the memory range. See Table 7-4 on page 217 for the type-field encodings. Variable-range MTRRs do not support the extended type-field encodings.
- *Range Physical Base-Address (PhysBase)*—Bits 51–12. The memory-range base-address in physical-address space. PhysBase is aligned on a 4-Kbyte (or greater) address in the 52-bit physical-address space supported by the AMD64 architecture. PhysBase represents the most-significant 40-address bits of the physical address. Physical-address bits 11–0 are assumed to be 0.

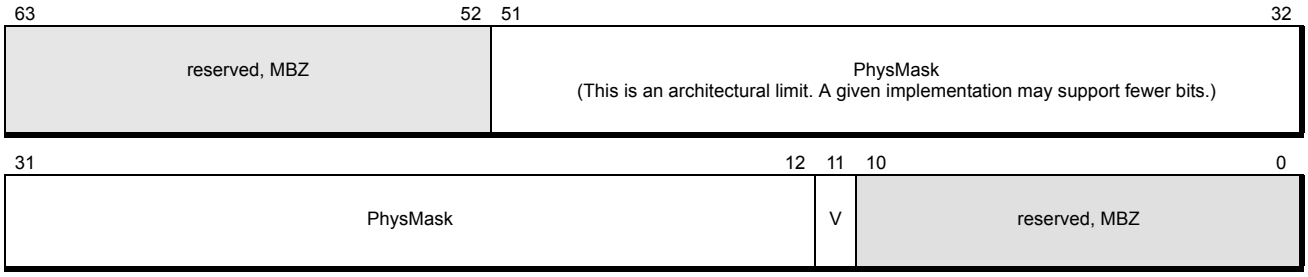


Bits	Mnemonic	Description	R/W
63-52	reserved	Reserved, Must be Zero	
51-12	PhysBase	Range Physical Base Address	R/W
11-8	reserved	Reserved, Must be Zero	
7-0	Type	Default Memory Type	R/W

Figure 7-6. MTRRphysBase*n* Register

MTRRphysMask*n* Registers. The fields in these variable-range MTRRs , shown in Figure 7-7 on page 222, are:

- *Valid (V)*—Bit 11. Indicates that the MTRR pair is valid (enabled) when set to 1. When the valid bit is cleared to 0 the register pair is not used.
- *Range Physical Mask (PhysMask)*—Bits 51–12. The mask value used to specify the memory range. Like PhysBase, PhysMask is aligned on a 4-Kbyte physical-address boundary. Bits 11–0 of PhysMask are assumed to be 0.



Bits	Mnemonic	Description	R/W
63-52	reserved	Reserved, Must be Zero	
51-12	PhysMask	Range Physical Mask	R/W
11	V	MTRR Pair Enable (Valid)	R/W
10-0	reserved	Reserved, Must be Zero	

Figure 7-7. MTRRphysMask*n* Register

PhysMask and PhysBase are used together to determine whether a target physical-address falls within the specified address range. PhysMask is logically ANDed with PhysBase and separately ANDed with the upper 40 bits of the target physical-address. If the results of the two operations are identical, the target physical-address falls within the specified memory range. The pseudo-code for the operation is:

```
MaskBase = PhysMask AND PhysBase
MaskTarget = PhysMask AND Target_Address[51:12]
if MaskBase = MaskTarget
   then Target_Address_In_Range
   else Target_Address_Not_In_Range
```

Variable Range Size and Alignment. The size and alignment of variable memory-ranges (MTRRs) and I/O ranges (IORRs) are restricted as follows:

- The boundary on which a variable range is aligned must be equal to the range size. For example, a memory range of 16 Mbytes must be aligned on a 16-Mbyte boundary.
- The range size must be a power of 2 (2^n, $52 > n > 11$), with a minimum allowable size of 4 Kbytes. For example, 4 Mbytes and 8 Mbytes are allowable memory range sizes, but 6 Mbytes is not allowable.

PhysMask and PhysBase Values. Software can calculate the PhysMask value using the following procedure:

1. Subtract the memory-range physical base-address from the upper physical-address of the memory range.

2. Subtract the value calculated in Step 1 from the physical memory size.
3. Truncate the lower 12 bits of the result in Step 2 to create the PhysMask value to be loaded into the MTRRphysMask*n* register. Truncation is performed by right-shifting the value 12 bits.

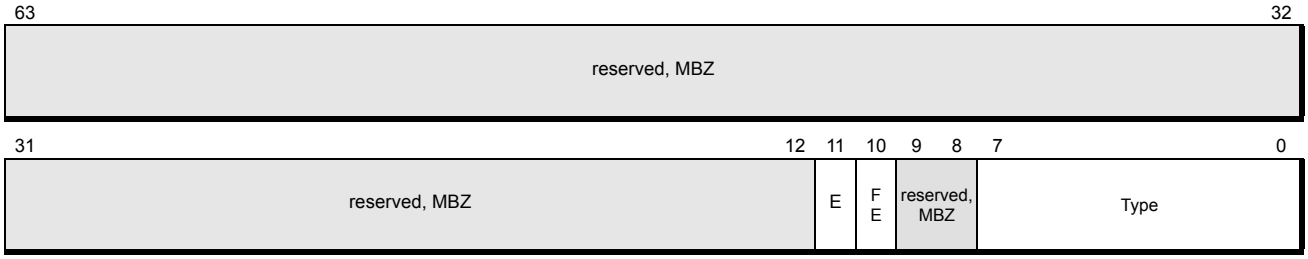
For example, assume a 32-Mbyte memory range is specified within the 52-bit physical address space, starting at address 200_0000h. The upper address of the range is 3FF_FFFFh. Following the process outlined above yields:

1. 3FF_FFFFh–200_0000h = 1FF_FFFFh
2. F_FFFF_FFFF_FFFF–1FF_FFFFh = F_FFFF_FE00_0000h
3. Right shift (F_FFFF_FE00_0000h) by 12 = FF_FFFF_E000h

In this example, the 40-bit value loaded into the PhysMask field is FF_FFFF_E000h.

Software must also truncate the lower 12 bits of the physical base-address before loading it into the PhysBase field. In the example above, the 40-bit PhysBase field is 00_0000_2000h.

Default-Range MTRRs. Physical addresses that are not within ranges established by fixed-range and variable-range MTRRs are set to a default memory-type using the MTRRdefType register. The format of this register is shown in Figure 7-8.



Bits	Mnemonic	Description	R/W
63-12	reserved	Reserved, Must be Zero	
11	E	MTRR Enable	R/W
10	FE	Fixed Range Enable	R/W
9-8	reserved	Reserved, Must be Zero	
7-0	Type	Default Memory Type	R/W

Figure 7-8. MTRR defType Register Format

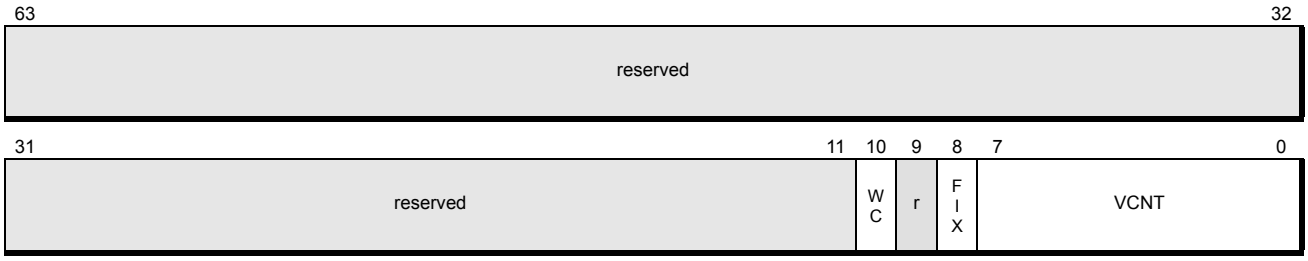
The fields within the MTRRdefType register are read/write. These fields are:

- *Type*—Bits 7–0. The default memory-type used to characterize physical-memory space. See Table 7-4 on page 217 for the type-field encodings. The extended type-field encodings are not supported by this register.
- *Fixed-Range Enable (FE)*—Bit 10. All fixed-range MTRRs are enabled when FE is set to 1. Clearing FE to 0 disables all fixed-range MTRRs. Setting and clearing FE has no effect on the variable-range MTRRs. The FE bit has no effect unless the E bit is set to 1 (see below).
- *MTRR Enable (E)*—Bit 11. This is the MTRR enable bit. All fixed-range and variable-range MTRRs are enabled when E is set to 1. Clearing E to 0 disables all fixed-range and variable-range MTRRs and sets the default memory-type to uncacheable (UC) regardless of the value of the Type field.

7.6.3 Using MTRRs

Identifying MTRR Features. Software determines whether a processor supports the MTRR mechanism by executing the CPUID instruction with either standard-function 1 or extended-function 8000_0001h. If MTRRs are supported, bit 12 in the EDX register is set to 1 by CPUID. See “Processor Feature Identification” on page 75 for more information on the CPUID instruction.

The MTRR capability register (MTRRcap) is a read-only register containing information describing the level of MTRR support provided by the processor. Figure 7-9 on page 225 shows the format of this register. If MTRRs are supported, software can read MTRRcap using the RDMSR instruction. Attempting to write to the MTRRcap register causes a general-protection exception (#GP).



Bits	Mnemonic	Description	R/W
63-11	reserved	Reserved	
10	WC	Write Combining	R
9	reserved	Reserved	
8	FIX	Fixed-Range Registers	R
7-0	VCNT	Variable-Range Register Count	R

Figure 7-9. MTRR Capability Register Format

The MTRRcap register field are:

- *Variable-Range Register Count (VCNT)*—Bits 7–0. The VCNT field contains the number of variable-range register pairs supported by the processor. For example, a processor supporting eight register pairs returns a 08h in this field.
- *Fixed-Range Registers (FIX)*—Bit 8. The FIX bit indicates whether or not the fixed-range registers are supported. If the processor returns a 1 in this bit, *all* fixed-range registers are supported. If the processor returns a 0 in this bit, *no* fixed-range registers are supported.
- *Write-Combining (WC)*—Bit 10. The WC bit indicates whether or not the write-combining memory type is supported. If the processor returns a 1 in this bit, WC memory is supported, otherwise it is not supported.

7.6.4 MTRRs and Page Cache Controls

When paging and the MTRRs are both enabled, the address ranges defined by the MTRR registers can span multiple pages, each of which can characterize memory with different types (using the PCD and PWT page bits). When caching is enabled (CR0.CD=0 and CR0.NW=0), the *effective memory-type* is determined as follows:

1. If the page is defined as cacheable and writeback (PCD=0 and PWT=0), then the MTRR defines the effective memory-type.
2. If the page is defined as not cacheable (PCD=1), then UC is the effective memory-type.

3. If the page is defined as cacheable and writethrough (PCD=0 and PWT=1), then the MTRR defines the effective memory-type *unless* the MTRR specifies WB memory, in which case WT is the effective memory-type.

Table 7-6 lists the MTRR and page-level cache-control combinations and their combined effect on the final memory-type, if the PAT register holds the default settings.

Table 7-6. Combined MTRR and Page-Level Memory Type with Unmodified PAT MSR

MTRR Memory Type	Page PCD Bit	Page PWT Bit	Effective Memory-Type
UC	–	–	UC
WC	0	–	WC
	1	0	WC[1]
	1	1	UC
WP	0	–	WP
	1	–	UC
WT	0	–	WT
	1	–	UC
WB	0	0	WB
	0	1	WT
	1	–	UC

Note:

1. The effective memory-type resulting from the combination of PCD=1, PWT=0 and an MTRR WC memory type is implementation dependent.

Large Page Sizes. When paging is enabled, software can use large page sizes (2 Mbytes and 4 Mbytes) in addition to the more typical 4-Kbyte page size. When large page sizes are used, it is possible for multiple MTRRs to span the memory range within a single large page. Each MTRR can characterize the regions within the page with different memory types. If this occurs, the effective memory-type used by the processor within the large page is undefined.

Software can avoid the undefined behavior in one of the following ways:

- Avoid using multiple MTRRs to characterize a single large page.
- Use multiple 4-Kbyte pages rather than a single large page.
- If multiple MTRRs must be used within a single large page, software can set the MTRR type fields to the same value.
- If the multiple MTRRs must have different type-field values, software can set the large page PCD and PWT bits to the most restrictive memory type defined by the multiple MTRRs.

Overlapping MTRR Registers. If the address ranges of two or more MTRRs overlap, the following rules are applied to determine the memory type used to characterize the overlapping address range:

1. Fixed-range MTRRs, which characterize only the first 1 Mbyte of physical memory, have precedence over variable-range MTRRs.
2. If two or more variable-range MTRRs overlap, the following rules apply:
   A. If the memory types are identical, then that memory type is used.
   B. If at least one of the memory types is UC, the UC memory type is used.
   C. If at least one of the memory types is WT, and the only other memory type is WB, then the WT memory type is used.
   D. If the combination of memory types is not listed Steps 1 through 3 immediately above, then the memory type used is undefined.

7.6.5 MTRRs in Multi-Processing Environments

In multi-processing environments, the MTRRs located in all processors must characterize memory in the same way. Generally, this means identical values are written to the MTRRs used by the processors. Processor implementations *do not* check the MTRR settings in other processors to ensure consistency. It is the responsibility of system software to initialize and maintain MTRR consistency across all processors.

7.7 Page-Attribute Table Mechanism

The page-attribute table (PAT) mechanism extends the page-table entry format and enhances the capabilities provided by the PCD and PWT page-level cache controls. PAT (and PCD, PWT) allow memory-type characterization based on the virtual (linear) address. The PAT mechanism provides the same memory-typing capabilities as the MTRRs but with the added flexibility of the paging mechanism. Software can use both the PAT and MTRR mechanisms to maximize flexibility in memory-type control.

7.7.1 PAT Register

Like the MTRRs, the PAT register is a 64-bit model-specific register (MSR). The format of the PAT registers is shown in Figure 7-10. See "Memory-Typing MSRs" on page 438 for more information on the PAT MSR number and reset value.

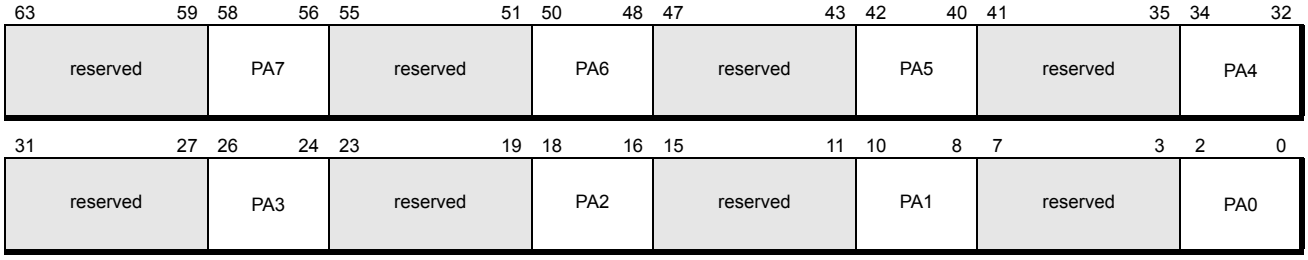


Figure 7-10. PAT Register

The PAT register contains eight page-attribute (PA) fields, numbered from PA0 to PA7. The PA fields hold the encoding of a memory type, as found in Table 7-7 on page 229. The PAT type-encodings match the MTRR type-encodings, with the exception that PAT adds the 07h encoding. The 07h encoding corresponds to a *UC-* type. The UC- type (07h) is identical to the UC type (00h) except it can be overridden by an MTRR type of WC.

Software can write any supported memory-type encoding into any of the eight PA fields. An attempt to write anything but zeros into the reserved fields causes a general-protection exception (#GP). An attempt to write an unsupported type encoding into a PA field also causes a #GP exception.

The PAT register fields are initiated at processor reset to the default values shown in Table 7-8 on page 230.

Table 7-7. PAT Type Encodings

Type Value	Type Name	Type Description
00h	UC–Uncacheable	All accesses are uncacheable. Write combining is not allowed. Speculative accesses are not allowed.
01h	WC–Write-Combining	All accesses are uncacheable. Write combining is allowed. Speculative reads are allowed.
04h	WT–Writethrough	Reads allocate cache lines on a cache miss, but only to the shared state. Cache lines are not allocated on a write miss. Write hits update the cache and main memory.
05h	WP–Write-Protect	Reads allocate cache lines on a cache miss, but only to the shared state. All writes update main memory. Cache lines are not allocated on a write miss. Write hits invalidate the cache line and update main memory.
06h	WB–Writeback	Reads allocate cache lines on a cache miss, and can allocate to either the shared or exclusive state. Writes allocate to the modified state on a cache miss.
07h	UC- (UC minus)	All accesses are uncacheable. Write combining is not allowed. Speculative accesses are not allowed. Can be overridden by an MTRR with the WC type.

7.7.2 PAT Indexing

PA fields in the PAT register are selected using three bits from the page-table entries. These bits are:

- *PAT (page attribute table)*—The PAT bit is bit 7 in 4-Kbyte PTEs; it is bit 12 in 2-Mbyte and 4-Mbyte PDEs. Page-table entries that don't have a PAT bit (PML4 entries, for example) assume PAT = 0.
- *PCD (page cache disable)*—The PCD bit is bit 4 in all page-table entries. The PCD from the PTE or PDE is selected depending on the paging mode.
- *PWT (page writethrough)*—The PWT bit is bit 3 in all page-table entries. The PWT from the PTE or PDE is selected depending on the paging mode.

Table 7-8 on page 230 shows the various combinations of the PAT, PCD, and PWT bits used to select a PA field within the PAT register. Table 7-8 also shows the default memory-type values established in the PAT register by the processor after a reset. The default values correspond to the memory types established by the PCD and PWT bits alone in processor implementations that do not support the PAT mechanism. In such

implementations, the PAT field in page-table entries is reserved and cleared to 0. See "Page-Translation-Table Entry Fields" on page 166 for more information on the page-table entries.

Table 7-8. PAT-Register PA-Field Indexing

Page-Table Entry Bits			PAT Register Field	Default Memory Type
PAT	PCD	PWT		
0	0	0	PA0	WB
0	0	1	PA1	WT
0	1	0	PA2	UC-[1]
0	1	1	PA3	UC
1	0	0	PA4	WB
1	0	1	PA5	WT
1	1	0	PA6	UC-[1]
1	1	1	PA7	UC

Note:
1. Can be overridden by WC memory type set by an MTRR.

7.7.3 Identifying PAT Support

Software determines whether a processor supports the PAT mechanism by executing the CPUID instruction with either standard-function 1 or extended-function 8000_0001h. If PAT is supported, bit 16 in the EDX register is set to 1 by CPUID. See "Processor Feature Identification" on page 75 for more information on the CPUID instruction.

If PAT is supported by a processor implementation, it is *always enabled*. The PAT mechanism cannot be disabled by software. Software can effectively avoid using PAT by:

- Not setting PAT bits in page-table entries to 1.
- Not modifying the reset values of the PA fields in the PAT register.

In this case, memory is characterized using the same types that are used by implementations that do not support PAT.

7.7.4 PAT Accesses

In implementations that support the PAT mechanism, all memory accesses that are translated through the paging mechanism use the PAT index bits to specify a PA field in the

PAT register. The memory type stored in the specified PA field is applied to the memory access. The process is summarized as:

1. A virtual address is calculated as a result of a memory access.
2. The virtual address is translated to a physical address using the page-translation mechanism.
3. The PAT, PCD and PWT bits are read from the corresponding page-table entry during the virtual-address to physical-address translation.
4. The PAT, PCD and PWT bits are used to select a PA field from the PAT register.
5. The memory type is read from the appropriate PA field.
6. The memory type is applied to the physical-memory access using the translated physical address.

Page-Translation Table Access. The PAT bit exists only in the PTE (4-K paging) or PDEs (2/4 Mbyte paging). In the remaining upper levels (PML4 PDP, 4K PDEs), only the PWT and PCD bits are used to index into the first 4 entries in the PAT register. The resulting memory type is used for the next lower paging level.

7.7.5 Combined Effect of MTRRs and PAT

The memory types established by the PAT mechanism can be combined with MTRR-established memory types to form an effective memory-type. The combined effect of MTRR and PAT memory types are shown in Table 7-9 on page 232. In the AMD64 architecture, reserved and undefined combinations of MTRR and PAT memory types result in undefined behavior. If the MTRRs are disabled in implementations that support the MTRR mechanism, the default memory type is set to uncacheable (UC).

Table 7-9. Combined Effect of MTRR and PAT Memory Types

PAT Memory Type	MTRR Memory Type	Effective Memory Type
UC	UC, WC, WP, WT, WB	UC
UC-	UC	UC
	WC	WC
	WP, WT, WB	UC[1]
WC	–	WC
WP	UC	UC
	WC	UC[1]
	WP	WP
	WT	UC[1]
	WB	WP
WT	UC	UC
	WC, WP	UC[1]
	WT, WB	WT
WB	UC	UC
	WC	WC
	WP	WP
	WT	WT
	WB	WB

Note:

1. *Previously reserved (undefined) combinations are set to the UC memory type by the AMD64 architecture.*

7.8 Memory-Mapped I/O

Processor implementations can independently direct reads and writes to either system memory or memory-mapped I/O. The method used for directing those memory accesses is implementation dependent. In some implementations, separate system-memory and memory-mapped I/O buses can be provided at the processor interface. In other implementations, system

memory and memory-mapped I/O share common data and address buses, and system logic uses sideband signals from the processor to route accesses appropriately. Refer to AMD data sheets and application notes for more information about particular hardware implementations of the AMD64 architecture.

The I/O range registers (IORRs), and the top-of-memory registers allow system software to specify where memory accesses are directed for a given address range. The MTRR extensions are described in the following section. "IORRs" on page 235 describes the IORRs and "Top of Memory" on page 237 describes the top-of-memory registers. *In implementations that support these features, the default action taken when the features are disabled is to direct memory accesses to memory-mapped I/O.*

7.8.1 Extended Fixed-Range MTRR Type-Field Encodings

The fixed-range MTRRs support extensions to the type-field encodings that allow system software to direct memory accesses to system memory or memory-mapped I/O. The extended MTRR type-field encodings use previously reserved bits 4–3 to specify whether reads and writes to a physical-address range are to system memory or to memory-mapped I/O. The format for this encoding is shown in Figure 7-11. The new bits are:

- *WrMem*—Bit 3. When set to 1, the processor directs write requests for this physical address range to system memory. When cleared to 0, writes are directed to memory-mapped I/O.
- *RdMem*—Bit 4. When set to 1, the processor directs read requests for this physical address range to system memory. When cleared to 0, reads are directed to memory-mapped I/O.

The type subfield (bits 2–0) allows the encodings specified in Table 7-4 on page 217 to be used for memory characterization.

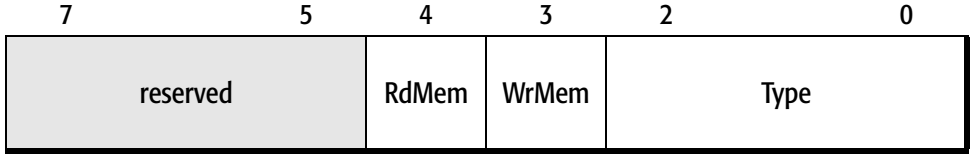


Figure 7-11. Extended MTRR Type-Field Format (Fixed-Range MTRRs)

These extensions are enabled using the following bits in the SYSCFG MSR:

- *MtrrFixDramEn*—Bit 18. When set to 1, RdMem and WrMem attributes are enabled. When cleared to 0, these attributes are disabled. *When disabled, accesses are directed to memory-mapped I/O space.*
- *MtrrFixDramModEn*—Bit 19. When set to 1, software can read and write the RdMem and WrMem bits. When cleared to 0, writes do not modify the RdMem and WrMem bits, and reads return 0.

To use the MTRR extensions, system software must first set MtrrFixDramModEn=1 to allow modification to the RdMem and WrMem bits. After the attribute bits are properly initialized in the fixed-range registers, the extensions can be enabled by setting MtrrFixDramEn=1.

RdMem and WrMem allow the processor to independently direct reads and writes to either system memory or memory-mapped I/O. The RdMem and WrMem controls are particularly useful when shadowing ROM devices located in memory-mapped I/O space. It is often useful to shadow such devices in RAM system memory to improve access performance, but writes into the RAM location can corrupt the shadowed ROM information. The MTRR extensions solve this problem. System software can create the shadow location by setting WrMem=1 and RdMem=0 for the specified memory range and then copy the ROM location into itself. Reads are directed to the memory-mapped ROM, but writes go to the same physical addresses in system memory. After the copy is complete, system software can change the bit values to WrMem=0 and RdMem=1. Now reads are directed to the faster copy located in system memory, and writes are directed to memory-mapped ROM. The ROM responds as it would normally to a write, which is to ignore it.

Not all combinations of RdMem and WrMem are supported for each memory type encoded by bits 2–0. Table 7-10 on page 235 shows the allowable combinations. The behavior of reserved encoding combinations (shown as gray-shaded cells) is undefined and results in unpredictable behavior.

Table 7-10. Extended Fixed-Range MTRR Type Encodings

RdMem	WrMem	Type	Implication or Potential Use
0	0	0 (UC)	UC I/O
		1 (WC)	WC I/O
		4 (WT)	WT I/O
		5 (WP)	WP I/O
		6 (WB)	reserved
0	1	0 (UC)	Used while creating a shadowed ROM
		1 (WC)	
		4 (WT)	reserved
		5 (WP)	
		6 (WB)	
1	0	0 (UC)	Used to access a shadowed ROM
		1 (WC)	reserved
		4 (WT)	WP Memory (Can be used to access shadowed ROM)
		5 (WP)	
		6 (WB)	reserved
1	1	0 (UC)	UC Memory
		1 (WC)	WC Memory
		4 (WT)	WT Memory
		5 (WP)	reserved
		6 (WB)	WB Memory

7.8.2 IORRs

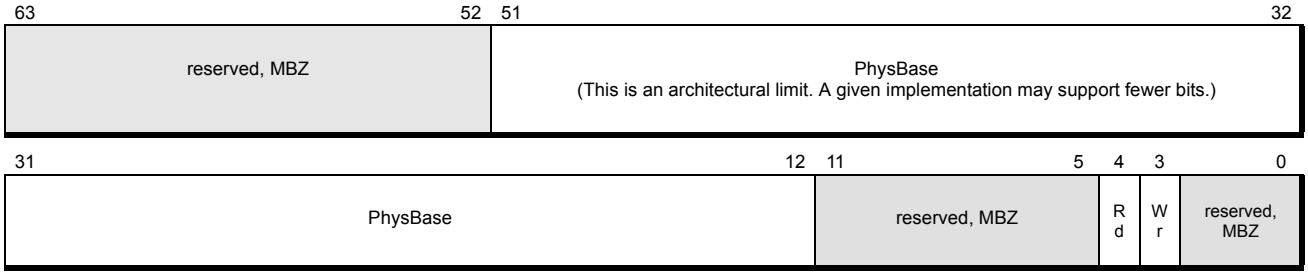
The IORRs operate similarly to the variable-range MTRRs. The IORRs specify whether reads and writes in any physical-address range map to system memory or memory-mapped I/O. Up to two address ranges of varying sizes can be controlled using the IORRs. A pair of IORRs are used to control each address range: IORRBase*n* and IORRMask*n* (*n* is the address-range number from 0 to 1).

Figure 7-12 on page 236 shows the format of the IORRBase*n* registers and Figure 7-13 on page 237 shows the format of the IORRMask*n* registers. The fields within the register pair are read/write.

IORRBase*n* Registers. The fields in these IORRs are:

- *WrMem*—Bit 3. When set to 1, the processor directs write requests for this physical address range to system memory. When cleared to 0, writes are directed to memory-mapped I/O.
- *RdMem*—Bit 4. When set to 1, the processor directs read requests for this physical address range to system memory. When cleared to 0, reads are directed to memory-mapped I/O.
- *Range Physical-Base-Address (PhysBase)*—Bits 51–12. The memory-range base-address in physical-address space. PhysBase is aligned on a 4-Kbyte (or greater) address in the 52-bit physical-address space supported by the AMD64 architecture. PhysBase represents the most-significant 40-address bits of the physical address. Physical-address bits 11–0 are assumed to be 0.

The format of these registers is shown in Figure 7-12.



Bits	Mnemonic	Description	R/W
63-52	reserved	Reserved, Must be Zero	
51-12	PhysBase	Range Physical Base Address	R/W
11-5	reserved	Reserved, Must be Zero	
4	Rd	RdMem Enable	R/W
3	Wr	WrMem Enable	R/W
2-0	reserved	Reserved, Must be Zero	

Figure 7-12. IORRBase*n* Register

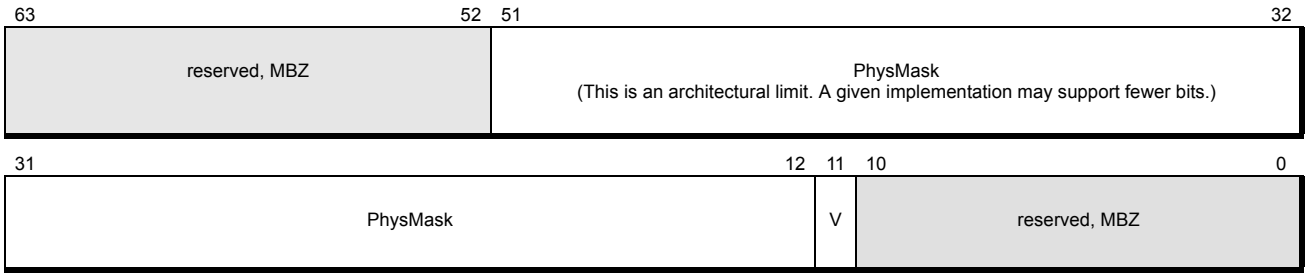
IORRMask*n* Registers. The fields in these IORRs are:

- *Valid (V)*—Bit 11. Indicates that the IORR pair is valid (enabled) when set to 1. When the valid bit is cleared to 0

the register pair is not used for memory-mapped I/O control (disabled).

- *Range Physical-Mask (PhysMask)*—Bits 51–12. The mask value used to specify the memory range. Like PhysBase, PhysMask is aligned on a 4-Kbyte physical-address boundary. Bits 11–0 of PhysMask are assumed to be 0.

The format of these registers is shown in Figure 7-13.



Bits	Mnemonic	Description	R/W
63-52	reserved	Reserved, Must be Zero	
51-12	PhysMask	Range Physical Mask	R/W
11	V	I/O Register Pair Enable (Valid)	R/W
10-0	reserved	Reserved, Must be Zero	

Figure 7-13. IORRMask*n* Register

The operation of the PhysMask and PhysBase fields is identical to that of the variable-range MTRRs. See page 222 for a description of this operation.

7.8.3 IORR Overlapping

The use of overlapping IORRs is not recommended. If overlapping IORRs are specified, the resulting behavior is implementation-dependent.

7.8.4 Top of Memory

The *top-of-memory* registers, TOP_MEM and TOP_MEM2, allow system software to specify physical addresses ranges as memory-mapped I/O locations. Processor implementations can direct accesses to memory-mapped I/O differently than system I/O, and the precise method depends on the implementation. System software specifies memory-mapped I/O regions by writing an address into each of the top-of-memory registers. The memory regions specified by the TOP_MEM registers are aligned on 8-Mbyte boundaries as follows:

- Memory accesses from physical address 0 to one less than the value in TOP_MEM are directed to system memory.

- Memory accesses from the physical address specified in TOP_MEM to FFFF_FFFFh are directed to memory-mapped I/O.
- Memory accesses from physical address 1_0000_0000h to one less than the value in TOP_MEM2 are directed to system memory.
- Memory accesses from the physical address specified in TOP_MEM2 to the maximum physical address supported by the system are directed to memory-mapped I/O.

Figure 7-14 shows how the top-of-memory registers organize memory into separate system-memory and memory-mapped I/O regions.

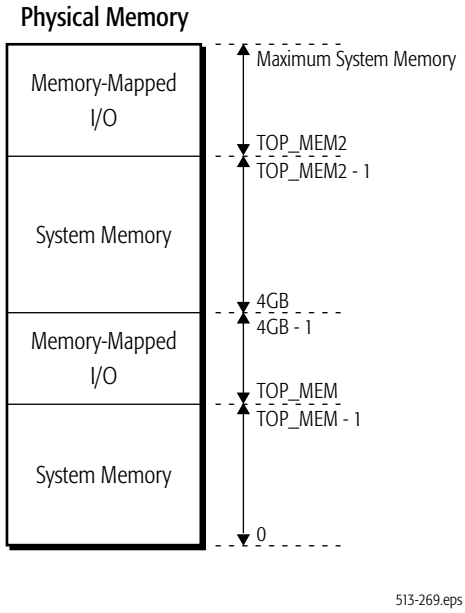


Figure 7-14. Memory Organization Using Top-of-Memory Registers

Figure 7-15 on page 239 shows the format of the TOP_MEM and TOP_MEM2 registers. Bits 51–23 specify an 8-Mbyte aligned physical address. All remaining bits are reserved and ignored by the processor. System software should clear those bits to zero to maintain compatibility with possible future extensions to the registers. The TOP_MEM registers are model-specific registers. See “Memory-Typing MSRs” on page 438 for information on the MSR address and reset values for these registers.

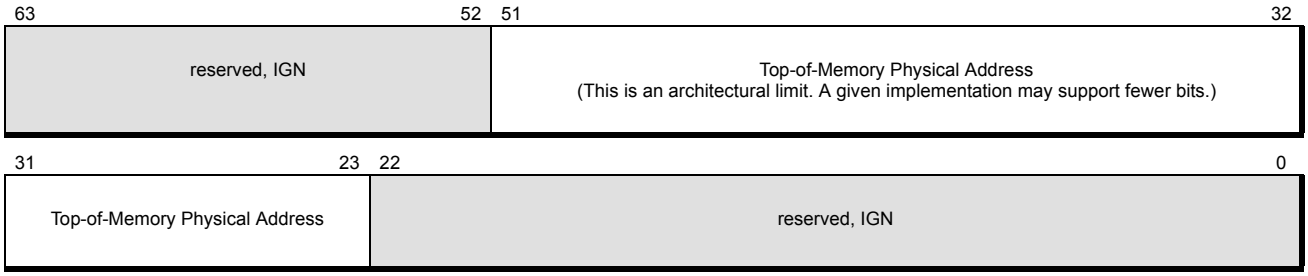


Figure 7-15. Top-of-Memory Registers (TOP_MEM, TOP_MEM2)

The TOP_MEM register is enabled by setting the MtrrVarDramEn bit in the SYSCFG MSR (bit 20) to 1. The TOP_MEM2 register is enabled by setting the MtrrTom2En bit in the SYSCFG MSR (bit 21) to 1. The registers are disabled when their respective enable bits are cleared to 0. When the top-of-memory registers are disabled, memory accesses default to memory-mapped I/O space.

8 Exceptions and Interrupts

8.1 Overview

Exceptions and interrupts force control transfers from the currently-executing program to a system-software service routine that handles the interrupting event. These routines are referred to as *exception handler*s and *interrupt handlers*, or collectively as *event handlers*. Typically, interrupt events can be handled by the service routine transparently to the interrupted program. During the control transfer to the service routine, the processor stops executing the interrupted program and saves its return pointer. The system-software service routine that handles the exception or interrupt is responsible for saving the state of the interrupted program. This allows the processor to restart the interrupted program after system software has handled the event.

When an exception or interrupt occurs, the processor uses the interrupt-vector number as an index into the interrupt-descriptor table (IDT). An IDT is used in all processor operating modes, including real mode (also called real-address mode), protected mode, and long mode.

Exceptions and interrupts come from three general sources:

- *Exceptions* occur as a result of software execution errors or other internal-processor errors. Exceptions also occur during non-error situations, such as program single stepping or address-breakpoint detection. Exceptions are considered *synchronous* events because they are a direct result of executing the interrupted instruction.
- *Software interrupts* occur as a result of executing interrupt instructions. Unlike exceptions and external interrupts, software interrupts allow intentional triggering of the interrupt-handling mechanism. Like exceptions, software interrupts are synchronous events.
- *External interrupts* are generated by system logic in response to an error or some other event outside the processor. They are reported over the processor bus using external signalling. External interrupts are *asynchronous* events that occur independently of the interrupted instruction.

Throughout this section, the term *masking* can refer to either disabling or delaying an interrupt. For example, masking external interrupts *delays* the interrupt, with the processor holding the interrupt as pending until it is unmasked. With floating-point exceptions (128-bit media and x87), masking *prevents* an interrupt from occurring and causes the processor to perform a default operation on the exception condition.

8.2 General Characteristics

Exceptions and interrupts have several different characteristics that depend on how events are reported and the implications for program restart.

8.2.1 Precision

Precision describes how the exception is related to the interrupted program:

- *Precise* exceptions are reported on a predictable instruction boundary. This boundary is generally the first instruction that has not completed when the event occurs. All previous instructions (in program order) are allowed to complete before transferring control to the event handler. The pointer to the instruction boundary is saved automatically by the processor. When the event handler completes execution, it returns to the interrupted program and restarts execution at the interrupted-instruction boundary.
- *Imprecise* exceptions are not guaranteed to be reported on a predictable instruction boundary. The boundary can be any instruction that has not completed when the interrupt event occurs. Imprecise events can be considered asynchronous, because the source of the interrupt is not necessarily related to the interrupted instruction. Imprecise exception and interrupt handlers typically collect machine-state information related to the interrupting event for reporting through system-diagnostic software. The interrupted program is not restartable.

8.2.2 Instruction Restart

As mentioned above, precise exceptions are reported on an instruction boundary. The instruction boundary can be reported in one of two locations:

- Most exceptions report the boundary *before* the instruction causing the exception. In this case, all previous instructions (in program order) are allowed to complete, but the

interrupted instruction is not. *No program state is updated as a result of partially executing an interrupted instruction.*

- Some exceptions report the boundary *after* the instruction causing the exception. In this case, all previous instructions—including the one executing when the exception occurred—are allowed to complete.

  Program state can be updated when the reported boundary is after the instruction causing the exception. This is particularly true when the event occurs as a result of a task switch. In this case, the general registers, segment-selector registers, page-base address register, and LDTR are all updated by the hardware task-switch mechanism. The event handler cannot rely on the state of those registers when it begins execution and must be careful in validating the state of the segment-selector registers before restarting the interrupted task. This is not an issue in long mode, however, because the hardware task-switch mechanism is disabled in long mode.

8.2.3 Types of Exceptions

There are three types of exceptions, depending on whether they are precise and how they affect program restart:

- *Faults* are precise exceptions reported on the boundary before the instruction causing the exception. Generally, faults are caused by an error condition involving the faulted instruction. Any machine-state changes caused by the faulting instruction are discarded so that the instruction can be restarted. The saved rIP points to the faulting instruction.
- *Traps* are precise exceptions reported on the boundary following the instruction causing the exception. The trapped instruction is completed by the processor and all state changes are saved. The saved rIP points to the instruction following the faulting instruction.
- *Aborts* are imprecise exceptions. Because they are imprecise, aborts typically do not allow reliable program restart.

8.2.4 Masking External Interrupts

General Masking Capabilities. Software can *mask* the occurrence of certain exceptions and interrupts. Masking can delay or even prevent triggering of the exception-handling or interrupt-handling mechanism when an interrupt-event occurs. External interrupts are classified as maskable or nonmaskable:

- *Maskable interrupts* trigger the interrupt-handling mechanism only when RFLAGS.IF=1. Otherwise they are

held pending for as long as the RFLAGS.IF bit is cleared to 0.

- *Nonmaskable interrupts* (NMI) are unaffected by the value of the rFLAGS.IF bit. However, the occurrence of an NMI masks further NMIs until an IRET instruction is executed.

Masking During Stack Switches. The processor delays recognition of maskable external interrupts and debug exceptions during certain instruction sequences that are often used by software to switch stacks. The typical programming sequence used to switch stacks is:

1. Load a stack selector into the SS register.
2. Load a stack offset into the ESP register.

If an interrupting event occurs after the selector is loaded but before the stack offset is loaded, the interrupted-program stack pointer is invalid during execution of the interrupt handler.

To prevent interrupts from causing stack-pointer problems, the processor does not allow external interrupts or debug exceptions to occur until the instruction immediately following the MOV SS or POP SS instruction completes execution.

The recommended method of performing this sequence is to use the LSS instruction. LSS loads both SS and ESP, and the instruction inhibits interrupts until both registers are updated successfully.

8.2.5 Masking Floating-Point and Media Instructions

x87 floating-point exceptions can be masked and reported later using bits in the x87 floating-point status register (FSW) and the x87 floating-point control register (FCW). The floating-point exception-pending exception is used for unmasked x87 floating-point exceptions (see "#MF—x87 Floating-Point Exception-Pending (Vector 16)" on page 260).

The SIMD floating-point exception is used for unmasked 128-bit media floating-point exceptions (see "#XF—SIMD Floating-Point Exception (Vector 19)" on page 263). 128-bit media floating-point exceptions are masked using the MXCSR register. The exception mechanism is not triggered when these exceptions are masked. Instead, the processor handles the exceptions in a default manner.

8.2.6 Disabling Exceptions

Disabling an exception prevents the exception condition from being recognized, unlike masking an exception which prevents

triggering the exception mechanism after the exception is recognized. Some exceptions can be disabled by system software running at CPL=0, using bits in the CR0 register or CR4 register:

- Alignment-check exception (see "#AC—Alignment-Check Exception (Vector 17)" on page 261).
- Device-not-available exception (see "#NM—Device-Not-Available Exception (Vector 7)" on page 252).
- Machine-check exception (see "#MC—Machine-Check Exception (Vector 18)" on page 262).

The debug-exception mechanism provides control over when specific breakpoints are enabled and disabled. See "Breakpoints" on page 392 for more information on how breakpoint controls are used for triggering the debug-exception mechanism.

8.3 Vectors

Specific exception and interrupt sources are assigned a fixed vector-identification number (also called an "interrupt vector", or simply "vector"). The interrupt vector is used by the interrupt-handling mechanism to locate the system-software service routine assigned to the exception or interrupt. Up to 256 unique interrupt vectors are available. The first 32 vectors are reserved for predefined exception and interrupt conditions. Software-interrupt sources can trigger an interrupt using any available interrupt vector.

Table 8-1 on page 246 lists the supported interrupt-vector numbers, the corresponding exception or interrupt name, the mnemonic, the source of the interrupt event, and a summary of the possible causes.

Table 8-1. Interrupt-Vector Source and Cause

Vector	Exception/Interrupt	Mnemonic	Cause
0	Divide-By-Zero-Error	#DE	DIV, IDIV instructions
1	Debug	#DB	Instruction accesses and data accesses
2	Non-Maskable-Interrupt	—	External NMI signal
3	Breakpoint	#BP	INT3 instruction
4	Overflow	#OF	INTO instruction
5	Bound-Range	#BR	BOUND instruction
6	Invalid-Opcode	#UD	Invalid instructions
7	Device-Not-Available	#NM	x87 instructions
8	Double-Fault	#DF	Exception during the handling of another exception or interrupt
9	Coprocessor-Segment-Overrun	—	Unsupported (reserved)
10	Invalid-TSS	#TS	Task-state segment access and task switch
11	Segment-Not-Present	#NP	Segment register loads
12	Stack	#SS	SS register loads and stack references
13	General-Protection	#GP	Memory accesses and protection checks
14	Page-Fault	#PF	Memory accesses when paging enabled
15	reserved	—	
16	x87 Floating-Point Exception-Pending	#MF	x87 floating-point instructions
17	Alignment-Check	#AC	Misaligned memory accesses
18	Machine-Check	#MC	Model specific
19	SIMD Floating-Point	#XF	128-bit media floating-point instructions
20—31	reserved	—	
0—255	External Interrupts (Maskable)	—	External interrupts
0—255	Software Interrupts	—	INT*n* instruction

Table 8-2 on page 247 shows how each interrupt vector is classified. Reserved interrupt vectors are indicated by the gray-shaded rows.

Table 8-2. Interrupt-Vector Classification

Vector	Interrupt (Exception)	Type	Precise	Class[2]
0	Divide-By-Zero-Error	Fault	yes	Contributory
1	Debug	Fault or Trap		Benign
2	Non-Maskable-Interrupt	—	—	
3	Breakpoint	Trap	yes	
4	Overflow			
5	Bound-Range	Fault		
6	Invalid-Opcode			
7	Device-Not-Available			
8	Double-Fault	Abort	no	
9	Coprocessor-Segment-Overrun			
10	Invalid-TSS	Fault	yes	Contributory
11	Segment-Not-Present			
12	Stack			
13	General-Protection			
14	Page-Fault			Benign or Contributory
15	reserved			
16	x87 Floating-Point Exception-Pending	Fault	no	Benign
17	Alignment-Check		yes	
18	Machine-Check	Abort	no	
19	SIMD Floating-Point	Fault	yes	

Note:

1. *External interrupts are not classified by type or whether or not they are precise.*
2. *See "#DF—Double-Fault Exception (Vector 8)" on page 252 for a definition of benign and contributory classes.*

Table 8-2. Interrupt-Vector Classification (continued)

Vector	Interrupt (Exception)	Type	Precise	Class[2]
20—31	reserved			
0—255	External Interrupts (Maskable)	—[1]	—[1]	Benign
0—255	Software Interrupts			

Note:

1. *External interrupts are not classified by type or whether or not they are precise.*
2. *See "#DF—Double-Fault Exception (Vector 8)" on page 252 for a definition of benign and contributory classes.*

The following sections describe each interrupt in detail. The format of the error code reported by each interrupt is described in "Error Codes" on page 265.

8.3.1 #DE—Divide-By-Zero-Error Exception (Vector 0)

A #DE exception occurs when the denominator of a DIV instruction or an IDIV instruction is 0. A #DE also occurs if the result is too large to be represented in the destination.

#DE cannot be disabled.

Error Code Returned. None.

Program Restart. #DE is a fault-type exception. The saved instruction pointer points to the instruction that caused the #DE.

8.3.2 #DB—Debug Exception (Vector 1)

When the debug-exception mechanism is enabled, a #DB exception can occur under any of the following circumstances:

- Instruction execution.
- Instruction single stepping.
- Data read.
- Data write.
- I/O read.
- I/O write.
- Task switch.
- Debug-register access, or *general detect fault* (debug register access when DR7.GD=1).
- Executing the INT1 instruction (opcode 0F1h).

#DB conditions are enabled and disabled using the debug-control register, DR7 and RFLAGS.TF. Each #DB condition is described in more detail in "Breakpoints" on page 392.

Error Code Returned. None. #DB information is returned in the debug-status register, DR6.

Program Restart. #DB can be either a fault-type or trap-type exception. In the following cases, the saved instruction pointer points to the instruction that caused the #DB:

- Instruction execution.
- Invalid debug-register access, or *general detect.*

In all other cases, the instruction that caused the #DB is completed, and the saved instruction pointer points to the instruction after the one that caused the #DB.

The RFLAGS.RF bit can be used to restart an instruction following an instruction breakpoint resulting in a #DB. In most cases, the processor clears RFLAGS.RF to 0 after every instruction is successfully executed. However, in the case of the IRET, JMP, CALL, and INT*n* (through a task gate) instructions, RFLAGS.RF is not cleared to 0 until the *next* instruction successfully executes.

When a non-debug exception occurs (or when a string instruction is interrupted), the processor normally sets RFLAGS.RF to 1 in the RFLAGS *image* that is pushed on the interrupt stack. A subsequent IRET back to the interrupted program pops the RFLAGS image off the stack and into the RFLAGS register, with RFLAGS.RF=1. The interrupted instruction executes without causing an instruction breakpoint, after which the processor clears RFLAGS.RF to 0.

However, when a #DB exception occurs, the processor clears RFLAGS.RF to 0 in the RFLAGS image that is pushed on the interrupt stack. The #DB handler has two options:

- Disable the instruction breakpoint completely.
- Set RFLAGS.RF to 1 in the interrupt-stack rFLAGS image. The instruction breakpoint condition is ignored immediately after the IRET, but reoccurs if the instruction address is accessed later, as can occur in a program loop.

8.3.3 NMI—Non-Maskable-Interrupt Exception (Vector 2)

An NMI exception occurs as a result of system logic signalling a non-maskable interrupt to the processor.

Error Code Returned. None.

Program Restart. NMI is an interrupt. The processor recognizes an NMI at an instruction boundary. The saved instruction pointer points to the instruction immediately following the boundary where the NMI was recognized.

Masking. NMI cannot be masked. However, when an NMI is recognized by the processor, recognition of subsequent NMIs are disabled until an IRET instruction is executed.

8.3.4 #BP—Breakpoint Exception (Vector 3)

A #BP exception occurs when an INT3 instruction is executed. The INT3 is normally used by debug software to set instruction breakpoints by replacing instruction-opcode bytes with the INT3 opcode.

#BP cannot be disabled.

Error Code Returned. None.

Program Restart. #BP is a trap-type exception. The saved instruction pointer points to the byte after the INT3 instruction. This location can be the start of the next instruction. However, if the INT3 is used to replace the first opcode bytes of an instruction, the restart location is likely to be in the middle of an instruction. In the latter case, the debug software must replace the INT3 byte with the correct instruction byte. The saved RIP instruction pointer must then be decremented by one before returning to the interrupted program. This allows the program to be restarted correctly on the interrupted-instruction boundary.

8.3.5 #OF—Overflow Exception (Vector 4)

An #OF exception occurs as a result of executing an INTO instruction while the overflow bit in RFLAGS is set to 1 (RFLAGS.OF=1).

#OF cannot be disabled.

Error Code Returned. None.

Program Restart. #OF is a trap-type exception. The saved instruction pointer points to the instruction following the INTO instruction that caused the #OF.

8.3.6 #BR—Bound-Range Exception (Vector 5)

A #BR exception can occur as a result of executing the BOUND instruction. The BOUND instruction compares an array index (first operand) with the lower bounds and upper bounds of an array (second operand). If the array index is not within the array boundary, the #BR occurs.

#BR cannot be disabled.

Error Code Returned. None.

Program Restart. #BR is a fault-type exception. The saved instruction pointer points to the BOUND instruction that caused the #BR.

8.3.7 #UD—Invalid-Opcode Exception (Vector 6)

A #UD exception occurs when an attempt is made to execute an invalid or undefined opcode. The validity of an opcode often depends on the processor operating mode. A #UD occurs under the following conditions:

- Execution of any reserved or undefined opcode in any mode.
- Execution of the UD2 instruction.
- Use of the LOCK prefix on an instruction that cannot be locked.
- Use of the LOCK prefix on a lockable instruction with a non-memory target location.
- Execution of an instruction with an invalid-operand type.
- Execution of the SYSENTER or SYSEXIT instructions in long mode.
- Execution of any of the following instructions in 64-bit mode: AAA, AAD, AAM, AAS, BOUND, CALL (opcode 9A), DAA, DAS, DEC, INC, INTO, JMP (opcode EA), LAHF, LDS, LES, POP (DS, ES, SS), POPA, PUSH (CS, DS, ES, SS), PUSHA, SAHF, SALC.
- Execution of the ARPL, LAR, LLDT, LSL, LTR, SLDT, STR, VERR, or VERW instructions when protected mode is not enabled, or when virtual-8086 mode is enabled.
- Execution of any 128-bit media instruction (uses XMM registers), or 64-bit media instruction (uses MMX™ registers) when CR0.NM = 1.
- Execution of any 128-bit media floating-point instruction (uses XMM registers) that causes a numeric exception when CR4.OSXMMEXCPT = 0.
- Use of the DR4 or DR5 debug registers when CR4.DE = 1.

- Execution of RSM when not in SMM mode.

See the specific instruction description (in the other volumes) for additional information on invalid conditions.

#UD cannot be disabled.

Error Code Returned. None.

Program Restart. #UD is a fault-type exception. The saved instruction pointer points to the instruction that caused the #UD.

8.3.8 #NM—Device-Not-Available Exception (Vector 7)

A #NM exception occurs under any of the following conditions:

- An FWAIT/WAIT instruction is executed when CR0.MP=1 and CR0.TS=1.
- Any x87 instruction other than FWAIT is executed when CR0.EM=1.
- Any x87 instruction is executed when CR0.TS=1. The CR0.MP bit controls whether the FWAIT/WAIT instruction causes an #NM exception when TS=1.
- Any 128-bit or 64-bit media instruction when CR0.TS=1.

#NM can be enabled or disabled under the control of the CR0.MP, CR0.EM, and CR0.TS bits as described above. See "CR0 Register" on page 53 for more information on the CR0 bits used to control the #NM exception.

Error Code Returned. None.

Program Restart. #NM is a fault-type exception. The saved instruction pointer points to the instruction that caused the #NM.

8.3.9 #DF—Double-Fault Exception (Vector 8)

A #DF exception can occur when a second exception occurs during the handling of a prior (first) exception or interrupt handler.

Usually, the first and second exceptions can be handled sequentially without resulting in a #DF. In this case, the first exception is considered *benign*, as it does not harm the ability of the processor to handle the second exception.

In some cases, however, the first exception adversely affects the ability of the processor to handle the second exception. These

exceptions contribute to the occurrence of a #DF, and are called *contributory* exceptions. If a contributory exception is followed by another contributory exception, a double-fault exception occurs. Likewise, if a page fault is followed by another page fault or a contributory exception, a double-fault exception occurs.

Table 8-3 shows the conditions under which a #DF occurs. Page faults are either benign or contributory, and are listed separately. See the "Class" column in Table 8-2 on page 247 for information on whether an exception is benign or contributory.

Table 8-3. Double-Fault Exception Conditions

First Interrupting Event	Second Interrupting Event
Contributory Exceptions • Divide-By-Zero-Error Exception • Invalid-TSS Exception • Segment-Not-Present Exception • Stack Exception • General-Protection Exception	Divide-By-Zero-Error Exception Invalid-TSS Exception Segment-Not-Present Exception Stack Exception General-Protection Exception
Page Fault Exception	Page Fault Exception Divide-By-Zero-Error Exception Invalid-TSS Exception Segment-Not-Present Exception Stack Exception General-Protection Exception

If a third interrupting event occurs while transferring control to the #DF handler, the processor shuts down. Only an NMI, RESET, or INIT can restart the processor in this case. However, if the processor shuts down as it is executing an NMI handler, the processor can only be restarted with RESET or INIT.

#DF cannot be disabled.

Error Code Returned. Zero.

Program Restart. #DF is an abort-type exception. The saved instruction pointer is undefined, and the program cannot be restarted.

8.3.10 Coprocessor-Segment-Overrun Exception (Vector 9)

This interrupt vector is reserved. It is for a discontinued exception originally used by processors that supported external x87-instruction coprocessors. On those processors, the exception condition is caused by an invalid-segment or invalid-page access on an x87-instruction coprocessor-instruction operand. On current processors, this condition causes a general-protection exception to occur.

Error Code Returned. Not applicable.

Program Restart. Not applicable.

8.3.11 #TS—Invalid-TSS Exception (Vector 10)

A #TS exception occurs when an invalid reference is made to a segment selector as part of a task switch. A #TS also occurs during a privilege-changing control transfer (through a call gate or an interrupt gate), if a reference is made to an invalid stack-segment selector located in the TSS. Table 8-4 on page 255 lists the conditions under which a #TS occurs and the error code returned by the exception mechanism.

#TS cannot be disabled.

Error Code Returned. See Table 8-4 on page 255 for a list of error codes returned by the #TS exception.

Program Restart. #TS is a fault-type exception. If the exception occurs before loading the segment selectors from the TSS, the saved instruction pointer points to the instruction that caused the #TS. However, most #TS conditions occur due to errors with the loaded segment selectors. When an error is found with a segment selector, the hardware task-switch mechanism completes loading the new task state from the TSS, and then triggers the #TS exception mechanism. In this case, the saved instruction pointer points to the first instruction in the new task.

In long mode, a #TS cannot be caused by a task switch, because the hardware task-switch mechanism is disabled. A #TS occurs only as a result of a control transfer through a gate descriptor that results in an invalid stack-segment reference using an SS selector in the TSS. In this case, the saved instruction pointer always points to the control-transfer instruction that caused the #TS.

Table 8-4. Invalid-TSS Exception Conditions

Selector Reference	Error Condition	Error Code
Task-State Segment	TSS limit check on a task switch	TSS Selector Index
	TSS limit check on an inner-level stack pointer	
LDT Segment	LDT does not point to GDT	LDT Selector Index
	LDT reference outside GDT	
	GDT entry is not an LDT descriptor	
	LDT descriptor is not present	
Code Segment	CS reference outside GDT or LDT	CS Selector Index
	Privilege check (conforming DPL > CPL)	
	Privilege check (non-conforming DPL ≠ CPL)	
	Type check (CS not executable)	
Data Segment	Data segment reference outside GDT or LDT	DS, ES, FS or GS Selector Index
	Type check (data segment not readable)	
Stack Segment	SS reference outside GDT or LDT	SS Selector Index
	Privilege check (stack segment descriptor DPL ≠ CPL)	
	Privilege check (stack segment selector RPL ≠ CPL)	
	Type check (stack segment not writable)	

8.3.12 #NP—Segment-Not-Present Exception (Vector 11)

An #NP occurs when an attempt is made to load a segment or gate with a clear present bit, as described in the following situations:

- Using the MOV, POP, LDS, LES, LFS, or LGS instructions to load a segment selector (DS, ES, FS, and GS) that references a segment descriptor containing a clear present bit (descriptor.P=0).
- Far transfer to a CS that is not present.
- Referencing a gate descriptor containing a clear present bit.
- Referencing a TSS descriptor containing a clear present bit. This includes attempts to load the TSS descriptor using the LTR instruction.

- Attempting to load a descriptor containing a clear present bit into the LDTR using the LLDT instruction.
- Loading a segment selector (CS, DS, ES, FS, or GS) as part of a task switch, with the segment descriptor referenced by the segment selector having a clear present bit. In long mode, an #NP cannot be caused by a task switch, because the hardware task-switch mechanism is disabled.

When loading a stack-segment selector (SS) that references a descriptor with a clear present bit, a stack exception (#SS) occurs. For information on the #SS exception, see the next section, "#SS—Stack Exception (Vector 12)."

#NP cannot be disabled.

Error Code Returned. The segment-selector index of the segment descriptor causing the #NP exception.

Program Restart. #NP is a fault-type exception. In most cases, the saved instruction pointer points to the instruction that loaded the segment selector resulting in the #NP. See "Exceptions During a Task Switch" on page 265 for a description of the consequences when this exception occurs during a task switch.

8.3.13 #SS—Stack Exception (Vector 12)

An #SS exception can occur in the following situations:

- Implied stack references in which the stack address is not in canonical form. Implied stack references include all push and pop instructions, and any instruction using RSP or RBP as a base register.
- Attempting to load a stack-segment selector that references a segment descriptor containing a clear present bit (descriptor.P=0).
- Any stack access that fails the stack-limit check.

#SS cannot be disabled.

Error Code Returned. The error code depends on the cause of the #SS, as shown in Table 8-5 on page 257:

Table 8-5. Stack Exception Error Codes

Stack Exception Cause	Error Code
Stack-segment descriptor present bit is clear	SS Selector Index
Stack-limit violation	0
Stack reference using a non-canonical address	0

Program Restart. #SS is a fault-type exception. In most cases, the saved instruction pointer points to the instruction that caused the #SS. See "Exceptions During a Task Switch" on page 265 for a description of the consequences when this exception occurs during a task switch.

8.3.14 #GP—General-Protection Exception (Vector 13)

Table 8-6 on page 258 describes the general situations that can cause a #GP exception. The table is not an exhaustive, detailed list of #GP conditions, but rather a guide to the situations that can cause a #GP. If an invalid use of an AMD64 architectural feature results in a #GP, the specific cause of the exception is described in detail in the section describing the architectural feature.

#GP cannot be disabled.

Error Code Returned. As shown in Table 8-6, a selector index is reported as the error code if the #GP is due to a segment-descriptor access. In all other cases, an error code of 0 is returned.

Program Restart. #GP is a fault-type exception. In most cases, the saved instruction pointer points to the instruction that caused the #GP. See "Exceptions During a Task Switch" on page 265 for a description of the consequences when this exception occurs during a task switch.

Table 8-6. General-Protection Exception Conditions

Error Condition	Error Code
Any segment privilege-check violation.	Selector Index
Any segment limit-check violation, except for the SS selector.	
Any segment type-check violation, including read, write, and execute violations.	
Accessing a long mode gate-descriptor containing a non-canonical address.	
Accessing a 64-bit mode system-descriptor containing a non-canonical address.	
Loading a non-canonical base address into the GDTR or IDTR in 64-bit mode.	
Loading the CS or SS register with a null segment selector.	
Accessing a gate descriptor containing a null segment selector.	
Accessing memory with a null DS, ES, FS, or GS data-segment selector.	
In long mode, referencing any 16-bit gate descriptor.	
In long mode, referencing a call gate or interrupt gate that does not point to a 64-bit code segment.	
In long mode, referencing any system descriptor that does not have the second type field loaded with 0.	
Attempting a control transfer to a TSS or task gate in long mode.	
Referencing an LDT descriptor or TSS descriptor located in an LDT.	
Control transfer to a busy TSS (except IRET).	
Using the LTR instruction to load a busy TSS descriptor into the TR register.	
Referencing virtual memory with a non-canonical address (except stack references).	0
Any long-mode consistency-check violation.	
Using WRMSR to write a non-canonical address into the FS.base or GS.base register.	
Using WRMSR to write a non-canonical address into the KernelGSbase register.	
Using WRMSR to write a non-canonical address into the CSTAR or LSTAR register.	
Using WRMSR to write read-only MSRs.	
Writing invalid type encodings into the MTRRs or the PAT MSR.	
Writing a 1 into any register field that is *reserved, must be zero* (MBZ).	
Using a 128-bit media instruction requiring 16-byte alignment with a memory operand not aligned on a 16-byte boundary.	

Table 8-6. General-Protection Exception Conditions (continued)

Error Condition	Error Code
Executing privileged instructions when CPL > 0.	0
Enabling paging when protected mode is disabled.	
Setting CR0.NW=1 while CR0.CD=0.	

8.3.15 #PF—Page-Fault Exception (Vector 14)

A #PF exception can occur during a memory access in any of the following situations:

- A page-translation-table entry or physical page involved in translating the memory access is not present in physical memory. This is indicated by a cleared present bit (P=0) in the translation-table entry.
- An attempt is made by the processor to load the instruction TLB with a translation for a non-executable page.
- The memory access fails the paging-protection checks (user/supervisor, read/write, or both).
- A reserved bit in one of the page-translation-table entries is set to 1. A #PF occurs for this reason only when CR4.PSE=1 *or* CR4.PAE=1.

#PF cannot be disabled.

CR2 Register. The virtual (linear) address that caused the #PF is stored in the CR2 register. Figure 8-1 shows the legacy 32-bit CR2 register. The CR2 register in the AMD64 architecture is 64 bits long, as shown in Figure 8-1. In AMD64 implementations, only the low-order 32 bits of CR2 can be used in legacy mode. The processor clears the high-order 32 bits of CR2 when it is written in legacy mode either by software or during a page fault.

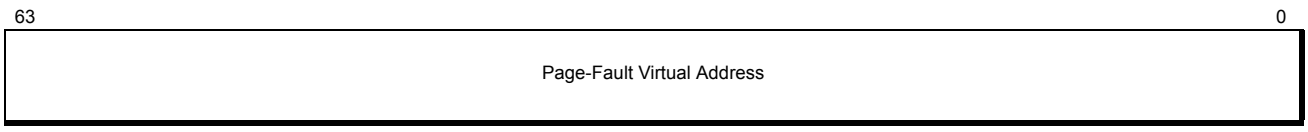


Figure 8-1. Control Register 2 (CR2)

Error Code Returned. The page-fault error code is pushed onto the page-fault exception-handler stack. See “Page-Fault Error Code” on page 266 for a description of this error code.

Program Restart. #PF is a fault-type exception. In most cases, the saved instruction pointer points to the instruction that caused the #PF. See "Exceptions During a Task Switch" on page 265 for a description of what can happen if this exception occurs during a task switch.

8.3.16 #MF—x87 Floating-Point Exception-Pending (Vector 16)

The #MF exception is used to handle unmasked x87 floating-point exceptions. An #MF occurs when *all* of the following conditions are true:

- CR0.NE=1.
- An unmasked x87 floating-point exception is pending. This is indicated by an exception bit in the x87 floating-point status-word register being set to 1
- The corresponding mask bit in the x87 floating-point control-word register is cleared to 0.
- The FWAIT/WAIT instruction or any waiting floating-point instruction is executed.

If there is an exception mask bit (in the FPU control word) set, the exception is not reported. Instead, the x87-instruction unit responds in a default manner and execution proceeds normally.

The x87 floating-point exceptions reported by the #MF exception are (including mnemonics):

- IE—Invalid-operation exception (also called #I), which is either:
  - IE alone—Invalid arithmetic-operand exception (also called #IA), or
  - SF and IE together—x87 Stack-fault exception (also called #IS).
- DE—Denormalized-operand exception (also called #D).
- ZE—Zero-divide exception (also called #Z).
- OE—Overflow exception (also called #O).
- UE—Underflow exception (also called #U).
- PE—Precision exception (also called #P or inexact-result exception).

Error Code Returned. None. Exception information is provided by the x87 status-word register. See "x87 Floating-Point Programming" in Volume 1 for more information on using this register.

Program Restart. #MF is a fault-type exception. The #MF exception is not precise, because multiple instructions and exceptions can occur before the #MF handler is invoked. Also, the saved instruction pointer does not point to the instruction that caused the exception resulting in the #MF. Instead, the saved instruction pointer points to the x87 floating-point instruction or FWAIT/WAIT instruction that is about to be executed when the #MF occurs. The address of the *last instruction* that caused an x87 floating-point exception is in the x87 instruction-pointer register. See "x87 Floating-Point Programming" in Volume 1 for information on accessing this register.

Masking. Each type of x87 floating-point exception can be masked by setting the appropriate bits in the x87 control-word register. See "x87 Floating-Point Programming" in Volume 1 for more information on using this register.

#MF can also be disabled by clearing the CR0.NE bit to 0. See "Numeric Error (NE) Bit" on page 55 for more information on using this bit.

8.3.17 #AC—Alignment-Check Exception (Vector 17)

An #AC exception occurs when an unaligned-memory data reference is performed while alignment checking is enabled.

After a processor reset, #AC exceptions are disabled. Software enables the #AC exception by setting the following register bits:

- CR0.AM=1.
- RFLAGS.AC=1.

When the above register bits are set, an #AC can occur only when CPL=3. #AC never occurs when CPL < 3.

Table 8-7 on page 262 lists the data types and the alignment boundary required to *avoid* an #AC exception when the mechanism is enabled.

Table 8-7. Data-Type Alignment

Supported Data Type	Required Alignment (Byte Boundary)
Word	2
Doubleword	4
Quadword	8
Bit string	2, 4 or 8 (depends on operand size)
128-bit media	16
64-bit media	8
Segment selector	2
32-bit near pointer	4
32-bit far pointer	2
48-bit far pointer	4
x87 Floating-point single-precision	4
x87 Floating-point double-precision	8
x87 Floating-point extended-precision	8
x87 Floating-point save areas	2 or 4 (depends on operand size)

Error Code Returned. Zero.

Program Restart. #AC is a fault-type exception. The saved instruction pointer points to the instruction that caused the #AC.

8.3.18 #MC—Machine-Check Exception (Vector 18)

The #MC exception is model specific. Processor implementations are not required to support the #MC exception, and those implementations that do support #MC can vary in how the #MC exception mechanism works.

The exception is enabled by setting CR4.MCE to 1. The machine-check architecture can include model-specific masking for controlling the reporting of some errors. Refer to Chapter 9, “Machine Check Mechanism,” for more information.

Error Code Returned. None. Error information is provided by model-specific registers (MSRs) defined by the machine-check architecture.

Program Restart. #MC is an abort-type exception. There is no reliable way to restart the program. If the EIPV flag (EIP valid) is set in the MCG_Status MSR, the saved CS and rIP point to the instruction that caused the error. If EIP is clear, the CS:rIP of the instruction causing the failure is not known or the machine check is not related to a specific instruction.

8.3.19 #XF—SIMD Floating-Point Exception (Vector 19)

The #XF exception is used to handle unmasked 128-bit media floating-point exceptions. A #XF exception occurs when all of the following conditions are true:

- A 128-bit media floating-point exception occurs. The exception causes the processor to set the appropriate exception-status bit in the MXCSR register to 1.
- The exception-mask bit in the MXCSR that corresponds to the 128-bit media floating-point exception is clear (=0).
- CR4.OSXMMEXCPT=1, indicating that the operating system supports handling of 128-bit media floating-point exceptions.

The exception-mask bits are used by software to specify the handling of 128-bit media floating-point exceptions. When the corresponding mask bit is cleared to 0, an exception occurs under the control of the CR4.OSXMMEXCPT bit. However, if the mask bit is set to 1, the 128-bit media floating-point unit responds in a default manner and execution proceeds normally.

The CR4.OSXMMEXCPT bit specifies the interrupt vector to be taken when an unmasked 128-bit media floating-point exception occurs. When CR4.OSXMMEXCPT=1, the #XF interrupt vector is taken when an exception occurs. When CR4.OSXMMEXCPT=0, the #UD (undefined opcode) interrupt vector is taken when an exception occurs.

The 128-bit media floating-point exceptions reported by the #XF exception are (including mnemonics):

- IE—Invalid-operation exception (also called #I).
- DE—Denormalized-operand exception (also called #D).
- ZE—Zero-divide exception (also called #Z).
- OE—Overflow exception (also called #O).

- UE—Underflow exception (also called #U).
- PE—Precision exception (also called #P or inexact-result exception).

Each type of 128-bit media floating-point exception can be masked by setting the appropriate bits in the MXCSR register. #XF can also be disabled by clearing the CR4.OSXMMEXCPT bit to 0.

Error Code Returned. None. Exception information is provided by the 128-bit media floating-point MXCSR register. See "128-Bit Media and Scientific Programming" in Volume 1 for more information on using this register.

Program Restart. #XF is a fault-type exception. Unlike the #MF exception, the #XF exception is precise. The saved instruction pointer points to the instruction that caused the #XF.

8.3.20 User-Defined Interrupts (Vectors 32–255)

User-defined interrupts can be initiated either by system logic or software. They occur when:

- System logic signals an external interrupt request to the processor. The signalling mechanism and the method of communicating the interrupt vector to the processor are implementation dependent.
- Software executes an INT*n* instruction. The INT*n* instruction operand provides the interrupt vector number.

Both methods can be used to initiate an interrupt into vectors 0 through 255. However, because vectors 0 through 31 are defined or reserved by the AMD64 architecture, software should not use vectors in this range for purposes other than their defined use.

Error Code Returned. None.

Program Restart. The saved instruction pointer depends on the interrupt source:

- External interrupts are recognized on instruction boundaries. The saved instruction pointer points to the instruction immediately following the boundary where the external interrupt was recognized.
- If the interrupt occurs as a result of executing the INT*n* instruction, the saved instruction pointer points to the instruction after the INT*n*.

Masking. The ability to mask user-defined interrupts depends on the interrupt source:

- External interrupts can be masked using the rFLAGS.IF bit. Setting rFLAGS.IF to 1 enables external interrupts, while clearing rFLAGS.IF to 0 inhibits them.
- Software interrupts (initiated by the INT*n* instruction) cannot be disabled.

8.4 Exceptions During a Task Switch

An exception can occur during a task switch while loading a segment selector. Page faults can also occur when accessing a TSS. In these cases, the hardware task-switch mechanism completes loading the new task state from the TSS, and then triggers the appropriate exception mechanism. No other checks are performed. When this happens, the saved instruction pointer points to the first instruction in the new task.

In long mode, an exception cannot occur during a task switch, because the hardware task-switch mechanism is disabled.

8.5 Error Codes

The processor exception-handling mechanism reports error and status information for some exceptions using an error code. The error code is pushed onto the stack by the exception-mechanism during the control transfer into the exception handler. The error code has two formats: a selector format for most error-reporting exceptions, and a page-fault format for page faults. These formats are described in the following sections.

8.5.1 Selector-Error Code

Figure 8-2 shows the format of the selector-error code.

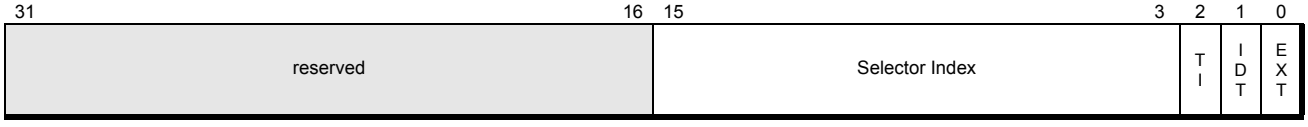


Figure 8-2. Selector Error Code

The information reported by the selector-error code includes:

- *EXT*—Bit 0. If this bit is set to 1, the exception source is external to the processor. If cleared to 0, the exception source is internal to the processor.
- *IDT*—Bit 1. If this bit is set to 1, the error-code selector-index field references a gate descriptor located in the interrupt-descriptor table (IDT). If cleared to 0, the selector-index field references a descriptor in either the global-descriptor table (GDT) or local-descriptor table (LDT), as indicated by the TI bit.
- *TI*—Bit 2. If this bit is set to 1, the error-code selector-index field references a descriptor in the LDT. If cleared to 0, the selector-index field references a descriptor in the GDT. The TI bit is relevant only when the IDT bit is cleared to 0.
- *Selector Index*—Bits 15–3. The selector-index field specifies the index into either the GDT, LDT, or IDT, as specified by the IDT and TI bits.

Some exceptions return a zero in the selector-error code.

8.5.2 Page-Fault Error Code

Figure 8-3 shows the format of the page-fault error code.

Figure 8-3. Page-Fault Error Code

The information reported by the page-fault error code includes:

- *P*—Bit 0. If this bit is cleared to 0, the page fault was caused by a not-present page. If this bit is set to 1, the page fault was caused by a page-protection violation.
- *R/W*—Bit 1. If this bit is cleared to 0, the access that caused the page fault is a memory read. If this bit is set to 1, the memory access that caused the page fault was a write. This bit does not necessarily indicate the cause of the page fault was a read or write violation.
- *U/S*—Bit 2. If this bit is cleared to 0, an access in supervisor mode (CPL=0, 1, or 2) caused the page fault. If this bit is set to 1, an access in user mode (CPL=3) caused the page fault.

This bit does not necessarily indicate the cause of the page fault was a privilege violation.

- *RSV*—Bit 3. If this bit is set to 1, the page fault is a result of the processor reading a 1 from a reserved field within a page-translation-table entry. This type of page fault occurs only when CR4.PSE=1 or CR4.PAE=1. If this bit is cleared to 0, the page fault was not caused by the processor reading a 1 from a reserved field.
- *I/D*—Bit 4. If this bit is set to 1, it indicates that the access that caused the page fault was an instruction fetch. Otherwise, this bit is cleared to 0. This bit is only defined if no-execute feature is enabled (EFER.NXE=1 && CR4.PAE=1).

8.6 Priorities

To allow for consistent handling of multiple-interrupt conditions, simultaneous interrupts are prioritized by the processor. The AMD64 architecture defines priorities between groups of interrupts, and interrupt prioritization within a group is implementation dependent. Table 8-8 on page 268 shows the interrupt priorities defined by the AMD64 architecture.

When simultaneous interrupts occur, the processor transfers control to the highest-priority interrupt handler. Lower-priority interrupts from external sources are held pending by the processor, and they are handled after the higher-priority interrupt is handled. Lower-priority interrupts that result from internal sources are discarded. Those interrupts reoccur when the high-priority interrupt handler completes and transfers control back to the interrupted instruction. Software interrupts are discarded as well, and reoccur when the software-interrupt instruction is restarted.

Table 8-8. Simultaneous Interrupt Priorities

Interrupt Priority	Interrupt Condition	Interrupt Vector
(High)	Processor Reset	—
0	Machine-Check Exception	18
1	External Processor Initialization (INIT)	—
	SMI Interrupt	
	External Clock Stop (Stpclk)	
2	Data, and I/O Breakpoint (Debug Register)	1
	Single-Step Execution Instruction Trap (rFLAGS.TF=1)	
3	Non-Maskable Interrupt	2
4	Maskable External Interrupt (INTR)	32—255
5	Instruction Breakpoint (Debug Register)	1
	Code-Segment-Limit Violation	13
	Instruction-Fetch Page Fault	14
6	Invalid Opcode Exception	6
	Device-Not-Available Exception	7
	Instruction-Length Violation (> 15 Bytes)	13
7	Invalid-TSS Exception	10
	Segment-Not-Present Exception	11
	Stack Exception	12
	General-Protection Exception	13
	Data-Access Page Fault	14
	Floating-Point Exception-Pending Exception	16
	Alignment-Check Exception	17
	SIMD Floating-Point Exception	19

8.6.1 Floating-Point Exception Priorities

Floating-point exceptions (128-bit media and x87 floating-point) can be handled in one of two ways:

- Unmasked exceptions are reported in the appropriate floating-point status register, and a software-interrupt handler is invoked. See "#MF—x87 Floating-Point Exception-Pending (Vector 16)" on page 260 and "#XF—SIMD Floating-Point Exception (Vector 19)" on page 263 for more information on the floating-point interrupts.
- Masked exceptions are also reported in the appropriate floating-point status register. Instead of transferring control to an interrupt handler, however, the processor handles the exception in a default manner and execution proceeds.

If the processor detects more than one exception while executing a single floating-point instruction, it prioritizes the exceptions in a predictable manner. When responding in a default manner to masked exceptions, it is possible that the processor acts only on the high-priority exception and ignores lower-priority exceptions. In the case of vector (SIMD) floating-point instructions, priorities are set on sub-operations, not across all operations. For example, if the processor detects and acts on a QNaN operand in one sub-operation, the processor can still detect and act on a denormal operand in another sub-operation.

When reporting 128-bit media floating-point exceptions before taking an interrupt or handling them in a default manner, the processor first classifies the exceptions as follows:

- *Input exceptions* include SNaN operand (#I), invalid operation (#I), denormal operand (#D), or zero-divide (#Z). Using a NaN operand with a maximum, minimum, compare, or convert instruction is also considered an input exception.
- *Output exceptions* include numeric overflow (#O), numeric underflow (#U), and precision (#P).

Using the above classification, the processor applies the following procedure to report the exceptions:

1. The exceptions for all sub-operations are prioritized.
2. The exception conditions for all sub-operations are logically ORed together to form a single set of exceptions covering all operations. For example, if two sub-operations produce a denormal result, only one denormal exception is reported.

3. If the set of exceptions includes any *unmasked* input exceptions, all input exceptions are reported in MCXSR, and no output exceptions are reported. Otherwise, all input and output exceptions are reported in MCXSR.
4. If any exceptions are unmasked, control is transferred to the appropriate interrupt handler.

Table 8-9 lists the priorities for simultaneous floating-point exceptions.

Table 8-9. Simultaneous Floating-Point Exception Priorities

Exception Priority	Exception Condition	
(High) 0	SNaN Operand	#I
	NaN Operand of Maximum, Minimum, Compare, and Convert Instructions (Vector Floating-Point)	
	Stack Overflow (x87 Floating-Point)	
	Stack Underflow (x87 Floating-Point)	
1	QNaN Operand	–
2	Invalid Operation (Remaining Conditions)	#I
	Zero Divide	#Z
3	Denormal Operand	#D
4	Numeric Overflow	#O
	Numeric Underflow	#U
5 (Low)	Precision	#P

8.6.2 External Interrupt Priorities

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. The organization of these priority classes is implementation dependent. A typical method is to use the upper four bits of the interrupt vector number to define the priority. Thus, interrupt vector 53h has a priority of 5 and interrupt vector 37h has a priority of 3.

A new control register (CR8) is introduced by the AMD64 architecture for managing priority classes. This register, called the *task-priority register* (TPR), uses its four low-order bits to specify a task priority. The remaining 60 bits are reserved and must be written with zeros. Figure 8-4 shows the format of the TPR.

The TPR is available only in 64-bit mode.

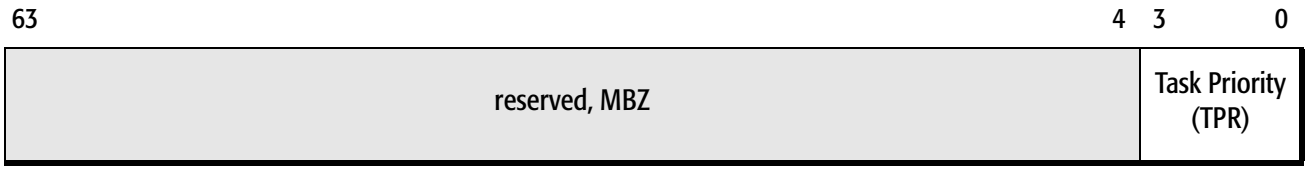


Figure 8-4. Task Priority Register (CR8)

System software can use the TPR register to temporarily block low-priority interrupts from interrupting a high-priority task. This is accomplished by loading TPR with a value corresponding to the highest-priority interrupt that is to be blocked. For example, loading TPR with a value of 9 (1001b) blocks all interrupts with a priority class of 9 or less, while allowing all interrupts with a priority class of 10 or more to be recognized. Loading TPR with 0 enables all external interrupts. Loading TPR with 15 (1111b) disables all external interrupts. The TPR is cleared to 0 on reset.

System software reads and writes the TPR using a MOV CR8 instruction. The MOV CR8 instruction requires a privilege level of 0. Programs running at any other privilege level cannot read or write the TPR, and an attempt to do so results in a general-protection exception (#GP).

A serializing instruction is not required after loading the TPR, because a new priority level is established when the MOV instruction completes execution. For example, assume two sequential TPR loads are performed, in which a low value is first loaded into TPR and immediately followed by a load of a higher value. Any pending, lower-priority interrupt enabled by the first MOV CR8 is recognized between the two MOVs.

The TPR is an architectural abstraction of the interrupt controller (IC), which prioritizes and manages external interrupt delivery to the processor. The IC can be an external

system device, or it can be integrated on the chip like the local advanced programmable interrupt controller (APIC). Typically, the IC contains a priority mechanism similar, if not identical to, the TPR. The IC, however, is implementation dependent, and the underlying priority mechanisms are subject to change. The TPR, by contrast, is part of the AMD64 architecture.

Effect of IC on TPR. The features of the implementation-specific IC can impact the operation of the TPR. For example, the TPR might affect interrupt delivery only if the IC is enabled. Also, the mapping of an external interrupt to a specific interrupt priority is an implementation-specific behavior of the IC.

8.7 Real-Mode Interrupt Control Transfers

In real mode, the IDT is a table of 4-byte entries, one entry for each of the 256 possible interrupts implemented by the system. The real mode IDT is often referred to as an *interrupt-vector table*, or IVT. Table entries contain a far pointer (CS:IP pair) to an exception or interrupt handler. The base of the IDT is stored in the IDTR register, which is loaded with a value of 00h during a processor reset. Figure 8-5 shows how the real-mode interrupt handler is located by the interrupt mechanism.

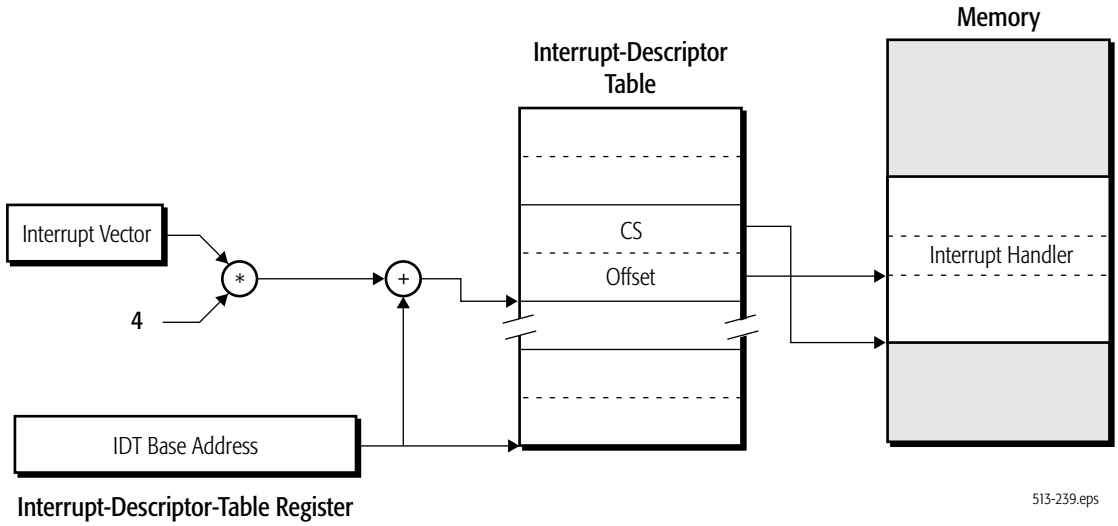


Figure 8-5. Real-Mode Interrupt Control Transfer

When an exception or interrupt occurs in real mode, the processor performs the following:

1. Pushes the FLAGS register (EFLAGS[15:0]) onto the stack.
2. Clears EFLAGS.IF to 0 and EFLAGS.TF to 0.
3. Saves the CS register and IP register (RIP[15:0]) by pushing them onto the stack.
4. Locates the interrupt-handler pointer (CS:IP) in the IDT by scaling the interrupt vector by four and adding the result to the value in the IDTR.
5. Transfers control to the interrupt handler referenced by the CS:IP in the IDT.

Figure 8-6 shows the stack after control is transferred to the interrupt handler in real mode.

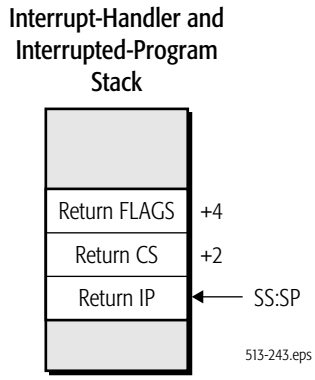


Figure 8-6. Stack After Interrupt in Real Mode

An IRET instruction is used to return to the interrupted program. When an IRET is executed, the processor performs the following:

1. Pops the saved CS value off the stack and into the CS register. The saved IP value is popped into RIP[15:0].
2. Pops the FLAGS value off of the stack and into EFLAGS[15:0].
3. Execution begins at the saved CS.IP location.

8.8 Legacy Protected-Mode Interrupt Control Transfers

In protected mode, the interrupt mechanism transfers control to an exception or interrupt handler through gate descriptors. In protected mode, the IDT is a table of 8-byte gate entries, one for each of the 256 possible interrupt vectors implemented by the system. Three gate types are allowed in the IDT:

- Interrupt gates.
- Trap gates.
- Task gates.

If a reference is made to any other descriptor type in the IDT, a general-protection exception (#GP) occurs.

Interrupt-gate control transfers are similar to CALLs and JMPs through call gates. The interrupt mechanism uses gates (interrupt, trap, and task) to establish protected entry-points into the exception and interrupt handlers.

The remainder of this chapter discusses control transfers through interrupt gates and trap gates. If the gate descriptor in the IDT is a task gate, a TSS-segment selector is referenced, and a task switch occurs. See Chapter 12, “Task Management,” for more information on the hardware task-switch mechanism.

8.8.1 Locating the Interrupt Handler

When an exception or interrupt occurs, the processor scales the interrupt-vector number by eight and uses the result as an offset into the IDT. If the gate descriptor referenced by the IDT offset is an interrupt gate or a trap gate, it contains a segment-selector and segment-offset field (see “Legacy Segment Descriptors” on page 95 for a detailed description of the gate-descriptor format and fields). These two fields perform the same function as the pointer operand in a far control-transfer instruction. The gate-descriptor segment-selector field points to the target code-segment descriptor located in either the GDT or LDT. The gate-descriptor segment-offset field is the instruction-pointer offset into the interrupt-handler code segment. The code-segment base taken from the code-segment descriptor is added to the gate-descriptor segment-offset field to create the interrupt-handler virtual address (linear address).

Figure 8-7 on page 275 shows how the protected-mode interrupt handler is located by the interrupt mechanism.

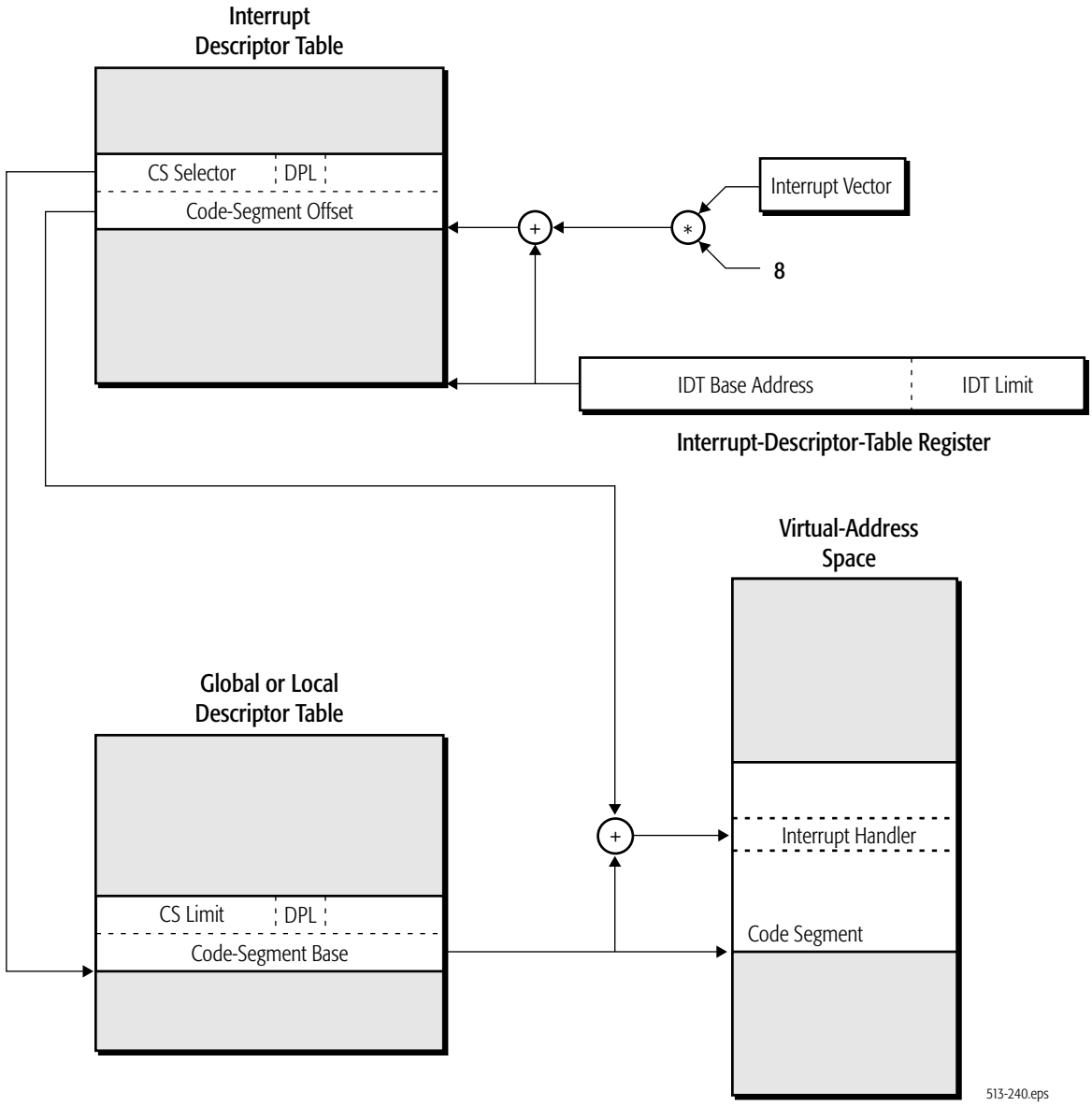


Figure 8-7. Protected-Mode Interrupt Control Transfer

8.8.2 Interrupt To Same Privilege

When a control transfer to an exception or interrupt handler at the same privilege occurs (through an interrupt gate or a trap gate), the processor performs the following:

1. Pushes the EFLAGS register onto the stack.
2. Clears the TF, NT, RF, and VM bits in EFLAGS to 0.

3. The processor handles EFLAGS.IF based on the gate-descriptor type:
   - If the gate descriptor is an interrupt gate, EFLAGS.IF is cleared to 0.
   - If the gate descriptor is a trap gate, EFLAGS.IF is not modified.
4. Saves the return CS register and EIP register (RIP[31:0]) by pushing them onto the stack. The CS value is padded with two bytes to form a doubleword.
5. If the interrupt has an associated error code, the error code is pushed onto the stack.
6. The CS register is loaded from the segment-selector field in the gate descriptor, and the EIP is loaded from the offset field in the gate descriptor.
7. The interrupt handler begins executing with the instruction referenced by new CS:EIP.

Figure 8-8 shows the stack after control is transferred to the interrupt handler.

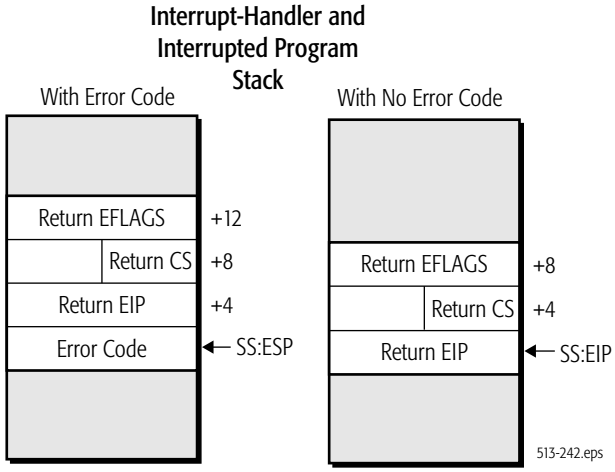


Figure 8-8. Stack After Interrupt to Same Privilege Level

8.8.3 Interrupt To Higher Privilege

When a control transfer to an exception or interrupt handler running at a higher privilege occurs (numerically lower CPL value), the processor performs a stack switch using the following steps:

1. The target CPL is read by the processor from the target code-segment DPL and used as an index into the TSS for selecting the new stack pointer (SS:ESP). For example, if the target CPL is 1, the processor selects the SS:ESP for privilege-level 1 from the TSS.
2. Pushes the return stack pointer (old SS:ESP) onto the new stack. The SS value is padded with two bytes to form a doubleword.
3. Pushes the EFLAGS register onto the new stack.
4. Clears the following EFLAGS bits to 0: TF, NT, RF, and VM.
5. The processor handles the EFLAGS.IF bit based on the gate-descriptor type:
   - If the gate descriptor is an interrupt gate, EFLAGS.IF is cleared to 0.
   - If the gate descriptor is a trap gate, EFLAGS.IF is not modified.
6. Saves the return-address pointer (CS:EIP) by pushing it onto the stack. The CS value is padded with two bytes to form a doubleword.
7. If the interrupt-vector number has an error code associated with it, the error code is pushed onto the stack.
8. The CS register is loaded from the segment-selector field in the gate descriptor, and the EIP is loaded from the offset field in the gate descriptor.
9. The interrupt handler begins executing with the instruction referenced by new CS:EIP.

Figure 8-9 on page 278 shows the new stack after control is transferred to the interrupt handler.

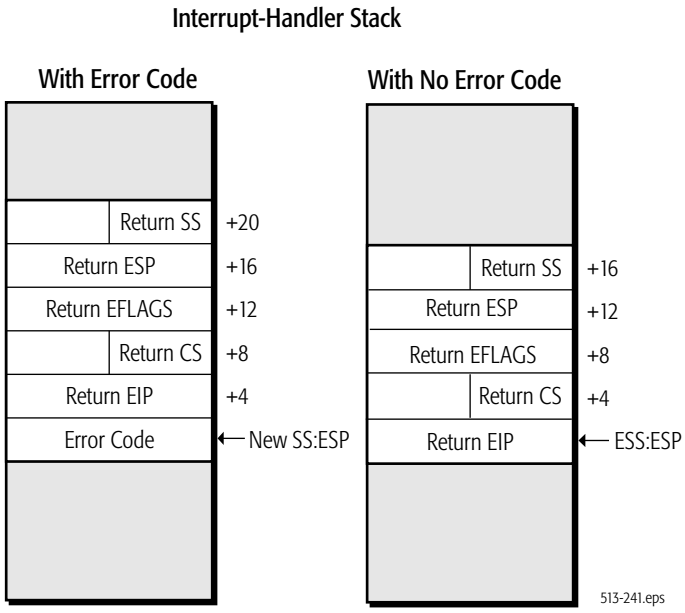


Figure 8-9. Stack After Interrupt to Higher Privilege

8.8.4 Privilege Checks

Before loading the CS register with the interrupt-handler code-segment selector (located in the gate descriptor), the processor performs privilege checks similar to those performed on call gates. The checks are performed when either conforming or nonconforming interrupt handlers are referenced:

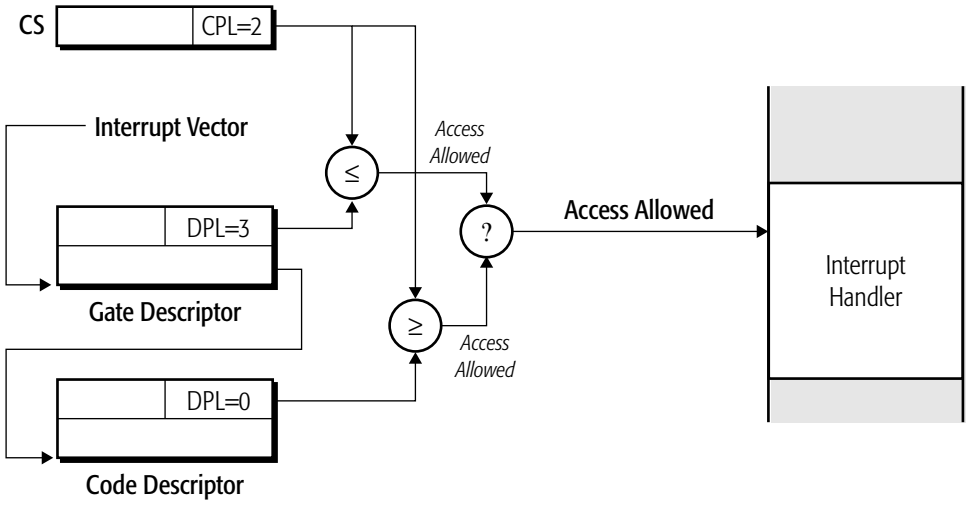
1. The processor reads the gate DPL from the interrupt-gate or trap-gate descriptor. The gate DPL is the *minimum privilege-level* (numerically-highest value) needed by a program to access the gate. The processor compares the CPL with the gate DPL. The CPL must be numerically *less-than or equal-to* the gate DPL for this check to pass.
2. The processor compares the CPL with the interrupt-handler code-segment DPL. For this check to pass, the CPL must be numerically *greater-than or equal-to* the code-segment DPL. This check prevents control transfers to less-privileged interrupt handlers.

Unlike call gates, no RPL comparison takes place. This is because the gate descriptor is referenced in the IDT using the interrupt-vector number rather than a selector, and no RPL field exists in the interrupt-vector number.

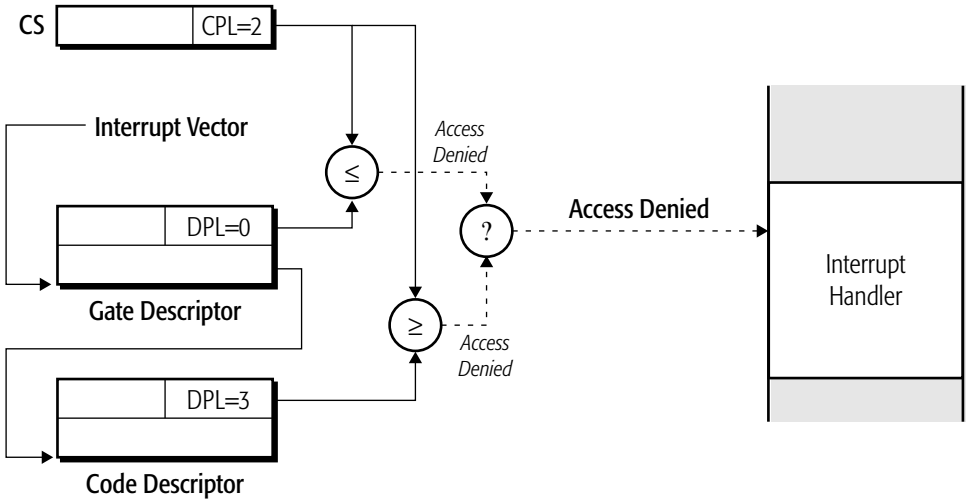
Exception and interrupt handlers should be made reachable from software running at any privilege level that requires them. If the gate DPL value is too low (requiring more privilege), or the interrupt-handler code-segment DPL is too high (runs at lower privilege), the interrupt control transfer can fail the privilege checks. Setting the gate DPL=3 and interrupt-handler code-segment DPL=0 makes the exception handler or interrupt handler reachable from any privilege level.

Figure 8-10 on page 280 shows two examples of interrupt privilege checks. In Example 1, both privilege checks pass:

- The interrupt-gate DPL is at the lowest privilege (3), which means that software running at any privilege level (CPL) can access the interrupt gate.
- The interrupt-handler code segment is at the highest-privilege level, as indicated by DPL=0. This means software running at any privilege can enter the interrupt handler through the interrupt gate.



Example 1: Privilege Check Passes



Example 2: Privilege Check Fails

513-244.eps

Figure 8-10. Privilege-Check Examples for Interrupts

In Example 2, both privilege checks fail:

- The interrupt-gate DPL specifies that only software running at privilege-level 0 can access the gate. The current program does not have enough privilege to access the interrupt gate, because its CPL=2.

- The interrupt handler has a lower privilege (DPL=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 both privilege checks fail, only one such failure is required to deny access to the interrupt handler.

8.8.5 Returning From Interrupt Procedures

A return to an interrupted program should be performed using the IRET instruction. An IRET is a far return to a different code segment, with or without a change in privilege level. The actions of an IRET in both cases are described in the following sections.

IRET, Same Privilege. Before performing the IRET, the stack pointer must point to the return EIP. If there was an error code pushed onto the stack as a result of the exception or interrupt, that error code should have been popped off the stack earlier by the handler. The IRET reverses the actions of the interrupt mechanism:

1. Pops the return pointer off of the stack, loading both the CS register and EIP register (RIP[31:0]) with the saved values. The return code-segment RPL is read by the processor from the CS value stored on the stack to determine that an equal-privilege control transfer is occurring.
2. Pops the saved EFLAGS image off of the stack and into the EFLAGS register.
3. Transfers control to the return program at the target CS:EIP.

IRET, Less Privilege. If an IRET changes privilege levels, the return program must be at a lower privilege than the interrupt handler. The IRET in this case causes a stack switch to occur:

1. The return pointer is popped off of the stack, loading both the CS register and EIP register (RIP[31:0]) with the saved values. 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 saved EFLAGS image is popped off of the stack and loaded into the EFLAGS register.

3. The return-program stack pointer is popped off of the stack, loading both the SS register and ESP register (RSP[31:0]) with the saved values.
4. Control is transferred to the return program at the target CS:EIP.

8.9 Virtual-8086 Mode Interrupt Control Transfers

This section describes interrupt control transfers as they relate to virtual-8086 mode. Virtual-8086 mode is not supported by long mode. Therefore, the control-transfer mechanism described here is not applicable to long mode.

When a software interrupt occurs (not external interrupts, INT1, or INT3) while the processor is running in virtual-8086 mode (EFLAGS.VM=1), the control transfer that occurs depends on three system controls:

- *EFLAGS.IOPL*—This field controls interrupt handling based on the CPL. See "I/O Privilege Level Field (IOPL) Field" on page 64 for more information on this field.

  Setting IOPL<3 redirects the interrupt to the general-protection exception (#GP) handler.
- *CR4.VME*—This bit enables virtual-mode extensions. See "Virtual-8086 Mode Extensions (VME) Bit" on page 59 for more information on this bit.
- *TSS Interrupt-Redirection Bitmap*—The TSS interrupt-redirection bitmap contains 256 bits, one for each possible INT*n* vector (software interrupt). When CR4.VME=1, the bitmap is used by the processor to direct interrupts to the handler provided by the currently-running 8086 program (bitmap entry is 0), or to the protected-mode operating-system interrupt handler (bitmap entry is 1). See "Legacy Task-State Segment" on page 363 for information on the location of this field within the TSS.

If IOPL<3, CR4.VME=1, and the corresponding interrupt redirection bitmap entry is 0, the processor uses the virtual-interrupt mechanism. See "Virtual Interrupts" on page 293 for more information on this mechanism.

Table 8-10 on page 283 summarizes the actions of the above system controls on interrupts taken when the processor is running in virtual-8086 mode.

Table 8-10. Virtual-8086 Mode Interrupt Mechanisms

EFLAGS.IOPL	CR4.VME	TSS Interrupt Redirection Bitmap Entry	Interrupt Mechanism
0, 1, or 2	0	–	General-Protection Exception
	1	1	
	1	0	Virtual Interrupt
3	0	–	Protected-Mode Handler
	1	1	
	1	0	Virtual 8086 Handler

8.9.1 Protected-Mode Handler Control Transfer

Control transfers to protected-mode handlers from virtual-8086 mode differ from standard protected-mode transfers in several ways. The processor follows these steps in making the control transfer:

1. Reads the CPL=0 stack pointer (SS:ESP) from the TSS.
2. Pushes the GS, FS, DS, and ES selector registers onto the stack. Each push is padded with two bytes to form a doubleword.
3. Clears the GS, FS, DS, and ES selector registers to 0. This places a null selector in each of the four registers
4. Pushes the return stack pointer (old SS:ESP) onto the new stack. The SS value is padded with two bytes to form a doubleword.
5. Pushes the EFLAGS register onto the new stack.
6. Clears the following EFLAGS bits to 0: TF, NT, RF, and VM.
7. Handles EFLAGS.IF based on the gate-descriptor type:
   - If the gate descriptor is an interrupt gate, EFLAGS.IF is cleared to 0.
   - If the gate descriptor is a trap gate, EFLAGS.IF is not modified.
8. Pushes the return-address pointer (CS:EIP) onto the stack. The CS value is padded with two bytes to form a doubleword.

9. If the interrupt has an associated error code, pushes the error code onto the stack.
10. Loads the segment-selector field from the gate descriptor into the CS register, and loads the offset field from the gate descriptor into the EIP register.
11. Begins execution of the interrupt handler with the instruction referenced by the new CS:EIP.

Figure 8-11 shows the new stack after control is transferred to the interrupt handler with an error code.

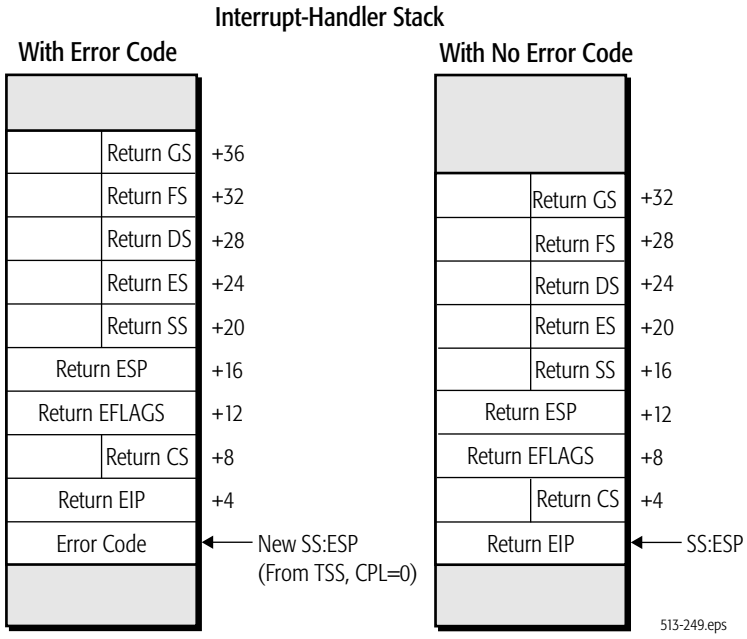


Figure 8-11. Stack After Virtual-8086 Mode Interrupt to Protected Mode

An IRET from privileged protected-mode software (CPL=0) to virtual-8086 mode reverses the stack-build process. After the return pointer, EFLAGS, and return stack-pointer are restored, the processor restores the ES, DS, FS, and GS registers by popping their values off the stack.

8.9.2 Virtual-8086 Handler Control Transfer

When a control transfer to an 8086 handler occurs from virtual-8086 mode, the processor creates an interrupt-handler stack identical to that created when an interrupt occurs in real mode (see Figure 8-6 on page 273). The processor performs the following actions during a control transfer:

1. Pushes the FLAGS register (EFLAGS[15:0]) onto the stack.
2. Clears the EFLAGS.IF and EFLAGS.RF bits to 0.
3. Saves the CS register and IP register (RIP[15:0]) by pushing them onto the stack.
4. Locates the interrupt-handler pointer (CS:IP) in the 8086 IDT by scaling the interrupt vector by four and adding the result to the virtual (linear) address 0. The value in the IDTR is not used.
5. Transfers control to the interrupt handler referenced by the CS:IP in the IDT.

An IRET from the 8086 handler back to virtual-8086 mode reverses the stack-build process.

8.10 Long-Mode Interrupt Control Transfers

The long-mode architecture expands the legacy interrupt-mechanism to support 64-bit operating systems and applications. These changes include:

- All interrupt handlers are 64-bit code and operate in 64-bit mode.
- The size of an interrupt-stack push is fixed at 64 bits (8 bytes).
- The interrupt-stack frame is aligned on a 16-byte boundary.
- The stack pointer, SS:RSP, is pushed unconditionally on interrupts, rather than conditionally based on a change in CPL.
- The SS selector register is loaded with a null selector as a result of an interrupt, if the CPL changes.
- The IRET instruction behavior changes, to unconditionally pop SS:RSP, allowing a null SS to be popped.
- A new interrupt stack-switch mechanism, called the interrupt-stack table or IST, is introduced.

8.10.1 Interrupt Gates and Trap Gates

Only long-mode interrupt and trap gates can be referenced in long mode (64-bit mode and compatibility mode). The legacy 32-bit interrupt-gate and 32-bit trap-gate types (0Eh and 0Fh, as described in “System Descriptors” on page 109) are redefined in long mode as 64-bit interrupt-gate and 64-bit trap-gate types. 32-bit and 16-bit interrupt-gate and trap-gate types do not exist

in long mode, and software is prohibited from using task gates. If a reference is made to any gate other than a 64-bit interrupt gate or a 64-bit trap gate, a general-protection exception (#GP) occurs.

The long-mode gate types are 16 bytes (128 bits) long. They are an extension of the legacy-mode gate types, allowing a full 64-bit segment offset to be stored in the descriptor. See "Legacy Segment Descriptors" on page 95 for a detailed description of the gate-descriptor format and fields.

8.10.2 Locating the Interrupt Handler

When an interrupt occurs in long mode, the processor multiplies the interrupt-vector number by 16 and uses the result as an offset into the IDT. The gate descriptor referenced by the IDT offset contains a segment-selector and a 64-bit segment-offset field. The gate-descriptor segment-offset field contains the complete virtual address for the interrupt handler. The gate-descriptor segment-selector field points to the target code-segment descriptor located in either the GDT or LDT. The code-segment descriptor is only used for privilege-checking purposes and for placing the processor in 64-bit mode. The code segment-descriptor base field, limit field, and most attributes are ignored.

Figure 8-12 on page 287 shows how the long-mode interrupt handler is located by the interrupt mechanism.

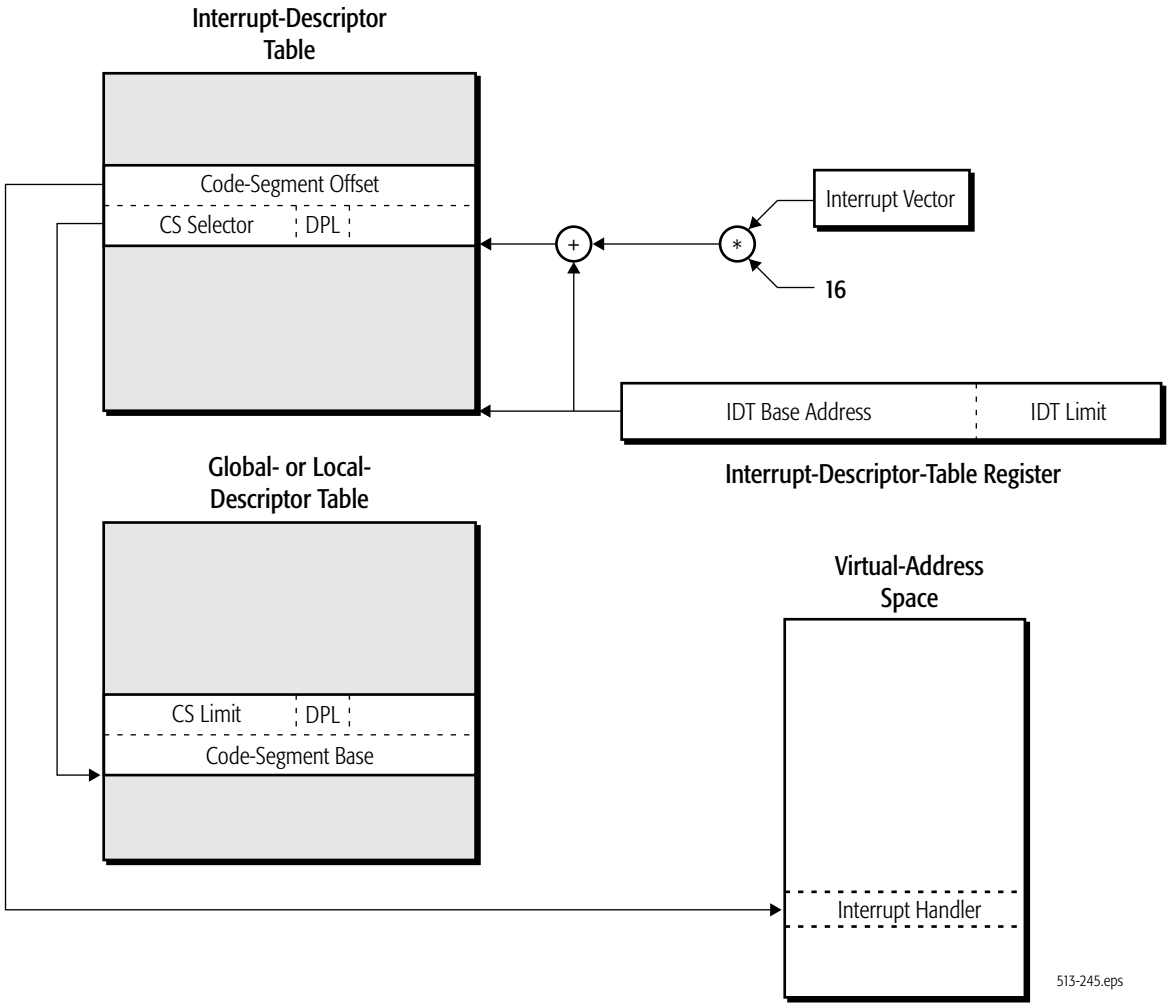


Figure 8-12. Long-Mode Interrupt Control Transfer

8.10.3 Interrupt Stack Frame

In long mode, the return-program stack pointer (SS:RSP) is always pushed onto the interrupt-handler stack, regardless of whether or not a privilege change occurs. Although the SS register is not used in 64-bit mode, SS is pushed to allow returns into compatibility mode. Pushing SS:RSP unconditionally presents operating systems with a consistent interrupt-stack-frame size for all interrupts, except for error codes. Interrupt service-routine entry points that handle interrupts generated by non-error-code interrupts can push an error code on the stack for consistency.

In long mode, when a control transfer to an interrupt handler occurs, the processor performs the following:

1. Aligns the new interrupt-stack frame by masking RSP with FFFF_FFFF_FFFF_FFF0h.
2. If IST field in interrupt gate is not 0, reads IST pointer into RSP.
3. If a privilege change occurs, the target DPL is used as an index into the long-mode TSS to select a new stack pointer (RSP).
4. If a privilege change occurs, SS is cleared to zero indicating a null selector.
5. Pushes the return stack pointer (old SS:RSP) onto the new stack. The SS value is padded with six bytes to form a quadword.
6. Pushes the 64-bit RFLAGS register onto the stack. The upper 32 bits of the RFLAGS image on the stack are written as zeros.
7. Clears the TF, NT, and RF bits in RFLAGS bits to 0.
8. Handles the RFLAGS.IF bit according to the gate-descriptor type:
   - If the gate descriptor is an interrupt gate, RFLAGS.IF is cleared to 0.
   - If the gate descriptor is a trap gate, RFLAGS.IF is not modified.
9. Pushes the return CS register and RIP register onto the stack. The CS value is padded with six bytes to form a quadword.
10. If the interrupt-vector number has an error code associated with it, pushes the error code onto the stack. The error code is padded with four bytes to form a quadword.
11. Loads the segment-selector field from the gate descriptor into the CS register. The processor checks that the target code-segment is a 64-bit mode code segment.
12. Loads the offset field from the gate descriptor into the target RIP. The interrupt handler begins execution when control is transferred to the instruction referenced by the new RIP.

Figure 8-13 shows the stack after control is transferred to the interrupt handler.

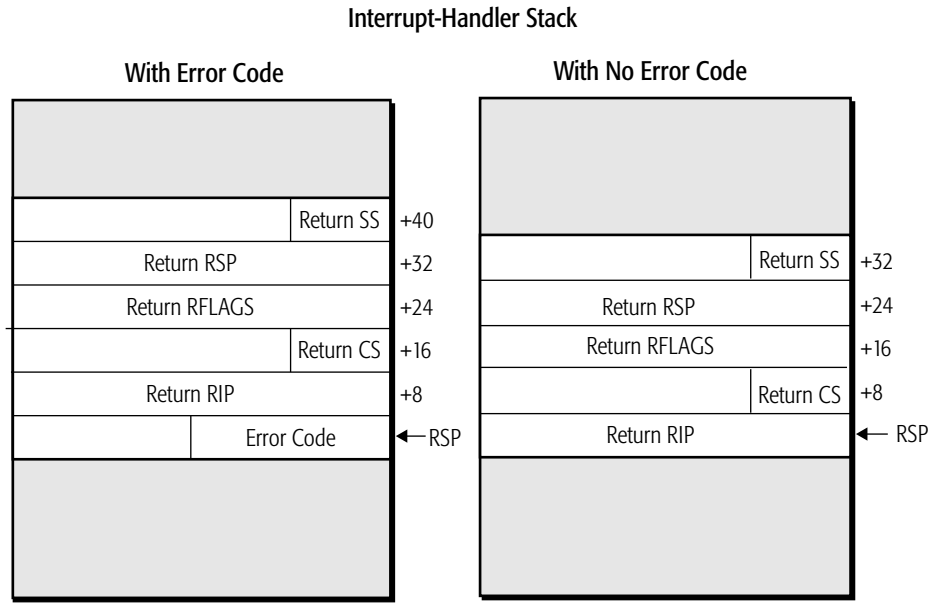


Figure 8-13. Long-Mode Stack After Interrupt—Same Privilege

Interrupt-Stack Alignment. In legacy mode, the interrupt-stack pointer can be aligned at any address boundary. Long mode, however, aligns the stack on a 16-byte boundary. This alignment is performed by the processor in hardware before pushing items onto the stack frame. The previous RSP is saved unconditionally on the new stack by the interrupt mechanism. A subsequent IRET instruction automatically restores the previous RSP.

Aligning the stack on a 16-byte boundary allows optimal performance for saving and restoring the 16-byte XMM registers. The interrupt handler can save and restore the XMM registers using the faster 16-byte aligned loads and stores (MOVAPS), rather than unaligned loads and stores (MOVUPS). Although the RSP alignment is always performed in long mode, it is only of consequence when the interrupted program is already running at CPL=0, and it is generally used only within the operating-system kernel. The operating system should put 16-byte aligned RSP values in the TSS for interrupts that change privilege levels.

Stack Switch. In long mode, the stack-switch mechanism differs slightly from the legacy stack-switch mechanism (see "Interrupt To Higher Privilege" on page 276). When stacks are switched during a long-mode privilege-level change resulting from an interrupt, a new SS descriptor is *not* loaded from the TSS. Long mode only loads an inner-level RSP from the TSS. However, the SS selector is loaded with a null selector, allowing nested control transfers, including interrupts, to be handled properly in 64-bit mode. The SS.RPL is set to the new CPL value. See "Nested IRETs to 64-Bit Mode Procedures" on page 293 for additional information.

The interrupt-handler stack that results from a privilege change in long mode looks identical to a long-mode stack when no privilege change occurs. Figure 8-14 shows the stack after the switch is performed and control is transferred to the interrupt handler.

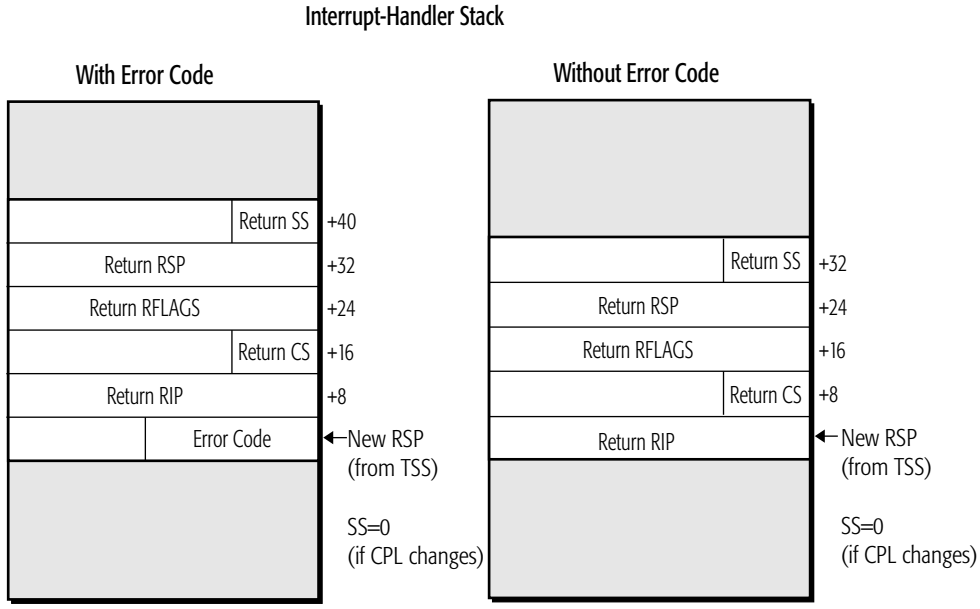


Figure 8-14. Long-Mode Stack After Interrupt—Higher Privilege

8.10.4 Interrupt-Stack Table

In long mode, a new interrupt-stack table (IST) mechanism is introduced as an alternative to the modified legacy stack-switch mechanism described above. The IST mechanism provides a method for specific interrupts, such as NMI, double-fault, and machine-check, to always execute on a known-good stack. In legacy mode, interrupts can use the hardware task-switch

mechanism to set up a known-good stack by accessing the interrupt service routine through a task gate located in the IDT. However, the hardware task-switch mechanism is not supported in long mode.

When enabled, the IST mechanism unconditionally switches stacks. It can be enabled on an individual interrupt-vector basis using a new field in the IDT gate-descriptor entry. This allows some interrupts to use the modified legacy mechanism, and others to use the IST mechanism. The IST mechanism is only available in long mode.

The IST mechanism uses new fields in the 64-bit TSS format and the long-mode interrupt-gate and trap-gate descriptors:

- Figure 12-8 on page 370 shows the format of the 64-bit TSS and the location of the seven IST pointers. The 64-bit TSS offsets from 24h to 5Bh provide space for seven IST pointers, each of which are 64 bits (8 bytes) long.
- The long-mode interrupt-gate and trap-gate descriptors define a 3-bit IST-index field in bits 2–0 of byte +4. Figure 4-24 on page 113 shows the format of long-mode interrupt-gate and trap-gate descriptors and the location of the IST-index field.

To enable the IST mechanism for a specific interrupt, system software stores a non-zero value in the interrupt gate-descriptor IST-index field. If the IST index is zero, the modified legacy stack-switching mechanism (described in the previous section) is used.

Figure 8-15 on page 292 shows how the IST mechanism is used to create the interrupt-handler stack. When an interrupt occurs and the IST index is non-zero, the processor uses the index to select the corresponding IST pointer from the TSS. The IST pointer is loaded into the RSP to establish a new stack for the interrupt handler. The SS register is loaded with a null selector if the CPL changes and the SS.RPL is set to the new CPL value. After the stack is loaded, the processor pushes the old stack pointer, RFLAGS, the return pointer, and the error code (if applicable) onto the stack. Control is then transferred to the interrupt handler.

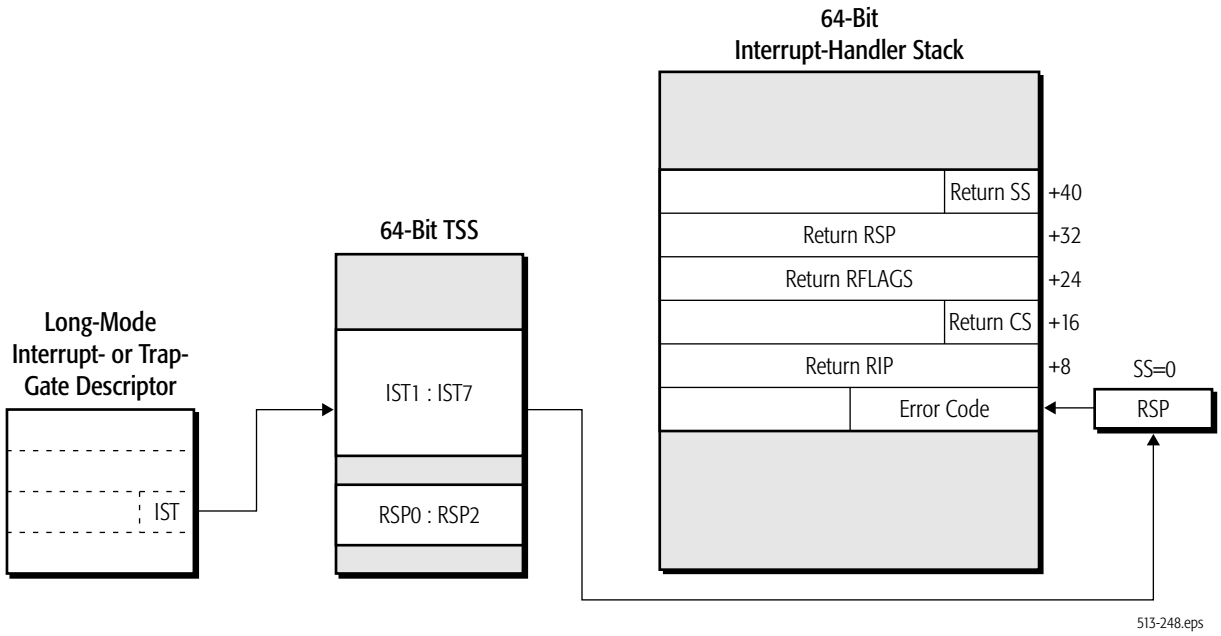


Figure 8-15. Long-Mode IST Mechanism

8.10.5 Returning From Interrupt Procedures

As with legacy mode, a return to an interrupted program in long mode should be performed using the IRET instruction. However, in long mode, the IRET semantics are different from legacy mode:

- *In 64-bit mode*, IRET pops the return-stack pointer unconditionally off the interrupt-stack frame and into the SS:RSP registers. This reverses the action of the long-mode interrupt mechanism, which saves the stack pointer whether or not a privilege change occurs. IRET also allows a null selector to be popped off the stack and into the SS register. See "Nested IRETs to 64-Bit Mode Procedures" on page 293 for additional information.
- *In compatibility mode*, IRET behaves as it does in legacy mode. The SS:ESP is popped off the stack only if a control transfer to less privilege (numerically greater CPL) is performed. Otherwise, it is assumed that a stack pointer is not present on the interrupt-handler stack.

The long-mode interrupt mechanism always uses a 64-bit stack when saving values for the interrupt handler, and the interrupt handler is always entered in 64-bit mode. To work properly, an

IRET used to exit the 64-bit mode interrupt-handler requires a series of eight-byte pops off the stack. This is accomplished by using a 64-bit operand-size prefix with the IRET instruction. The default stack size assumed by an IRET in 64-bit mode is 32 bits, so a 64-bit REX prefix is needed by 64-bit mode interrupt handlers.

Nested IRETs to 64-Bit Mode Procedures. In long mode, an interrupt causes a null selector to be loaded into the SS register if the CPL changes (this is the same action taken by a far CALL in long mode). If the interrupt handler performs a far call, or is itself 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.

The null selector enables the processor to properly handle nested returns to 64-bit mode (which do not use the SS register), and returns to compatibility mode (which do use the SS register). Normally, an IRET that pops a null selector into the SS register causes a general-protection exception (#GP) to occur. However, in long mode, the null selector indicates the existence of nested interrupt handlers and/or privileged software in 64-bit mode. Long mode allows an IRET to pop a null selector into SS from the stack under the following conditions:

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

8.11 Virtual Interrupts

The term *virtual interrupts* includes two classes of extensions to the interrupt-handling mechanism:

- *Virtual-8086 Mode Extensions (VME)*—These allow virtual interrupts and interrupt redirection in virtual-8086 mode. VME has no effect on protected-mode programs.
- *Protected-Mode Virtual Interrupts (PVI)*—These allow virtual interrupts in protected mode when CPL=3. Interrupt

redirection is not available in protected mode. PVI has no effect on virtual-8086-mode programs.

Because virtual-8086 mode is not supported in long mode, VME extensions are not supported in long mode. PVI extensions are, however, supported in long mode.

8.11.1 Virtual-8086 Mode Extensions

The virtual-8086-mode extensions (VME) enable performance enhancements for 8086 programs running as protected tasks in virtual-8086 mode. These extensions are enabled by setting CR4.VME (bit 0) to 1. The extensions enabled by CR4.VME are:

- Virtualizing control and notification of maskable external interrupts with the EFLAGS VIF (bit 19) and VIP (bit 20) bits.
- Selective interception of software interrupts (INT*n* instructions) using the TSS interrupt redirection bitmap (IRB).

Background. Legacy-8086 programs expect to have full access to the EFLAGS interrupt flag (IF) bit, allowing programs to enable and disable maskable external interrupts. When those programs run in virtual-8086 mode under a multitasking protected-mode environment, it can disrupt the operating system if programs enable or disable interrupts for their own purposes. This is particularly true if interrupts associated with one program can occur during execution of another program. For example, a program could request that an area of memory be copied to disk. System software could suspend the program before external hardware uses an interrupt to acknowledge that the block has been copied. System software could subsequently start a second program which enables interrupts. This second program could receive the external interrupt indicating that the memory block of the first program has been copied. If that were to happen, the second program would probably be unprepared to handle the interrupt properly.

Access to the IF bit must be managed by system software on a task-by-task basis to prevent corruption of system resources. In order to completely manage the IF bit, system software must be able to interrupt all instructions that can read or write the bit. These instructions include STI, CLI, PUSHF, POPF, INT*n*, and IRET. These instructions are part of an instruction class that is *IOPL-sensitive*. The processor takes a general-protection exception (#GP) whenever an IOPL-sensitive instruction is

executed and the EFLAGS.IOPL field is less than the CPL. Because all virtual-8086 programs run at CPL=3, system software can interrupt all instructions that modify the IF bit by setting IOPL<3.

System software maintains a virtual image of the IF bit for each virtual-8086 program by emulating the actions of IOPL-sensitive instructions that modify the IF bit. When an external maskable-interrupt occurs, system software checks the state of the IF image for the current virtual-8086 program to determine whether the program is masking interrupts. If the program is masking interrupts, system software saves the interrupt information until the virtual-8086 program attempts to re-enable interrupts. When the virtual-8086 program unmasks interrupts with an IOPL-sensitive instruction, system software traps the action with the #GP handler.

The performance of a processor can be significantly degraded by the overhead of trapping and emulating IOPL-sensitive instructions, and the overhead of maintaining images of the IF bit for each virtual-8086 program. This performance loss can be eliminated by running virtual-8086 programs with IOPL set to 3, thus allowing changes to the real IF flag from any privilege level. Unfortunately, this can leave critical system resources unprotected.

In addition to the performance problems caused by virtualizing the IF bit, software interrupts (INT*n* instructions) cannot be masked by the IF bit or virtual copies of the IF bit. The IF bit only affects maskable external interrupts. Software interrupts in virtual-8086 mode are normally directed to the real mode interrupt-vector table (IVT), but it can be desirable to redirect certain interrupts to the protected-mode interrupt-descriptor table (IDT).

The virtual-8086-mode extensions are designed to support both external interrupts and software interrupts, with mechanisms that preserve high performance without compromising protection. Virtualization of external interrupts is supported using two bits in the EFLAGS register: the virtual-interrupt flag (VIF) bit and the virtual-interrupt pending (VIP) bit. Redirection of software interrupts is supported using the interrupt-redirection bitmap (IRB) in the TSS. A separate TSS can be created for each virtual-8086 program, allowing system

software to control interrupt redirection independently for each virtual-8086 program.

VIF and VIP Extensions for External Interrupts. When VME extensions are enabled, the IF-modifying instructions normally trapped by system software are allowed to execute. However, instead of modifying the IF bit, they modifying the EFLAGS VIF bit. This leaves control over maskable interrupts to the system software. It can also be used as an indicator to system software that the virtual-8086 program is able to, or is expecting to, receive external interrupts.

When an unmasked external interrupt occurs, the processor transfers control from the virtual-8086 program to a protected-mode interrupt handler. If the interrupt handler determines that the interrupt is for the virtual-8086 program, it can check the state of the VIF bit in the EFLAGS value pushed on the stack for the virtual-8086 program. If the VIF bit is set (indicating the virtual-8086 program attempted to unmask interrupts), system software can allow the interrupt to be handled by the appropriate virtual-8086 interrupt handler.

If the VIF bit is clear (indicating the virtual-8086 program attempted to mask interrupts) and the interrupt is for the virtual-8086 program, system software can hold the interrupt pending. System software holds an interrupt pending by saving appropriate information about the interrupt, such as the interrupt vector, and setting the virtual-8086 program's VIP bit in the EFLAGS image on the stack. When the virtual-8086 program later attempts to set IF, the previously set VIP bit causes a general-protection exception (#GP) to occur. System software can then pass the saved interrupt information to the virtual-8086 interrupt handler.

To summarize, when the VME extensions are enabled (CR4.VME=1), the VIF and VIP bits are set and cleared as follows:

- *VIF Bit*—This bit is set and cleared by the processor in virtual-8086 mode in response to an attempt by a virtual-8086 program to set and clear the EFLAGS.IF bit. VIF is used by system software to determine whether a maskable external interrupt should be passed on to the virtual-8086 program, emulated by system software, or held pending. VIF is also cleared during software interrupts through interrupt

gates, with the original VIF value preserved in the EFLAGS image on the stack.

- *VIP Bit*—System software sets and clears this bit in the EFLAGS image saved on the stack after an interrupt. It can be set when an interrupt occurs for a virtual-8086 program that has a clear VIF bit. The processor examines the VIP bit when an attempt is made by the virtual-8086 program to set the IF bit. If VIP is set when the program attempts to set IF, a general-protection exception (#GP) occurs *before* execution of the IF-setting instruction. System software must clear VIP to avoid repeated #GP exceptions when returning to the interrupted instruction.

The VIF and VIP bits can be used by system software to minimize the overhead associated with managing maskable external interrupts because virtual copies of the IF flag do not have to be maintained by system software. Instead, VIF and VIP are maintained during context switches along with the remaining EFLAGS bits.

Table 8-11 on page 299 shows how the behavior of instructions that modify the IF bit are affected by the VME extensions.

Interrupt Redirection of Software Interrupts. In virtual-8086 mode, software interrupts (INT*n* instructions) are trapped using a #GP exception handler if the IOPL is less than 3 (the CPL for virtual-8086 mode). This allows system software to interrupt and emulate 8086-interrupt handlers. System software can set the IOPL to 3, in which case the INT*n* instruction is vectored through a gate descriptor in the protected-mode IDT. System software can use the gate to control access to the virtual-8086 mode interrupt-vector table (IVT), or to redirect the interrupt to a protected-mode interrupt handler.

When VME extensions are enabled, for INT*n* instructions to execute normally, vectoring directly to a virtual-8086 interrupt handler through the virtual-8086 IVT (located at address 0 in the virtual-address space of the task). For security or performance reasons, however, it can be necessary to intercept INT*n* instructions on a vector-specific basis to allow servicing by protected-mode interrupt handlers. This is performed by using the interrupt-redirection bitmap (IRB), located in the TSS and enabled when CR4.VME=1. The IRB is available only in virtual-8086 mode.

Figure 12-6 on page 364 shows the format of the TSS, with the interrupt redirection bitmap located near the top. The IRB contains 256 bits, one for each possible software-interrupt vector. The most-significant bit of the IRB controls interrupt vector 255, and is located immediately before the IOPB base. The least-significant bit of the IRB controls interrupt vector 0.

The bits in the IRB function as follows:

- When set to 1, the INT*n* instruction behaves as if the VME extensions are not enabled. The interrupt is directed through the IDT to a protected-mode interrupt handler if IOPL=3. If IOPL<3, the INT*n* causes a #GP exception.
- When cleared to 0, the INT*n* instruction is directed through the IVT for the virtual-8086 program to the corresponding virtual-8086 interrupt handler.

Only software interrupts can be redirected using the IRB mechanism. External interrupts are asynchronous events that occur outside the context of a virtual-8086 program. Therefore, external interrupts require system-software intervention to determine the appropriate context for the interrupt. The VME extensions described in "VIF and VIP Extensions for External Interrupts" on page 296 are provided to assist system software with external-interrupt intervention.

8.11.2 Protected Mode Virtual Interrupts

The protected-mode virtual-interrupt (PVI) bit in CR4 enables support for interrupt virtualization in protected mode. When enabled, the processor maintains program-specific VIF and VIP bits similar to the manner defined by the virtual-8086 mode extensions (VME). However, unlike VME, only the STI and CLI instructions are affected by the PVI extension. When a program is running at CPL=3, it can use STI and CLI to set and clear its copy of the VIF flag without causing a general-protection exception. The last section of Table 8-11 on page 299 describes the behavior of instructions that modify the IF bit when PVI extensions are enabled.

The interrupt redirection bitmap (IRB) defined by the VME extensions is not supported by the PVI extensions.

8.11.3 Effect of Instructions that Modify EFLAGS.IF

Table 8-11 on page 299 shows how the behavior of instructions that modify the IF bit are affected by the VME and PVI extensions. The table columns specify the following:

- *Operating Mode*—the processor mode in effect when the instruction is executed.
- *Instruction*—the IF-modifying instruction.
- *IOPL*—the value of the EFLAGS.IOPL field.
- *VIP*—the value of the EFLAGS.VIP bit.
- *#GP*—indicates whether the conditions in the first four columns cause a general-protection exception (#GP) to occur.
- *Effect on IF Bit*—indicates the effect the conditions in the first four columns have on the EFLAGS.IF bit and the image of EFLAGS.IF on the stack.
- *Effect on VIF Bit*—indicates the effect the conditions in the first four columns have on the EFLAGS.VIF bit and the image of EFLAGS.VIF on the stack.

Table 8-11. Effect of Instructions that Modify the IF Bit

Operating Mode	Instruction	IOPL	VIP	#GP	Effect on IF Bit	Effect on VIF Bit
Real Mode *CR0.PE=0* *EFLAGS.VM=0* *CR4.VME=0* *CR4.PVI=0*	CLI			no	IF = 0	
	STI				IF = 1	
	PUSHF				EFLAGS.IF Stack Image = IF	
	POPF				IF = EFLAGS.IF stack image	
	INT*n*				EFLAGS.IF Stack Image = IF IF = 0	
	IRET				IF = EFLAGS.IF Stack Image	

Note:
Gray-shaded boxes indicate the bits are unsupported (ignored) in the specified operating mode.
"x" indicates the value of the bit is a "don't care".
"–" indicates the instruction causes a general-protection exception (#GP).

Note:
1. *If the EFLAGS.IF stack image is 0, no #GP exception occurs and the IRET instruction is executed.*
2. *If the EFLAGS.IF stack image is 1, the IRET is not executed, and a #GP exception occurs.*

Table 8-11. Effect of Instructions that Modify the IF Bit (continued)

Operating Mode	Instruction	IOPL	VIP	#GP	Effect on IF Bit	Effect on VIF Bit
Protected Mode *CR0.PE=1* *EFLAGS.VM=0* *CR4.VME=x* *CR4.PVI=0*	CLI	≥CPL		no	IF = 0	
		<CPL		yes	–	
	STI	≥CPL		no	IF = 1	
		<CPL		yes	–	
	PUSHF	x		no	EFLAGS.IF Stack Image = IF	
	POPF	≥CPL			IF = EFLAGS.IF Stack Image	
		<CPL			No Change	
	INT*n* gate	x			EFLAGS.IF Stack Image = IF IF = 0	
	IRET				IF = EFLAGS.IF Stack Image	
	IRETD					
Virtual-8086 Mode *CR0.PE=1* *EFLAGS.VM=1* *CR4.VME=0* *CR4.PVI=x*	CLI	3		no	IF = 0	
		< 3		yes	–	
	STI	3		no	IF = 1	
		< 3		yes	–	
	PUSHF	3		no	EFLAGS.IF Stack Image = IF	
		< 3		yes	–	
	POPF	3		no	IF = EFLAGS.IF Stack Image	
		< 3		yes	–	
	INT*n* gate	3		no	EFLAGS.IF Stack Image = IF IF = 0	
		< 3		yes	–	
	IRET	3		no	IF = EFLAGS.IF Stack Image	
		< 3		yes	–	
	IRETD	3		no	IF = EFLAGS.IF Stack Image	
		< 3		yes	–	

Note:

Gray-shaded boxes indicate the bits are unsupported (ignored) in the specified operating mode.
"x" indicates the value of the bit is a "don't care".
"–" indicates the instruction causes a general-protection exception (#GP).

Note:

1. *If the EFLAGS.IF stack image is 0, no #GP exception occurs and the IRET instruction is executed.*
2. *If the EFLAGS.IF stack image is 1, the IRET is not executed, and a #GP exception occurs.*

Table 8-11. Effect of Instructions that Modify the IF Bit (continued)

Operating Mode	Instruction	IOPL	VIP	#GP	Effect on IF Bit	Effect on VIF Bit
Virtual-8086 Mode *with VME Extensions* *CR0.PE=1* *EFLAGS.VM=1* *CR4.VME=1* *CR4.PVI=x*	CLI	3	x	no	IF = 0	No Change
		<3			No Change	VIF = 0
	STI	3	x	no	IF = 1	No Change
		<3	0	no	No Change	VIF = 1
			1	yes	–	
	PUSHF	3	x	no	EFLAGS.IF Stack Image = IF	Not Pushed
		<3			Not Pushed	EFLAGS.IF Stack Image = VIF
	PUSHFD	3	x	no	EFLAGS.IF Stack Image = IF	EFLAGS.VIF Stack Image = VIF
		<3		yes	–	
	POPF	3	x	no	IF = EFLAGS.IF Stack Image	No Change
		<3	0	no	No Change	VIF = EFLAGS.IF Stack Image
			1	yes	–	
	POPFD	3	x	no	IF = EFLAGS.IF Stack Image	No Change
		<3		yes	–	
	INT*n* gate	3	x	no	EFLAGS.IF Stack Image = IF IF = 0	No Change
		<3			No Change	EFLAGS.IF Stack Image = VIF VIF = 0
	IRET	3	x	no	IF = EFLAGS.IF Stack Image	No Change
		<3	0	no	No Change	VIF = EFLAGS.IF Stack Image
			1	no[1]	No Change	VIF = EFLAGS.IF Stack Image
				yes[2]	–	
	IRETD	3	x	no	IF = EFLAGS.IF Stack Image	VIF = EFLAGS.IF Stack Image
		<3		yes	–	

Note:

Gray-shaded boxes indicate the bits are unsupported (ignored) in the specified operating mode.

"x" indicates the value of the bit is a "don't care".

"–" indicates the instruction causes a general-protection exception (#GP).

Note:

1. *If the EFLAGS.IF stack image is 0, no #GP exception occurs and the IRET instruction is executed.*
2. *If the EFLAGS.IF stack image is 1, the IRET is not executed, and a #GP exception occurs.*

Table 8-11. Effect of Instructions that Modify the IF Bit (continued)

Operating Mode	Instruction	IOPL	VIP	#GP	Effect on IF Bit	Effect on VIF Bit
Protected Mode *with PVI Extensions* *CR0.PE=1* *EFLAGS.VM=0* *CR4.VME=x* *CR4.PVI=1* *CPL=3*	CLI	3	x	no	IF = 0	No Change
		<3			No Change	VIF = 0
	STI	3	x	no	IF = 1	No Change
		<3	0	no	No Change	VIF = 1
			1	yes	–	
	PUSHF	x	x	no	EFLAGS.IF Stack Image = IF	Not Pushed
	PUSHFD					EFLAGS.VIF Stack Image = VIF
	POPF				IF = EFLAGS.IF Stack Image	No Change
	POPFD					VIF = 0
	INT*n* gate				EFLAGS.IF Stack Image = IF IF = 0 (if interrupt gate)	No Change
	IRET				IF = EFLAGS.IF Stack Image	No Change
	IRETD					VIF = EFLAGS.VIF Stack Image

Note:

Gray-shaded boxes indicate the bits are unsupported (ignored) in the specified operating mode.

"x" indicates the value of the bit is a "don't care".

"–" indicates the instruction causes a general-protection exception (#GP).

Note:

1. *If the EFLAGS.IF stack image is 0, no #GP exception occurs and the IRET instruction is executed.*
2. *If the EFLAGS.IF stack image is 1, the IRET is not executed, and a #GP exception occurs.*

9 Machine Check Mechanism

The machine-check mechanism allows the processor to detect and report a variety of hardware errors. The types of errors that can be reported include:

- Cache errors associated with reading and writing data, probing, cache-line fills, and cache-line writebacks.
- Parity errors associated with the caches (including external caches) and TLBs.
- ECC errors associated with the caches (including external caches) and DRAM.
- Bus errors associated with reading and writing on the processor external bus.

Software can enable the processor to report machine-check errors through the machine-check exception (for additional information, see "#MC—Machine-Check Exception (Vector 18)" on page 262). Most machine-check error conditions do not allow reliable restarting of interrupted programs. System software instead uses the machine-check mechanism to report the source of hardware problems for possible servicing.

The basic machine-check mechanism is consistent across processor implementations, but the error-reporting registers are model specific. Processor implementations are not required to support the mechanism, and those implementations that do support it can vary in how the mechanism works.

9.1 Determining Machine-Check Support

The availability of machine-check registers and support of the machine-check exception is implementation dependent. System software executes the CPUID instruction to determine whether a processor implements these features. After CPUID is executed, the values of the machine-check architecture (MCA) bit and the machine-check exception (MCE) bit loaded in the EDX register indicate whether the processor implements the machine-check registers and the machine-check exception, respectively. See "Processor Feature Identification" on page 75 for more information on using the CPUID instruction to determine the level of machine-check support.

Once system software determines that the machine-check registers are available, it must determine the extent of processor support for the machine-check mechanism. This is accomplished by reading the machine-check capabilities register (MCG_CAP). See "Machine-Check Global-Capabilities Register" on page 306 for more information on interpreting the MCG_CAP contents.

Implementation-specific information concerning the machine-check mechanism can be found in the BIOS writer's guide for the implementation.

9.2 Machine-Check Errors

The following classes of machine-check errors are defined for the AMD64 architecture:

- *Recoverable*—The error has been corrected by the processor. Recoverable errors do not cause a machine check exception (#MC). However, the error is still logged in the machine-check registers. It is the responsibility of system software to periodically poll the machine-check registers to determine whether recoverable errors have occurred.
- *Fatal/Unrecoverable*—The error cannot be corrected by the processor. Unrecoverable errors cause a machine check exception if CR4.MCE is set to 1.

In both cases, the contents of the machine-check registers are maintained through a warm reset, which allows errors to be reported even if a reset occurs.

9.2.1 Error Sources

Implementations can detect errors from any number of sources located within the various processor units. Those processor units can include the following:

- *Data-Cache Unit (DC)*—Includes the cache structures that hold data and tags, the data TLBs, and cache-probing logic.
- *Instruction-Cache Unit (IC)*—Includes the cache structures that hold instructions and tags, the instruction TLBs, and cache-probing logic.
- *Bus Unit (BU)*—Includes the L2 cache and any external caches.
- *Load/Store Unit (LS)*—Includes logic used to manage loads and stores.

- *Northbridge (NB)*—Includes the system-bus interface and DRAM controller.

A given processor implementation can monitor machine-check errors in sources other than those listed above. The number is implementation-specific and is determined by examining the MCG_CAP register (see "Machine-Check Global-Capabilities Register" on page 306). For further information, see the documentation for particular implementations of the architecture.

9.3 Machine Check MSRs

The AMD64 architecture defines a set of model-specific registers (MSRs) in support of the machine-check mechanism. These registers include:

- Global-status and global-control registers:
  - Machine-check global-capabilities register (MCG_CAP).
  - Machine-check global-status register (MCG_STATUS).
  - Machine-check global-control register (MCG_CTL).
- Error-reporting register banks, each containing:
  - Machine-check control register (MC*i*_CTL).
  - Machine-check status register (MC*i*_STATUS).
  - Machine-check address register (MC*i*_ADDR).
  - Machine-check miscellaneous error-information register (MC*i*_MISC).

  Each error-reporting register bank is associated with a specific processor unit (or group of processor units).

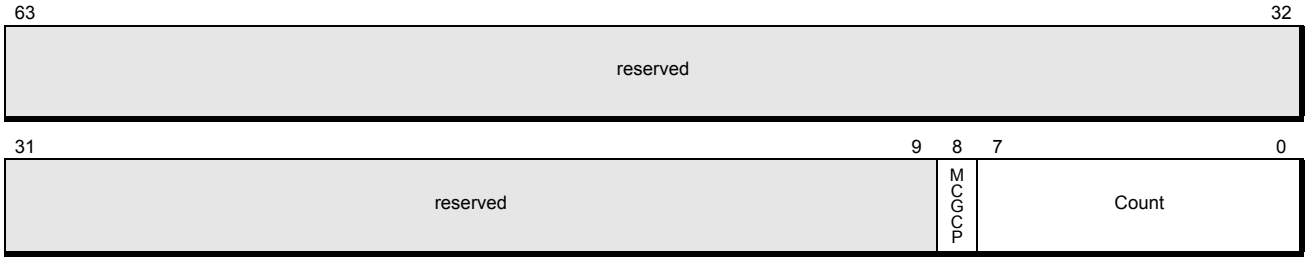
In some cases, the machine-check handler cannot be invoked due to an error, the error-reporting registers retain their values through a warm reset. (A warm reset is a reset that occurs while the processor is powered up, as opposed to a cold reset, which occurs during power-up.) This allows BIOS or other system-boot software to recover and report information associated with the error.

The RDMSR and WRMSR instructions are used to read and write the machine-check MSRs. See "Machine-Check MSRs" on page 441 for a listing of the machine-check MSR numbers and their reset values. The following sections describe each machine-check MSR and its function.

9.3.1 Global Status and Control Registers

The global-status and global-control MSRs supported by the machine-check mechanism include the MCG_CAP, MCG_STATUS, and MCG_CTL registers.

Machine-Check Global-Capabilities Register. Figure 9-1 shows the format of the machine-check global-capabilities register (MCG_CAP). MCG_CAP is a read-only register that specifies the machine-check mechanism capabilities supported by the processor implementation.



Bits	Mnemonic	Description	R/W
63–9	reserved		
8	MCG_CTL_P	MCG_CTL_PMCG_CTL register present	R
7-0	Count	Number of reporting banks	R

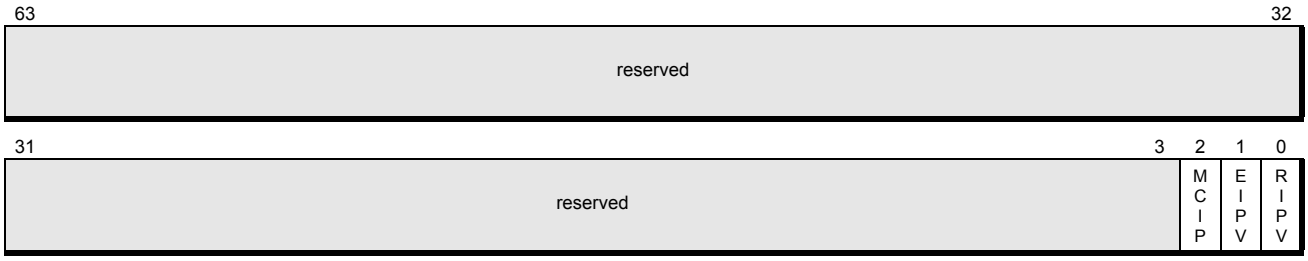
Figure 9-1. MCG_CAP Register

The fields within the MCG_CAP register are:

- *Count*—Bits 7–0. This field specifies how many error-reporting register banks are supported by the processor implementation.
- *MCG_CTL Register Present (MCG_CTL_P)*—Bit 8. This bit specifies whether or not the MCG_CTL register is supported by the processor. When the bit is set to 1, the register is supported. When the bit is cleared to 0, the register is unsupported.

All remaining bits in the MCG_CAP register are reserved. Writing values to the MCG_CAP register produces undefined results.

Machine-Check Global-Status Register. Figure 9-2 on page 307 shows the format of the machine-check global-status register (MCG_STATUS). MCG_STATUS provides basic information about the processor state after the occurrence of a machine-check error.



Bits	Mnemonic	Description	R/W
63–3	reserved		
2	MCIP	Machine Check In-Progress	R/W
1	EIPV	Error IP Valid Flag	R/W
0	RIPV	Restart IP Valid Flag	R/W

Figure 9-2. MCG_STATUS Register

The fields within the MCG_STATUS register are:

- *Restart-IP Valid (RIPV)*—Bit 0. When this bit is set to 1, the interrupted program can be reliably restarted at the instruction addressed by the instruction pointer pushed onto the stack by the machine-check error mechanism. If this bit is cleared to 0, the interrupted program cannot be reliably restarted.
- *Error-IP Valid (EIPV)*—Bit 1. When this bit is set to 1, the instruction that is referenced by the instruction pointer pushed onto the stack by the machine-check error mechanism is responsible for the machine-check error. If this bit is cleared to 0, it is possible that the instruction referenced by the instruction pointer is not responsible for the machine-check error.
- *Machine Check In-Progress (MCIP)*—Bit 2. When this bit is set to 1, it indicates that a machine-check error is in progress. If another machine-check error occurs while this bit is set, the processor enters the shutdown state. The processor sets this bit whenever a machine check exception is generated. Software is responsible for clearing it after the machine check exception is handled.

All remaining bits in the MCG_STATUS register are reserved.

Machine-Check Global-Control Register. Figure 9-3 on page 308 shows the format of the machine-check global-control register (MCG_CTL). MCG_CTL is used by software to control reporting machine-check errors from various sources. Each error-reporting register bank supported by the processor is

controlled by a corresponding enable bit in this register. Setting all bits to 1 in this register enables all error-reporting register banks. The number of controls and how they are used is implementation-specific (for further information, see the documentation for particular implementations of the architecture). The presence of the MCG_CTL register is indicated by the MCG_CAP register MCG_CTL_P bit, described on page 306.

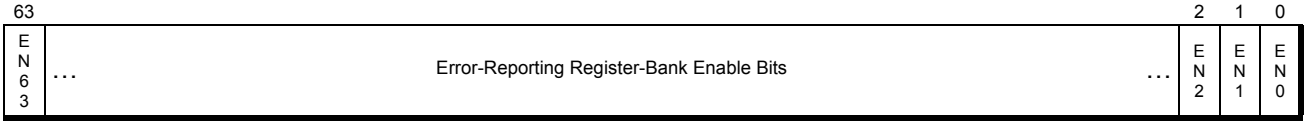


Figure 9-3. MCG_CTL Register

9.3.2 Error-Reporting Register Banks

Error-reporting register banks contain the following registers:

- Machine-check control registers (MC*i*_CTL).
- Machine-check status register (MC*i*_STATUS).
- Machine-check address register (MC*i*_ADDR).
- Machine-check miscellaneous error-information register (MC*i*_MISC).

The *i* in each register name corresponds to the number of a supported register bank. Each error-reporting register bank is associated with a specific processor unit (or group of processor units). The number of error-reporting register banks is implementation-specific. For more information, see the *BIOS and Kernel Develop's Guide for AMD Athlon™ 64 and AMD Opteron™ Processors (order# 26094)* for particular implementations of the AMD64 architecture.

Software reads the MCG_CAP register to determine the number of supported register banks. The first error-reporting register (MC0_CTL) always starts with MSR address 400h, followed by MC0_STATUS (401h), MC0_ADDR (402h), and MC0_MISC (403h). Error-reporting-register MSR addresses are assigned sequentially through the remaining supported register banks. Using this information, software can access all error-reporting registers in an implementation-independent manner.

Machine-Check Control Registers. The machine-check control registers (MC*i*_CTL), as shown in Figure 9-4 on page 309, contain an enable bit for each error source within an error-

reporting register bank. Setting an enable bit to 1 enables error-reporting for the specific feature controlled by the bit, and clearing the bit to 0 disables error reporting for the feature. For more information, see the *BIOS and Kernel Developer's Guide for the AMD Athlon™ 64 and AMD Opteron™ Processors (order# 26094)* for particular implementations of the AMD64 architecture.

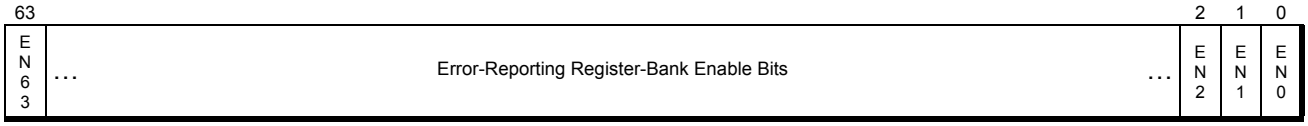
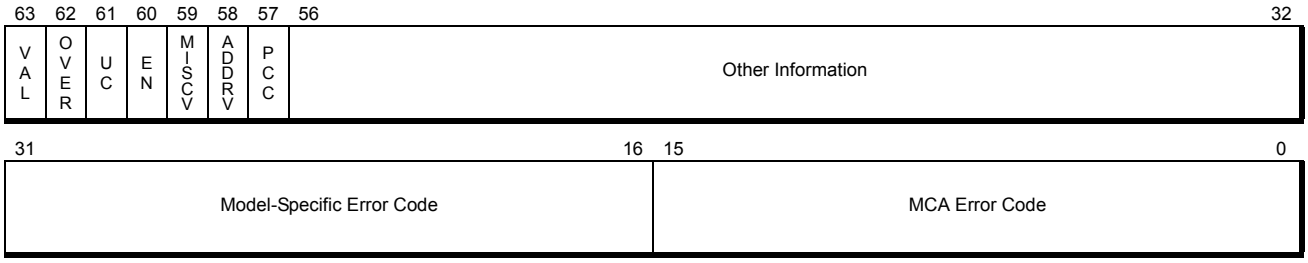


Figure 9-4. MC*i*_CTL Registers

Machine-Check Status Registers. Each error-reporting register bank includes a machine-check status register (MC*i*_STATUS) that the processor uses to report machine-check error information. The machine-check mechanism writes the status-register bits when an error is detected, and sets the valid bit in the register (bit 63) to 1, indicating that the status information is valid. Error reporting for the detected error *does not* need to be enabled for the processor to write the status register. Error reporting must be enabled for the error to result in a machine-check exception. Software is responsible for clearing the status register after the exception has been handled. Attempting to write a value other than 0 to an MC*i*_STATUS register will raise a general protection (#GP) exception.

Figure 9-5 shows the format of the MC*i*_STATUS register.



Bits	Mnemonic	Description	R/W
63	VAL	Valid	R/W*
62	OVER	Status Register Overflow	R/W*
61	UC	Uncorrected Error	R/W*
60	EN	Error Condition Enabled	R/W*
59	MISCV	Miscellaneous-Error Register Valid	R/W*
58	ADDRV	Error-Address Register Valid	R/W*
57	PCC	Processor-Context Corrupt	R/W*
56–32		Other Information	R/W*
31–16		Model-Specific Error Code	R/W*
15–0		MCA Error Code	R/W*

*Applications can only clear this bit to 0.

Figure 9-5. MC*i*_STATUS Register

9.3.3 Error Codes

When a machine-check error occurs, the processor loads an error code into the appropriate MC*i*_STATUS register MCA error-code field. The MC*i*_STATUS.VAL bit is set to 1, indicating that the MC*i*_STATUS register contents are valid. The machine-check mechanism also allows the processor to load a model-specific error code into the MC*i*_STATUS register.

MCA error-codes are used to report errors in the memory hierarchy, the system bus, and the system-interconnection logic. Error-codes are divided into subfields that are used to describe the cause of an error. The information is implementation specific. It includes the location in the memory hierarchy where the error occurred and the type of transaction that caused the error. For further information, see the documentation for particular implementations of the architecture.

The fields within the MC*i*_STATUS register are:

- *MCA Error Code*—Bits 15–0. This field encodes information about the error, including:
  - The type of transaction that caused the error.
  - The memory-hierarchy level involved in the error.
  - The type of request that caused the error.
  - Other information concerning the transaction type.

  See “Error Codes” on page 310 for information on the format and encoding of the MCA error code.

- *Model-Specific Error Code*—Bits 31–16. This field encodes model-specific information about the error. For further information, see the documentation for particular implementations of the architecture.
- *Other Information*—Bits 56–32. This field holds model-specific error information. Software should not rely on the field definitions being consistent between processor implementations. Presently, the bits in this field are defined as:
  - Bits 44–32—Reserved.
  - Bit 45—When set to 1, this bit indicates the error is an uncorrectable ECC error.
  - Bit 46—When set to 1, this bit indicates the error is a correctable ECC error.
  - Bits 54–47—This field holds the ECC syndrome when an ECC error occurs.
  - Bits 56–55—Reserved.
- *PCC*—Bit 57. When set to 1, this bit indicates that the processor state is likely to be corrupt due to the machine-check error. In this case, it is possible software cannot restart the processor reliably. When this bit is cleared to 0, the processor state is not corrupted by the machine-check error. If the PCC bit is set in any error bank, the processor will clear RIPV and EIPV in the MCG_Status register.
- *ADDRV*—Bit 58. When set to 1, this bit indicates that the address saved in the corresponding error-reporting address register (MC*i*_ADDR) is valid, and contains the address where the error was detected. When this bit is cleared to 0, MC*i*_ADDR does not contain a valid error address.
- *MISCV*—Bit 59. When set to 1, this bit indicates that additional information about the machine-check error is saved in the corresponding error-reporting miscellaneous register (MC*i*_MISC). This bit is cleared to 0 when the MC*i*_MISC registers are not implemented.
- *EN*—Bit 60. When set to 1, this bit indicates that the error condition is enabled in the corresponding error-reporting control register (MC*i*_CTL). Errors disabled by MC*i*_CTL do not cause a machine-check exception, but the machine-check mechanism can log errors when error reporting is disabled in MC*i*_CTL.

- *UC*—Bit 61. When set to 1, this bit indicates that the processor did not correct the error condition. When this bit is cleared to 0, the processor corrected the error condition.
- *OVER*—Bit 62. This bit is set to 1 by the processor if the VAL bit is already set to 1 as the processor attempts to load error information into MC*i*_STATUS. This indicates that the results of a previous machine-check error are still in the MC*i*_STATUS register. In this situation, the machine-check mechanism handles the contents of MC*i*_STATUS as follows:
  - Status for an enabled error replaces status for a disabled error.
  - Status for an uncorrectable error replaces status for a correctable error.
  - Status for an enabled uncorrectable error is never replaced.
- *VAL*—Bit 63. This bit is set to 1 by the processor if the contents of MC*i*_STATUS are valid. Software should clear the VAL bit after reading the MC*i*_STATUS register, otherwise a subsequent machine-check error sets the OVER bit as described above.

Machine-Check Address Registers. Each error-reporting register bank includes a machine-check address register (MC*i*_ADDR) that the processor uses to report the address associated with the machine-check error. The address field can hold either a virtual (linear) or physical address, depending on the type of error. For further information, see the documentation for particular implementations of the architecture. The contents of this register are valid only if the ADDRV bit in the corresponding MC*i*_STATUS register is set to 1.

Machine-Check Miscellaneous-Error Information Registers. Each error-reporting register bank can include a machine-check miscellaneous error-information register (MC*i*_MISC) that the processor uses to report additional information concerning the machine-check error. The contents of this register are valid only if the MISCV bit in the corresponding MC*i*_STATUS register is set to 1. The miscellaneous error information reported in this register is implementation dependent.

9.4 Initializing the Machine-Check Mechanism

Following a processor reset, all machine-check error-reporting enable bits are disabled. System software must enable these bits before machine-check errors can be reported. Generally, system software should initialize the machine-check mechanism using the following process:

- Execute the CPUID instruction and verify that the processor supports the machine-check exception (MCE) and machine-check registers (MCA). MCE is supported when EDX bit 7 is set to 1, and MCA is supported when EDX bit 14 is set to 1. Software should not proceed with initializing the machine-check mechanism if the machine-check registers are not supported.
- If the machine-check registers are supported, system software should take the following steps:
  - Check to see if the MCG_CTL_P bit in the MCG_CAP register is set to 1. If it is, then the MCG_CTL register is supported by the processor. If the MCG_CTL register is supported, software should set its enable bits to 1 for the machine-check features it uses. Software can load MCG_CTL with all 1s to enable all machine-check features.
  - Read the COUNT field from the MCG_CAP register to determine the number of error-reporting register banks supported by the processor. For each error-reporting register bank, software should set the enable bits to 1 in the MC*i*_CTL register for the error types it wants the processor to report. Software can load each MC*i*_CTL with all 1s to enable all error-reporting mechanisms.

    The error-reporting register banks are numbered from 0 to one less than the value found in the MCG_CAP.COUNT field. For example, if the COUNT field indicates five register banks are supported, they are numbered 0 to 4.
  - For each error-reporting register bank, software should clear all status fields in the MC*i*_STATUS register by writing all 0s to the register.

    It is possible that valid error-status is already reported by the MC*i*_STATUS registers at the time software clears them. The status can reflect fatal errors recorded before a warm reset, or errors recorded during the system power-up and boot process. Before clearing the MC*i*_STATUS

registers, software should examine their contents and log any errors found.

- As a final step in the initialization process, system software should enable the machine-check exception by setting CR4.MCE (bit 6) to 1.

9.5 Using Machine Check Features

System software can detect and handle machine-check errors using two methods:

- Software can periodically examine the machine-check status registers for reported errors, and log any errors found.
- Software can enable the machine-check exception (#MC). When an uncorrectable error occurs, the processor immediately transfers control to the machine-check exception handler. In this case, system software provides a machine-check exception handler that, at a minimum, logs detected errors. The exception handler can be designed for a specific processor implementation or can be generalized to work on multiple implementations.

9.5.1 Handling Machine Check Exceptions

The processor uses the interrupt control-transfer mechanism to invoke an exception handler after a machine-check exception occurs. This requires system software to initialize the interrupt-descriptor table (IDT) with either an interrupt gate or a trap gate that references the interrupt handler. See “Legacy Protected-Mode Interrupt Control Transfers” on page 274 and “Long-Mode Interrupt Control Transfers” on page 285 for more information on interrupt control transfers.

At a minimum, the machine-check exception handler must be capable of logging errors for later examination. Because most machine-check errors are single-bit DRAM-ECC errors, which are hardware recoverable, this can be a sufficient implementation for some handlers. More thorough exception-handler implementations can analyze the error to determine if it is unrecoverable, and whether it can be recovered in software.

Machine-check exception handlers that attempt to correct unrecoverable errors must be thorough in their analysis and their corrective actions. The following guidelines should be used when writing such a handler:

- All status registers in the error-reporting register banks must be examined to identify the cause or causes of the machine-check exception. Read the COUNT field from MCG_CAP to determine the number of status registers supported by the processor. The status registers are numbered from 0 to one less than the value found in the MCG_CAP.COUNT field. For example, if the COUNT field indicates five status registers are supported, they are named MC0_STATUS to MC4_STATUS.
- Check the valid bit in each status register (MC*i*_STATUS.VAL). The MC*i*_STATUS register does not need to be examined when its valid bit is clear.
- Check the valid MC*i*_STATUS registers to see if error recovery is possible. Error recovery is not possible when:
  - The processor-context corrupt bit (MC*i*_STATUS.PCC) is set to 1.
  - The error-overflow status bit (MC*i*_STATUS.OVER) is set to 1. This bit indicates that more than one machine-check error occurred, but only one error is reported by the status register.

  If error recovery is not possible, the handler should log the error information and return to the operating system.
- Check the MC*i*_STATUS.UC bit to see if the processor corrected the error. If UC=1, the processor did not correct the error, and the exception handler must correct the error before restarting the interrupted program. If the handler cannot correct the error, it should log the error information and return to the operating system.
- When identifying the error condition, portable exception handlers should examine only the MC*i*_STATUS register MCA error-code field. See “Error Codes” on page 310 for information on interpreting this field.
- If the MCG_STATUS.RIPV bit is set to 1, the interrupted program can be restarted reliably at the instruction-pointer address pushed onto the exception-handler stack. If RIPV=0, the interrupted program cannot be restarted reliably at that location, although it can be restarted at that location for debugging purposes.
- When logging errors, particularly those that are not recoverable, check the MCG_STATUS.EIPV bit to see if the instruction-pointer address pushed onto the exception-

handler stack is related to the machine-check error. If EIPV=0, the address is not guaranteed to be related to the error.

- Before exiting the machine-check handler, be sure to clear MCG_STATUS.MCIP to 0. MCIP indicates a machine-check exception occurred. If this bit is set when another machine-check exception occurs, the processor enters the shutdown state.
- When an exception handler is able to, at a minimum, successfully log an error condition, the MC*i*_STATUS registers should be cleared to 0 before exiting the machine-check handler. Software is responsible for clearing at least the MC*i*_STATUS.VAL bits.
- Additional machine-check exception-handler portability can be added by having the handler use the CPUID instruction to identify the processor and its capabilities. Implementation-specific software can be added to the machine-check exception handler based on the processor information reported by CPUID.

9.5.2 Reporting Correctable Machine Check Errors

Machine-check exceptions do not occur if the error is correctable by the processor. If system software wishes to log and report correctable machine-check errors, a system-service routine must be provided to check the contents of the machine-check status registers for correctable errors. The service routine can be invoked by system software on a periodic basis, or it can be manually invoked by the user as needed.

If the processor supports the machine-check registers, a service routine that reports correctable errors should perform the following:

- Examine each status register (MC*i*_STATUS) in the error-reporting register banks. For each MC*i*_STATUS register with a set valid bit (VAL=1), the service routine should:
  - Save the contents of the MC*i*_STATUS register.
  - Save the contents of the corresponding MC*i*_ADDR register if MC*i*_STATUS.ADDRV=1.
  - Save the contents of the corresponding MC*i*_MISC register if MC*i*_STATUS.MISCV=1.
  - Check to see if MCG_STATUS.MCIP=1, which indicates that the machine-check exception handler is in progress. If this is the case, then the machine-check exception

handler has called the service routine to log the errors. In this situation, the error-logging service routine should determine whether or not the interrupted program is restartable, and report the determination back to the exception handler. The program is *not restartable* if either of the following is true:

- MC*i*_STATUS.PCC=1, which indicates the processor context is corrupted, or
- MCG_STATUS.RIPV=0, which indicates the interrupted program cannot be restarted reliably at the instruction-pointer address pushed onto the exception-handler stack.

- Once the information found in the error-reporting register banks is saved, the MC*i*_STATUS register should be cleared to 0. This allows the processor to properly report any subsequent errors in the MC*i*_STATUS registers.
- The service routine can save the time-stamp counter with each error logged. This can help in determining how frequently errors occur. For further information, see "Time-Stamp Counter" on page 406.
- In multiprocessor configurations, the service routine can save the processor-node identifier. This can help locate a failing multiprocessor-system component, which can then be isolated from the rest of the system. For further information, see the documentation for particular implementations of the architecture.

10 System-Management Mode

System-management mode (SMM) is an operating mode designed for system-control activities like power management. Normally, these activities are transparent to conventional operating systems and applications. SMM is used by system-specific BIOS (basic input-output system) and specialized low-level device drivers, rather than the operating system.

The SMM interrupt-handling mechanism differs substantially from the standard interrupt-handling mechanism described in Chapter 8, "Exceptions and Interrupts." SMM is entered using a special external interrupt called the *system-management interrupt* (SMI). After an SMI is received by the processor, the processor saves the processor state in a separate address space, called *SMRAM*. The SMM-handler software and data structures are also located in the SMRAM space. Interrupts and exceptions that ordinarily cause control transfers to the operating system are disabled when SMM is entered. The processor exits SMM, restores the saved processor state, and resumes normal execution by using a special instruction, *RSM*.

In SMM, address translation is disabled and addressing is similar to real mode. SMM programs can address up to 4 Gbytes of physical memory. See "SMM Operating-Environment" on page 329 for additional information on memory addressing in SMM.

The following sections describe the components of the SMM mechanism:

- *"SMM Resources" on page 320*—this section describes SMRAM, the SMRAM save-state area used to hold the processor state, and special SMRAM save-state entries used in support of SMM.
- *"Using SMM" on page 329*—this section describes the mechanism of entering and exiting SMM. It also describes SMM memory allocation, addressing, and interrupts and exceptions.

Of these mechanisms, only the format of the SMRAM save-state area differs between the AMD64 architecture and the legacy architecture.

10.0.1 SMM Differences

There are functional differences between the SMM support in the AMD64 architecture and the SMM support found in previous architectures. These are:

- The SMRAM state-save area layout is changed to hold the 64-bit processor state.
- 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 in SMM.
- The auto-halt restart and I/O-instruction restart entries in the SMRAM state-save area are one byte each instead of two bytes each.
- 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.
- Some previous AMD x86 processors saved and restored the CR2 register in the SMRAM state-save area. This register is not saved by the SMM implementation in the AMD64 architecture. SMM handlers that save and restore CR2 must perform the operation in software.

10.1 SMM Resources

The SMM resources supported by the processor consist of SMRAM, the SMRAM state-save area, and special entries within the SMRAM state-save area. In addition to the save-state area, SMRAM includes space for the SMM handler.

10.1.1 SMRAM

SMRAM is the memory-address space accessed by the processor when in SMM. The default size of SMRAM is 64 Kbytes and can range in size between 32 Kbytes and 4 Gbytes. System logic can use physically separate SMRAM and main memory, directing memory transactions to SMRAM after recognizing SMM is entered, and redirecting memory transactions back to system memory after recognizing SMM is exited. When separate SMRAM and main memory are used, the system designer needs to provide a method of mapping SMRAM into main memory so that the SMI handler and data structures can be loaded.

Figure 10-1 on page 321 shows the default SMRAM memory map. The default SMRAM code-segment (CS) has a base

address of 0003_0000h (the base address is automatically scaled by the processor using the CS-selector register, which is set to the value 3000h). This default SMRAM-base address is known as *SMBASE*. A 64-Kbyte memory region, addressed from 0003_0000h to 0003_FFFFh, makes up the default SMRAM memory space. The top 32 Kbytes (0003_8000h to 0003_FFFFh) must be supported by system logic, with physical memory covering that entire address range. The top 512 bytes (0003_FE00h to 0003_FFFFh) of this address range are the default SMM state-save area. The default entry point for the SMM interrupt handler is located at 0003_8000h.

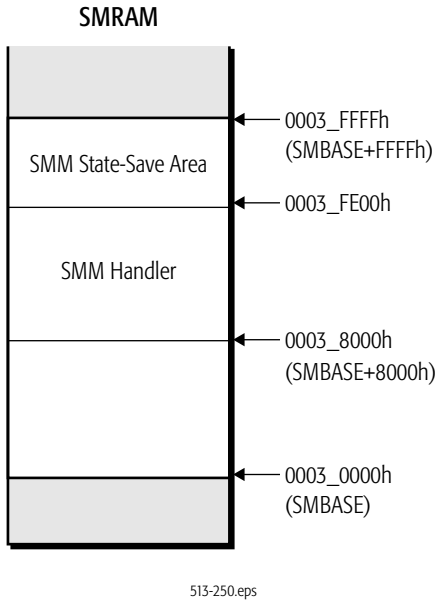


Figure 10-1. Default SMRAM Memory Map

10.1.2 SMBASE Register

The format of the SMBASE register is shown in Figure 10-2 on page 322. SMBASE is an internal processor register that holds the value of the SMRAM-base address. SMBASE is set to 30000h after a processor reset.

Figure 10-2. SMBASE Register

In some operating environments, relocation of SMRAM to a higher memory area can provide more low memory for legacy software. SMBASE relocation is supported when the SMM-base relocation bit in the SMM-revision identifier (bit 17) is set to 1. In processors implementing the AMD64 architecture, SMBASE relocation is always supported.

Software can only modify SMBASE (relocate the SMRAM-base address) by entering SMM, modifying the SMBASE image stored in the SMRAM state-save area, and exiting SMM. The SMM-handler entry point must be loaded at the new memory location specified by SMBASE+8000h. The next time SMM is entered, the processor saves its state in the new state-save area at SMBASE+0FE00h, and begins executing the SMM handler at SMBASE+8000h. The new SMBASE address is used for every SMM until it is changed, or a hardware reset occurs.

When SMBASE is used to relocate SMRAM to an address above 1 Mbyte, 32-bit address-size-override prefixes must be used to access this memory. This is because addressing in SMM behaves as it does in real mode, with a 16-bit default operand size and address size. The values in the 16-bit segment-selector registers are left-shifted four bits to form a 20-bit segment-base address. Without using address-size overrides, the maximum address that can be computed is 10FFEh.

Because SMM memory-addressing is similar to real-mode addressing, the SMBASE address must be less than 4 Gbytes. Physical-address extensions (CR4.PAE) should not be enabled in SMM, restricting the SMRAM address space to the range 0h to 0FFFF_FFFFh.

10.1.3 SMRAM State-Save Area

When an SMI occurs, the processor saves its state in the 512-byte SMRAM state-save area during the control transfer into SMM. The format of the state-save area defined by the AMD64

architecture is shown in Table 10-1. This table shows the offsets in the SMRAM state-save area relative to the SMRAM-base address. The state-save area is located between offset 0_FE00h (SMBASE+0_FE00h) and offset 0_FFFFh (SMBASE+0_FFFFh). Software should not modify offsets specified as read-only or reserved, otherwise unpredictable results can occur.

Table 10-1. AMD64 Architecture SMM State-Save Area

Offset (Hex) from SMBASE	Contents		Size	Allowable Access
FE00h	ES	Selector	Word	Read-Only
FE02h		Attributes	Word	
FE04h		Limit	Doubleword	
FE08h		Base	Quadword	
FE10h	CS	Selector	Word	Read-Only
FE12h		Attributes	Word	
FE14h		Limit	Doubleword	
FE18h		Base	Quadword	
FE20h	SS	Selector	Word	Read-Only
FE22h		Attributes	Word	
FE24h		Limit	Doubleword	
FE28h		Base	Quadword	
FE30h	DS	Selector	Word	Read-Only
FE32h		Attributes	Word	
FE34h		Limit	Doubleword	
FE38h		Base	Quadword	
FE40h	FS	Selector	Word	Read-Only
FE42h		Attributes	Word	
FE44h		Limit	Doubleword	
FE48h		Base	Quadword	

Note:

1. *The offset for the SMM-revision identifier is compatible with previous implementations.*

Table 10-1. AMD64 Architecture SMM State-Save Area (continued)

Offset (Hex) from SMBASE	Contents		Size	Allowable Access
FE50h	GS	Selector	Word	Read-Only
FE52h		Attributes	Word	
FE54h		Limit	Doubleword	
FE58h		Base	Quadword	
FE60h–FE61h	GDTR	reserved	2 Bytes	Read-Only
FE62h		reserved	Word	
FE64h		Limit	Word	
FE66h–FE67h		reserved	2 Bytes	
FE68h		Base	Quadword	
FE70h	LDTR	Selector	Word	Read-Only
FE72h		Attributes	Word	
FE74h		Limit	Doubleword	
FE78h		Base	Quadword	
FE80h–FEB1h	IDTR	reserved	2 Bytes	Read-Only
FE82h		reserved	Word	
FE84h		Limit	Word	
FEB6h–FEB7h		reserved	2 Bytes	
FE88h		Base	Quadword	
FE90h	TR	Selector	Word	Read-Only
FE92h		Attributes	Word	
FE94h		Limit	Doubleword	
FE98h		Base	Quadword	
FEA0h–FEC7h	reserved		40 Bytes	–
FEC8h	I/O Instruction Restart		Byte	Read/Write
FEC9h	Auto-Halt Restart		Byte	

Note:

1. *The offset for the SMM-revision identifier is compatible with previous implementations.*

Table 10-1. AMD64 Architecture SMM State-Save Area (continued)

Offset (Hex) from SMBASE	Contents	Size	Allowable Access
FECAh–FECFh	reserved	6 Bytes	–
FED0h	EFER	Quadword	Read-Only
FED8h–FEFBh	reserved	36 Bytes	–
FEFCh	SMM-Revision Identifier[1]	Doubleword	Read-Only
FF00h	SMBASE	Doubleword	Read/Write
FF04h–FF47h	reserved	68 Bytes	–
FF48h	CR4	Quadword	Read-Only
FF50h	CR3	Quadword	
FF58h	CR0	Quadword	
FF60h	DR7	Quadword	Read-Only
FF68h	DR6	Quadword	
FF70h	RFLAGS	Quadword	Read/Write
FF78h	RIP	Quadword	Read/Write
FF80h	R15	Quadword	
FF88h	R14	Quadword	
FF90h	R13	Quadword	
FF98h	R12	Quadword	
FFA0h	R11	Quadword	
FFA8h	R10	Quadword	
FFB0h	R9	Quadword	
FFB8h	R8	Quadword	

Note:

1. *The offset for the SMM-revision identifier is compatible with previous implementations.*

Table 10-1. AMD64 Architecture SMM State-Save Area (continued)

Offset (Hex) from SMBASE	Contents	Size	Allowable Access
FFC0h	RDI	Quadword	Read/Write
FFC8h	RSI	Quadword	
FFD0h	RBP	Quadword	
FFD8h	RSP	Quadword	
FFE0h	RBX	Quadword	
FFE8h	RDX	Quadword	
FFF0h	RCX	Quadword	
FFF8h	RAX	Quadword	

Note:
1. The offset for the SMM-revision identifier is compatible with previous implementations.

A number of other registers are not saved or restored automatically by the SMM mechanism. See "Saving Additional Processor State" on page 332 for information on using these registers in SMM.

As a reference for legacy processor implementations, the legacy SMM state-save area format is shown in Table 10-2. *Implementations of the AMD64 architecture do not use this format.*

Table 10-2. Legacy SMM State-Save Area (Not used by AMD64 Architecture)

Offset (Hex) from SMBASE	Contents	Size	Allowable Access
FE00h–FEF7h	reserved	248 Bytes	–
FEF8h	SMBASE	Doubleword	Read/Write
FEFCh	SMM-Revision Identifier	Doubleword	Read-Only
FF00h	I/O Instruction Restart	Word	Read/Write
FF02h	Auto-Halt Restart	Word	

Note:
1. The offset for the SMM-revision identifier is compatible with previous implementations.

Table 10-2. Legacy SMM State-Save Area (continued)**(Not used by AMD64 Architecture)**

Offset (Hex) from SMBASE	Contents	Size	Allowable Access
FF04h—FF87h	reserved	132 Bytes	—
FF88h	GDT Base	Doubleword	Read-Only
FF8Ch—FF93h	reserved	Quadword	—
FF94h	IDT Base	Doubleword	Read-Only
FF98h—FFA7h	reserved	16 Bytes	—
FFA8h	ES	Doubleword	Read-Only
FFACh	CS	Doubleword	
FFB0h	SS	Doubleword	
FFB4h	DS	Doubleword	
FFB8h	FS	Doubleword	
FFBCh	GS	Doubleword	
FFC0h	LDT Base	Doubleword	Read-Only
FFC4h	TR	Doubleword	
FFC8h	DR7	Doubleword	Read-Only
FFCCh	DR6	Doubleword	
FFD0h	EAX	Doubleword	Read/Write
FFD4h	ECX	Doubleword	
FFD8h	EDX	Doubleword	
FFDCh	EBX	Doubleword	
FFE0h	ESP	Doubleword	
FFE4h	EBP	Doubleword	
FFE8h	ESI	Doubleword	
FFECh	EDI	Doubleword	
FFF0h	EIP	Doubleword	Read/Write

Note:

1. The offset for the SMM-revision identifier is compatible with previous implementations.

Table 10-2. Legacy SMM State-Save Area (continued)**(Not used by AMD64 Architecture)**

Offset (Hex) from SMBASE	Contents	Size	Allowable Access
FFF4h	EFLAGS	Doubleword	Read/Write
FFF8h	CR3	Doubleword	Read-Only
FFFCh	CR0	Doubleword	

Note:
1. The offset for the SMM-revision identifier is compatible with previous implementations.

10.1.4 SMM-Revision Identifier

The SMM-revision identifier specifies the SMM version and the available SMM extensions implemented by the processor. Software reads the SMM-revision identifier from offset FEFCh in the SMM state-save area of SMRAM. This offset location is compatible with earlier versions of SMM. Software must not write to this location. Doing so can produce undefined results. Figure 10-3 shows the format of the SMM-revision identifier.

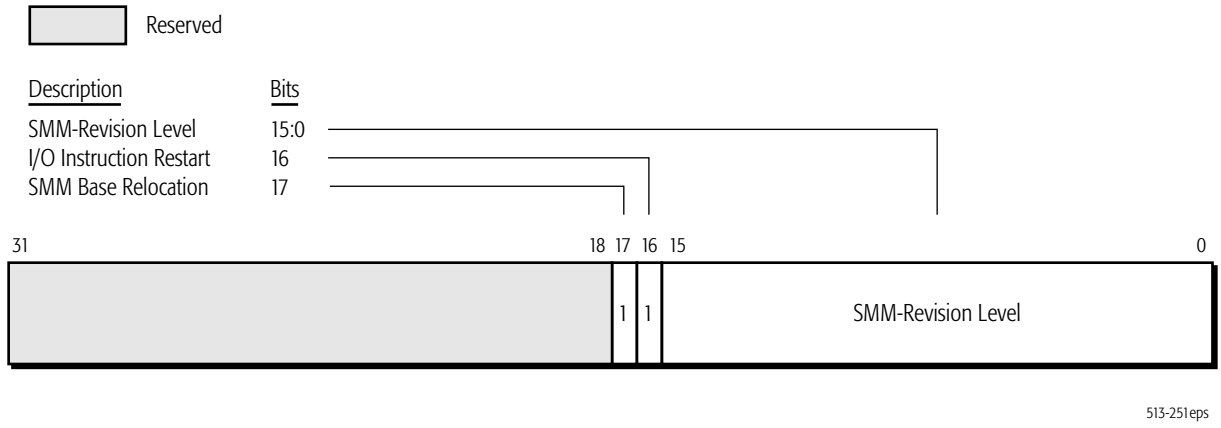


Figure 10-3. SMM-Revision Identifier

The fields within the SMM-revision identifier are:

- *SMM-revision Level*—Bits 15–0. Specifies the version of SMM supported by the processor. The SMM-revision level is of the form 0_xx64h, where *xx* starts with 00 and is incremented for later revisions to the SMM mechanism.

- *I/O Instruction Restart*—Bit 16. When set to 1, the processor supports restarting I/O instructions that are interrupted by an SMI. This bit is always set to 1 by implementations of the AMD64 architecture. See "I/O Instruction Restart" on page 334 for information on using this feature.
- *SMM Base Relocation*—Bit 17. When set to 1, the processor supports relocation of SMRAM. This bit is always set to 1 by implementations of the AMD64 architecture. See "SMBASE Register" on page 321 for information on using this feature.

All remaining bits in the SMM-revision identifier are reserved.

10.2 Using SMM

10.2.1 System-Management Interrupt (SMI)

SMM is entered using the system-management interrupt (SMI). SMI is an external non-maskable interrupt that operates differently from and independently of other interrupts. SMI has priority over all other external interrupts, including NMI (see "Priorities" on page 267 for a list of the interrupt priorities). SMIs are disabled when in SMM, which prevents reentrant calls to the SMM handler.

When an SMI is received by the processor, the processor stops fetching instructions and waits for currently-executing instructions to complete and write their results. The SMI also waits for all buffered memory writes to update the caches or system memory. When these activities are complete, the processor uses implementation-dependent external signalling to acknowledge back to the system that it has received the SMI.

10.2.2 SMM Operating-Environment

The SMM operating-environment is similar to real mode, except that the segment limits in SMM are 4 Gbytes rather than 64 Kbytes. This allows an SMM handler to address memory in the range from 0h to 0FFFF_FFFFh. As with real mode, segment-base addresses are restricted to 20 bits in SMM, and the default operand-size and address-size is 16 bits. To address memory locations above 1 Mbyte, the SMM handler must use the 32-bit operand-size-override and address-size-override prefixes.

After saving the processor state in the SMRAM state-save area, a processor running in SMM sets the segment-selector registers and control registers into a state consistent with real mode.

Other registers are also initialized upon entering SMM, as shown in Table 10-3.

Table 10-3. SMM Register Initialization

Register		Initial SMM Contents
CS	Selector	SMBASE right-shifted 4 bits
	Base	SMBASE
	Limit	FFFF_FFFFh
DS, ES, FS, GS, SS	Selector	0000h
	Base	0000_0000_0000_0000h
	Limit	FFFF_FFFFh
RIP		0000_0000_0000_8000h
RFLAGS		0000_0000_0000_0002h
CR0		PE, EM, TS, PG bits cleared to 0. All other bits are unmodified.
CR4		0000_0000_0000_0000h
DR7		0000_0000_0000_0400h
EFER		0000_0000_0000_0000h

10.2.3 Exceptions and Interrupts

All hardware interrupts are disabled upon entering SMM, but exceptions and software interrupts are not disabled. If necessary, the SMM handler can re-enable hardware interrupts. Software that handles interrupts in SMM should consider the following:

- *SMI*—If an SMI occurs while the processor is in SMM, it is latched by the processor. The latched SMI occurs when the processor leaves SMM.
- *NMI*—If an NMI occurs while the processor is in SMM, it is latched by the processor, but the NMI handler is not invoked until the processor leaves SMM with the execution of an RSM instruction. A pending NMI causes the handler to be invoked immediately after the RSM completes and before the first instruction in the interrupted program is executed.

  An SMM handler can unmask NMI interrupts by simply executing an IRET. Upon completion of the IRET

instruction, the processor recognizes the pending NMI, and transfers control to the NMI handler. Once an NMI is recognized within SMM using this technique, subsequent NMIs are recognized until SMM is exited. Later SMIs cause NMIs to be masked, until the SMM handler unmasks them.

- *Exceptions*—Exceptions (internal processor interrupts) are not disabled and can occur while in SMM. Therefore, the SMM-handler software should be written to avoid generating exceptions.
- *Software Interrupts*—The software-interrupt instructions (BOUND, INT*n*, INT3, and INTO) can be executed while in SMM. However, it is not recommended that the SMM handler use these instructions.
- *Maskable Interrupts*—RFLAGS.IF is cleared to 0 by the processor when SMM is entered. Software can re-enable maskable interrupts while in SMM, but it must follow the guidelines listed below for handling interrupts.
- *Debug Interrupts*—The processor disables the debug interrupts when SMM is entered by clearing DR7 to 0 and clearing RFLAGS.TF to 0. The SMM handler can re-enable the debug facilities while in SMM, but it must follow the guidelines listed below for handling interrupts.
- *INIT*—The processor does not recognize INIT while in SMM.

Because the RFLAGS.IF bit is cleared when entering SMM, the HLT instruction should not be executed in SMM without first setting the RFLAGS.IF bit to 1. Setting this bit to 1 allows the processor to exit the halt state by using an external maskable interrupt.

In the cases where an SMM handler must accept and handle interrupts and exceptions, several guidelines must be followed:

- Interrupt handlers must be loaded and accessible before enabling interrupts.
- A real-mode interrupt-vector table located at virtual (linear) address 0 is required.
- Segments accessed by the interrupt handler cannot have a base address greater than 20 bits because of the real-mode addressing used in SMM. In SMM, the 16-bit value stored in the segment-selector register is left-shifted four bits to form the 20-bit segment-base address, like real mode.

- Only the IP (rIP[15:0]) is pushed onto the stack as a result of an interrupt in SMM, because of the real-mode addressing used in SMM. If the SMM handler is interrupted at a code-segment offset above 64 Kbytes, then the return address on the stack must be adjusted by the interrupt-handler, and a RET instruction with a 32-bit operand-size override must be used to return to the SMM handler.
- If the interrupt-handler is located below 1 Mbyte, and the SMM handler is located above 1 Mbyte, a RET instruction cannot be used to return to the SMM handler. In this case, the interrupt handler can adjust the return pointer on the stack, and use a far CALL to transfer control back to the SMM handler.

10.2.4 Invalidating the Caches

The processor can cache SMRAM-memory locations. If the system implements physically separate SMRAM and system memory, it is possible for SMRAM and system memory locations to alias into identical cache locations. In some processor implementations, the cache contents must be written to memory and invalidated when SMM is entered *and* exited. This prevents the processor from using previously-cached main-memory locations as aliases for SMRAM-memory locations when SMM is entered, and vice-versa when SMM is exited.

Implementations of the AMD64 architecture *do not require cache invalidation* when entering and exiting SMM. Internally, the processor keeps track of SMRAM and system-memory accesses separately and properly handles situations where aliasing occurs. Cached system memory and SMRAM locations can persist across SMM mode changes. Removal of the requirement to writeback and invalidate the cache simplifies SMM entry and exit and allows SMM code to execute more rapidly.

10.2.5 Saving Additional Processor State

Several registers are not saved or restored automatically by the SMM mechanism. These are:

- The 128-bit media instruction registers.
- The 64-bit media instruction registers.
- The x87 floating-point registers.
- The page-fault linear-address register (CR2).
- The task-priority register (CR8).
- The debug registers, DR0, DR1, DR2, and DR3.

- The memory-type range registers (MTRRs).
- Model-specific registers (MSRs).

These registers are not saved because SMM handlers do not normally use or modify them. If an SMI results in a processor reset (due to powering down the processor, for example) or the SMM handler modifies the contents of the unsaved registers, the SMM handler should save and restore the original contents of those registers. The unsaved registers, along with those stored in the SMRAM state-save area, need to be saved in a non-volatile storage location if a processor reset occurs. The SMM handler should execute the CPUID instruction to determine the feature set available in the processor, and be able to save and restore the registers required by those features.

The SMM handler can execute any of the 128-bit media, 64-bit media, or x87 instructions. A simple method for saving and restoring those registers is to use the FXSAVE and FXRSTOR instructions, respectively, if it is supported by the processor. See “Saving Media and x87 Processor State” on page 344 for information on saving and restoring those registers.

Floating-point exceptions can occur when the SMM handler uses media or x87 floating-point instructions. If the SMM handler uses floating-point exception handlers, they must follow the usage guidelines established in “Exceptions and Interrupts” on page 330. A simple method for dealing with floating-point exceptions while in SMM is to simply mask all exception conditions using the appropriate floating-point control register. When the exceptions are masked, the processor handles floating-point exceptions internally in a default manner, and allows execution to continue uninterrupted.

10.2.6 Operating in Protected Mode and Long Mode

Software can enable protected mode from SMM and it can also enable and activate long mode. An SMM handler can use this capability to enter 64-bit mode and save additional processor state that cannot be accessed from outside 64-bit mode (for example, the most-significant 32 bits of CR2).

10.2.7 Auto-Halt Restart

The auto-halt restart entry is located at offset FEC9h in the SMM state-save area. The size of this field is one byte, as compared with two bytes in previous versions of SMM.

When entering SMM, the processor loads the auto-halt restart entry to indicate whether SMM was entered from the halt state, as follows:

- Bit 0 indicates the processor state upon entering SMM:
  - When set to 1, the processor entered SMM from the halt state.
  - When cleared to 0, the processor did not enter SMM from the halt state.
- Bits 7–1 are cleared to 0.

The SMM handler can write the auto-halt restart entry to specify whether the return from SMM should take the processor back to the halt state or to the instruction-execution state specified by the SMM state-save area. The values written are:

- *Clear to 00h*—The processor returns to the state specified by the SMM state-save area.
- *Set to any non-zero value*—The processor returns to the halt state.

If the return from SMM takes the processor back to the halt state, the HLT instruction is not re-executed. However, the halt special bus-cycle is driven on the processor bus after the RSM instruction executes.

The result of entering SMM from a non-halt state and returning to a halt state is not predictable.

10.2.8 I/O Instruction Restart

The I/O-instruction restart entry is located at offset FEC8h in the SMM state-save area. The size of this field is one byte, as compared with two bytes in previous versions of SMM. The I/O-instruction restart mechanism is supported when the I/O-instruction restart bit (bit 16) in the SMM-revision identifier is set to 1. This bit is always set to 1 in the AMD64 architecture.

When an I/O instruction is interrupted by an SMI, the I/O-instruction restart entry specifies whether the interrupted I/O instruction should be re-executed following an RSM that returns from SMM. Re-executing a trapped I/O instruction is useful, for example, when an I/O write is performed to a powered-down disk drive. When this occurs, the system logic monitoring the access can issue an SMI to have the SMM handler power-up the disk drive and retry the I/O write. The SMM handler does this by querying system logic and detecting the failed I/O write, asking system logic to initiate the disk-

drive power-up sequence, enabling the I/O instruction restart mechanism, and returning from SMM. Upon returning from SMM, the I/O write to the disk drive is restarted.

When an SMI occurs, the processor always clears the I/O-instruction restart entry to 0. If the SMI interrupted an I/O instruction, then the SMM handler can modify the I/O-instruction restart entry as follows:

- *Clear to 00h (default value)*—The I/O instruction is not restarted, and the instruction following the interrupted I/O-instruction is executed. When a REP (repeat) prefix is used with an I/O instruction, it is possible that the next instruction to be executed is the next I/O instruction in the repeat loop.
- *Set to any non-zero value*—The I/O instruction is restarted.

While in SMM, the handler must determine the cause of the SMI and examine the processor state at the time the SMI occurred to determine whether or not an I/O instruction was interrupted. Writing a non-zero value into the I/O-instruction restart entry when the interrupted instruction is a non-I/O instruction produces undefined results. Some implementations provide model-specific facilities to assist in this determination. For more information, refer to the BIOS writer's guide for the implementation.

10.3 Leaving SMM

Software leaves SMM and returns to the interrupted program by executing the RSM instruction. RSM causes the processor to load the interrupted state from the SMRAM state-save area and then transfer control back to the interrupted program. RSM cannot be executed in any mode other than SMM, otherwise an invalid-opcode exception (#UD) occurs.

An RSM causes a processor shutdown if an invalid-state condition is found in the SMRAM state-save area. Only an external reset, external processor-initialization, or non-maskable external interrupt (NMI) can cause the processor to leave the shutdown state. The invalid SMRAM state-save-area conditions that can cause a processor shutdown during an RSM are:

- CR0.PE=0 and CR0.PG=1.

- CR0.CD=0 and CR0.NW=1.
- Any unsupported bit in CR3 is set to 1, including unsupported physical-address bits (bits 51-12).
- Certain reserved bits are set to 1, including:
  - Any CR0 bit in the range 63–32 is set to 1.
  - Any unsupported bit in CR4 is set to 1.
  - Any DR6 bit or DR7 bit in the range 63–32 is set to 1.
  - Any unsupported bit in EFER is set to 1.
- Invalid returns to long mode, including:
  - EFER.LME=1, CR0.PG=1, and CR4.PAE=0.
  - EFER.LME=1, CR0.PG=1, CR4.PAE=1, CS.L=1, and CS.D=1.
- The SSM revision identifier is modified.

Some SMRAM state-save-area conditions are ignored, and the registers, or bits within the registers, are restored in a default manner by the processor. This avoids a processor shutdown when an invalid condition is stored in SMRAM. The default conditions restored by the processor are:

- The EFER.LMA register bit is set to the value obtained by logically ANDing the SMRAM values of EFER.LME, CR0.PG, and CR4.PAE.
- The rFLAGS.VM register bit is set to the value obtained by logically ANDing the SMRAM values of rFLAGS.VM, CR0.PE, and the inverse of EFER.LMA.
- The base values of FS, GS, GDTR, IDTR, LDTR, and TR are restored in canonical form. Those values are sign-extended to bit 63 using the most-significant implemented bit.
- Unimplemented segment-base bits in the CS, DS, ES, and SS registers are cleared to 0.

11 128-Bit, 64-Bit, and x87 Programming

This chapter describes the system-software implications of supporting applications that use the 128-bit media, 64-bit media, and x87 instructions. Throughout this chapter, these instructions are collectively referred to as *media and x87* (media/x87) instructions. A complete listing of the instructions that fall in this category—and the detailed operation of each instruction—can be found in volumes 4 and 5. Refer to Volume 1 for information on using these instructions in application software.

11.1 Overview of System-Software Considerations

Processor implementations can support different combinations of the 128-bit media, 64-bit media, and x87 instruction sets. Two sets of registers—independent of the general-purpose registers—support these instructions. The 128-bit media instructions operate on the XMM registers, and the 64-bit media and x87-instructions operate on the aliased MMX™/x87 registers. The 128-bit media and x87 floating-point instruction sets have special status registers, control registers, exception vectors, and system-software control bits for managing the operating environment. System software that supports use of these instructions must be able to manage these resources properly including:

- Detecting support for the instruction set, and enabling any optional features, as necessary.
- Saving and restoring the processor media or x87 state.
- Execution of floating-point instructions (media or x87) can produce exceptions. System software must supply exception handlers for all unmasked floating-point exceptions.

11.2 Determining Media and x87 Feature Support

The support of 128-bit media, 64-bit media, and x87 instructions is implementation dependent. System software executes the CPUID instruction to determine whether a processor implements any of these features (see “Processor Feature Identification” on page 75 for more information on using the CPUID instruction). After CPUID is executed with an extended

function codes 1 and 8000_0001h, feature support can be determined by examining the contents of the EDX register. General guidelines for determining feature support are given in the list below. A few instructions belong to more than one instruction subset. Refer to "Instruction Subsets and CPUID Feature Sets" in Volume 3 for specific information.

- 128-bit media instructions are supported when:
  - EDX[25]=1 for SSE instructions. (Returned by CPUID function code 1.)
  - EDX[26]=1 for SSE2 instructions. (Returned by CPUID function code 1.)
- 64-bit media instructions are supported when:
  - EDX[23]=1 for MMX instructions. (Returned by CPUID function codes 1h and 8000_0001h.)
  - EDX[22]=1 for AMD extensions to MMX instructions. (Returned by CPUID function code 8000_0001h.)
  - EDX[31]=1 for AMD 3DNow!™ instructions. (Returned by CPUID function code 8000_0001h.)
  - EDX[30]=1 for AMD extensions to 3DNow! instructions. (Returned by CPUID function code 8000_0001h.)
- x87 floating-point instructions are supported when:
  - EDX[0]=1. (Returned by function codes 1h and 8000_0001h.)
- FXSAVE and FXRSTOR instructions are supported when:
  - EDX[24]=1. These instructions save and restore the entire media and x87 processor state. (Returned by function codes 1h and 8000_0001h.)

If software attempts to execute an instruction belonging to an unsupported instruction subset, an invalid-opcode exception (#UD) occurs. For a summary of instruction subsets, see "Instruction Subsets and CPUID Feature Sets" in Volume 3.

11.3 Enabling 128-Bit Media Instructions

Use of the 128-bit media instructions requires system software to support SSE and/or SSE2 features, but also the FXSAVE and FXRSTOR instructions, which are used to save and restore the 128-bit media state (see "FXSAVE and FXRSTOR Instructions" on page 349). When these instructions are supported, system

software must set CR4.OSFXSR=1 to let the processor know that the software uses these instructions. When the processor detects CR4.OSFXSR=1, it allows execution of the 128-bit media instructions. If system software does not set CR4.OSFXSR to 1, attempts to execute 128-bit media instructions cause an invalid-opcode exception (#UD).

System software must also *clear* the CR0.EM (emulate coprocessor) bit to 0, otherwise an attempt to execute a 128-bit media instruction causes a #UD exception.

System software should also *set* the CR0.MP (monitor coprocessor) bit to 1. When CR0.EM=0 and CR0.MP=1, all media instructions, x87 instructions, and the FWAIT/WAIT instructions cause a device-not-available exception (#NM) when the CR0.TS bit is set. System software can use the #NM exception to perform lazy context switching, saving and restoring media and x87 state only when necessary after a task switch. See “CR0 Register” on page 53 for more information.

System software must supply an exception handler if unmasked 128-bit media floating-point exceptions are allowed to occur. When an unmasked exception is detected, the processor transfers control to the SIMD floating-point exception (#XF) handler provided by the operating system. System software must let the processor know that the #XF handler is available by setting CR4.OSXMMEXCPT to 1. If this bit is set to 1, the processor transfers control to the #XF handler when it detects an unmasked exception, otherwise a #UD exception occurs. When the processor detects a masked exception, it handles it in a default manner regardless of the CR4.OSXMMEXCPT value.

11.4 Media and x87 Processor State

The media and x87 processor state includes the contents of the registers used by 128-bit media, 64-bit media, and x87 instructions. System software that supports such applications must be capable of saving and restoring these registers.

11.4.1 128-Bit Media State

Figure 11-1 on page 340 shows the registers whose contents are affected by execution of 128-bit media instructions. These include:

- *xmm0–xmm15*—Sixteen 128-bit media registers. In legacy and compatibility modes, software access is limited to the first eight registers, XMM0–XMM7.
- *MXCSR*—The 32-bit control and status register.

All of the above registers are visible to 128-bit media application software. Refer to "128-Bit Media and Scientific Programming" in Volume 1 for more information on these registers.

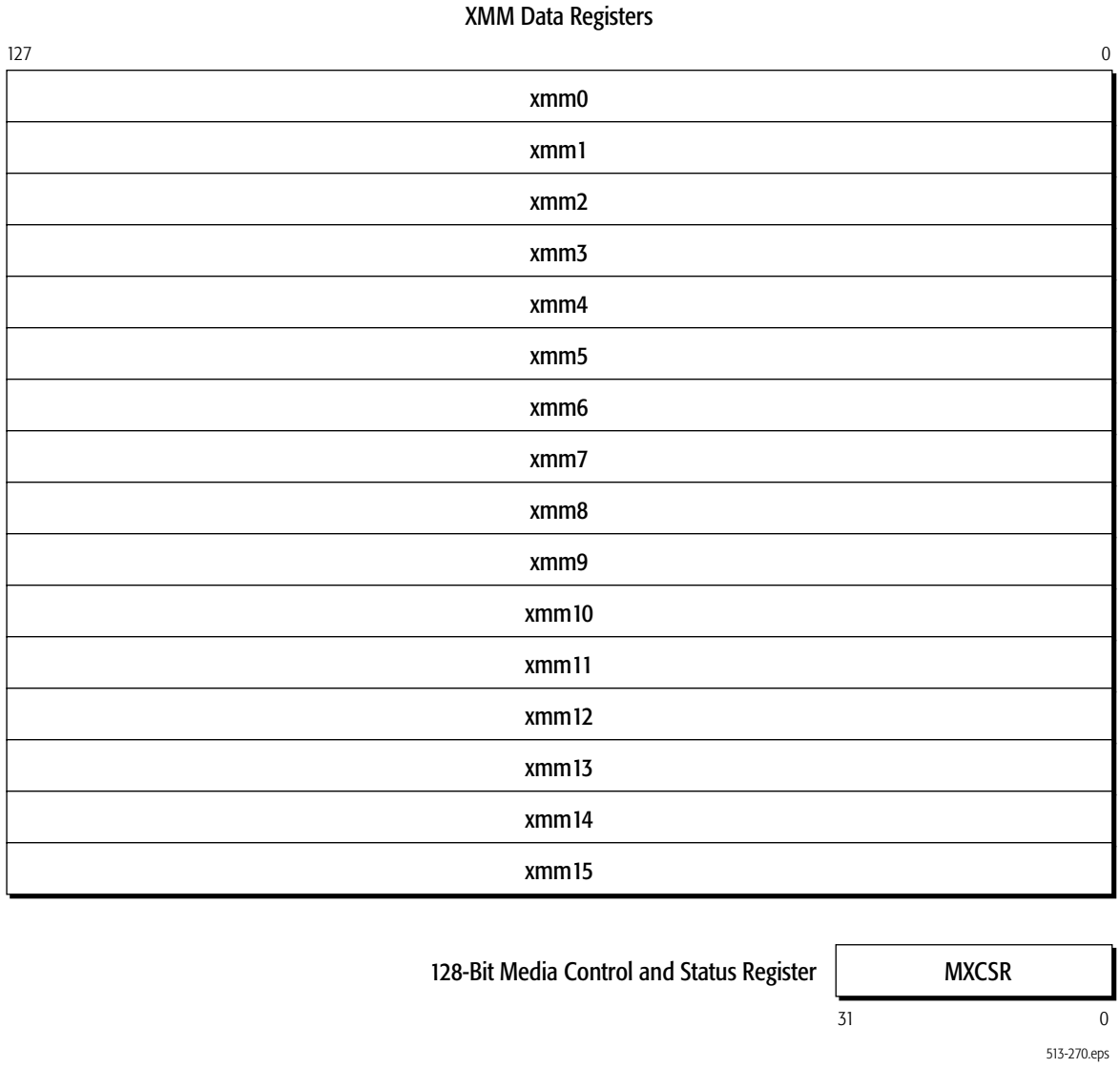


Figure 11-1. 128-Bit Media-Instruction State

11.4.2 **64-Bit Media State**

Figure 11-2 shows the register contents that are affected by execution of 64-bit media instructions. These registers include:

- *mmx0–mmx7*—Eight 64-bit media registers.
- *FSW*—Two fields (TOP and ES) in the 16-bit x87 status word register.
- *FTW*—The 16-bit x87 tag word.

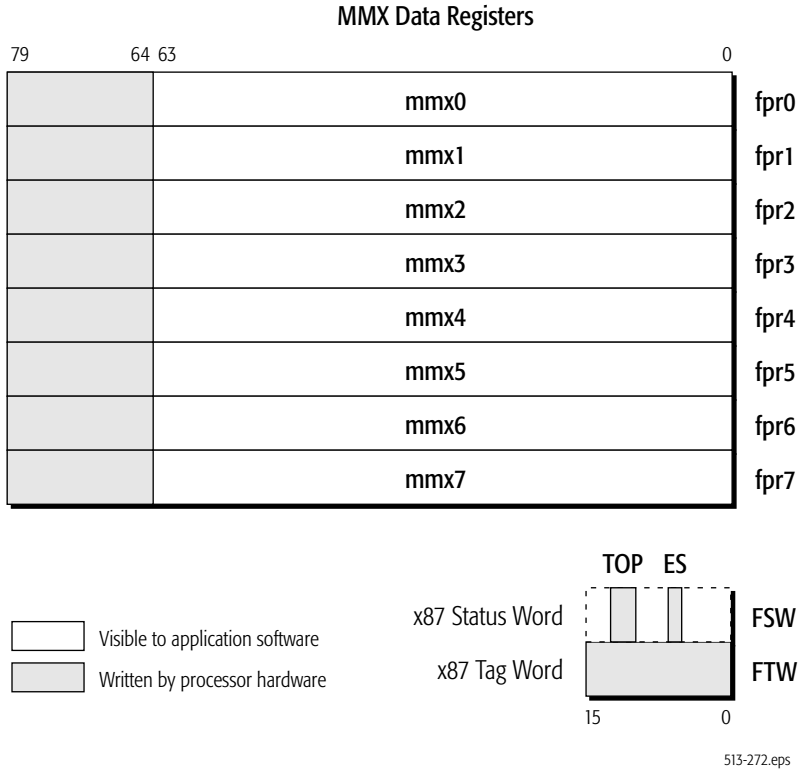


Figure 11-2. 64-Bit Media-Instruction State

The 64-bit media instructions and x87 floating-point instructions share the same physical data registers. Figure 11-2 shows how the 64-bit registers (MMX0–MMX7) are aliased onto the low 64 bits of the 80-bit x87 floating-point physical data registers (FPR0–FPR7). Refer to "64-Bit Media Programming" in Volume 1 for more information on these registers.

Of the registers shown in Figure 11-2, only the eight 64-bit MMX registers are visible to 64-bit media application software. The processor maintains the contents of the two fields of the x87 status word—top-of-stack-pointer (TOP) and exception

summary (ES)—and the 16-bit x87 tag word during execution of 64-bit media instructions, as described in "Actions Taken on Executing 64-Bit Media Instructions" in Volume 1.

64-bit media instructions do not generate x87 floating-point exceptions, nor do they set any status flags. However, 64-bit media instructions can trigger an unmasked floating-point exception caused by a previously executed x87 instruction. 64-bit media instructions do this by reading the x87 FSW.ES bit to determine whether such an exception is pending.

11.4.3 x87 State

Figure 11-3 on page 343 shows the registers whose contents are affected by execution of x87 floating-point instructions. These registers include:

- *fpr0–fpr7*—Eight 80-bit floating-point physical registers.
- *FCW*—The 16-bit x87 control word register.
- *FSW*—The 16-bit x87 status word register.
- *FTW*—The 16-bit x87 tag word.
- *Last x87 Instruction Pointer*—This value is a pointer (32-bit, 48-bit, or 64-bit, depending on effective operand size and mode) to the last non-control x87 floating-point instruction executed.
- *Last x87 Data Pointer*—The pointer (32-bit, 48-bit, or 64-bit, depending on effective operand size and mode) to the data operand referenced by the last non-control x87 floating-point instruction executed, if that instruction referenced memory; if it did not, then this value is implementation dependent.
- *Last x87 Opcode*—An 11-bit permutation of the instruction opcode from the last non-control x87 floating-point instruction executed.

Of the registers shown in Figure 11-3, only FPR0–FPR7, FCW, and FSW are directly updated by x87 application software. The processor maintains the contents of the FTW, instruction and data pointers, and opcode registers during execution of x87 instructions. Refer to "Registers" in Volume 1 for more information on these registers.

The 11-bit instruction opcode register holds a permutation of the two-byte instruction opcode from the last non-control x87 instruction executed by the processor. (For a definition of *non-*

control x87 instruction, see "Control" in Volume 1.) The opcode field is formed as follows:

- Opcode Register Field[10:8] = First x87 opcode byte[2:0].
- Opcode Register Field[7:0] = Second x87 opcode byte[7:0].

For example, the x87 opcode D9 F8h is stored in the opcode register as 001_1111_1000b. The low-order three bits of the first opcode byte, D9h (1101_1001b), are stored in opcode-register bits 10–8. The second opcode byte, F8h (1111_1000b), is stored in bits 7–0 of the opcode register. The high-order five bits of the first opcode byte (1101_1b) are not needed because they are identical for all x87 instructions.

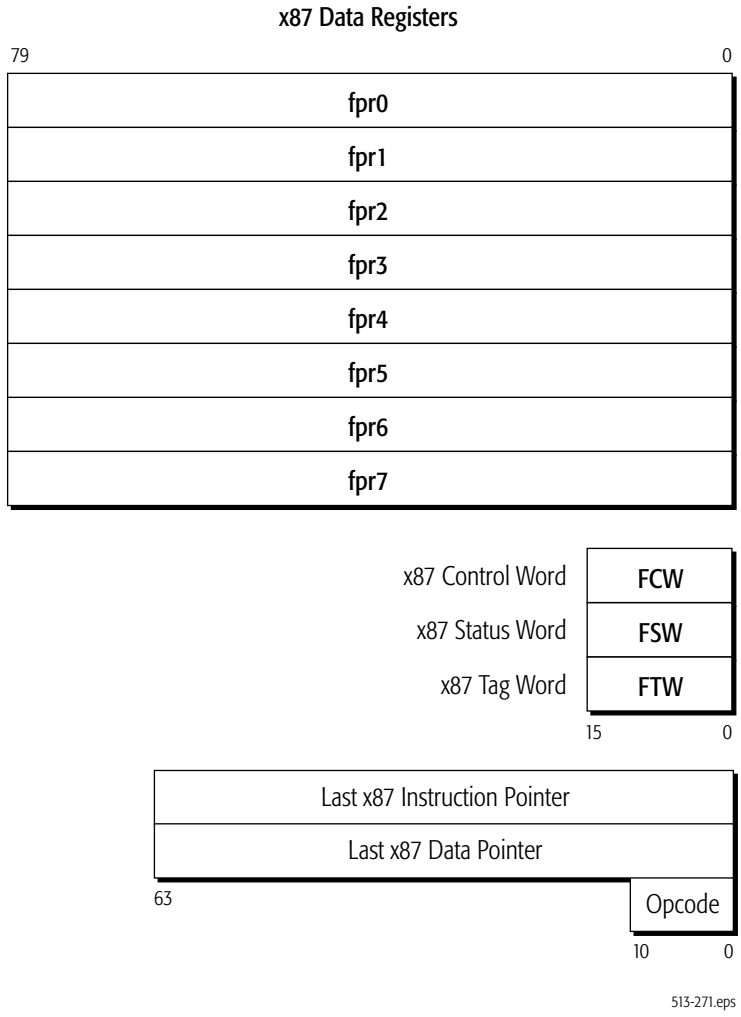


Figure 11-3. x87-Instruction State

11.4.4 Saving Media and x87 Processor State

In most cases, operating systems, exception handlers, and device drivers should save and restore the media and/or x87 processor state between task switches or other interventions in the execution of 128-bit, 64-bit, or x87 procedures. Application programs are also free to save and restore state at any time.

In general, system software should use the FXSAVE and FXRSTOR instructions to save and restore the entire media and x87 processor state. The FSAVE/FNSAVE and FRSTOR instructions can be used for saving and restoring the x87 state. Because the 64-bit media registers are physically aliased onto the x87 registers, the FSAVE/FNSAVE and FRSTOR instructions can also be used to save and restore the 64-bit media state. However, FSAVE/FNSAVE and FRSTOR do not save or restore the 128-bit media state.

FSAVE/FNSAVE and FRSTOR Instructions. The FSAVE/FNSAVE and FRSTOR instructions save and restore the entire register state for 64-bit media instructions and x87 floating-point instructions. The FSAVE instruction stores the register state, but only after handling any pending unmasked-x87 floating-point exceptions. The FNSAVE instruction stores the register state but skips the reporting and handling of these exceptions. The state of all MMX/FPR registers is saved, as well as all other x87 state (the control word register, status word register, tag word, instruction pointer, data pointer, and last opcode). After saving this state, the tag state for all MMX/FPR registers is changed to *empty* and is thus available for a new procedure.

Starting on page 346, Figure 11-4 through Figure 11-7 show the memory formats used by the FSAVE/FNSAVE and FRSTOR instructions when storing the x87 state in various processor modes and using various effective-operand sizes. This state includes:

- *x87 Data Registers*
  - FPR0–FPR7 80-bit physical data registers.
- *x87 Environment*
  - FCW: x87 control word register
  - FSW: x87 status word register
  - FTW: x87 tag word
  - Last x87 instruction pointer
  - Last x87 data pointer

- Last x87 opcode

The eight data registers are stored in the 80 bytes following the environment information. Instead of storing these registers in their physical order (FPR0–FPR7), the processor stores the registers in the their stack order, ST(0)–ST(7), beginning with the top-of-stack, ST(0).

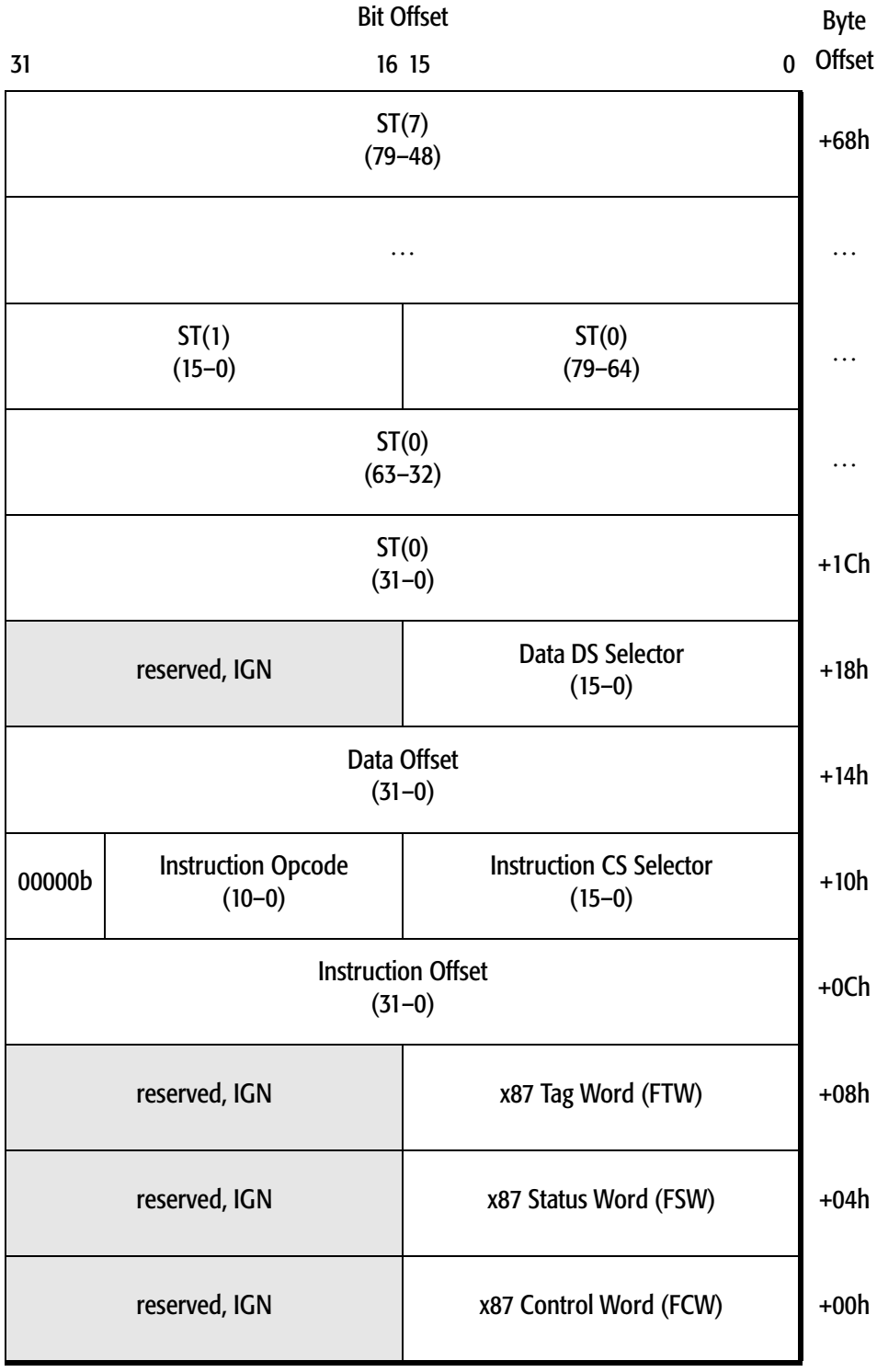


Figure 11-4. FSAVE/FNSAVE Image (32-Bit, Protected Mode)

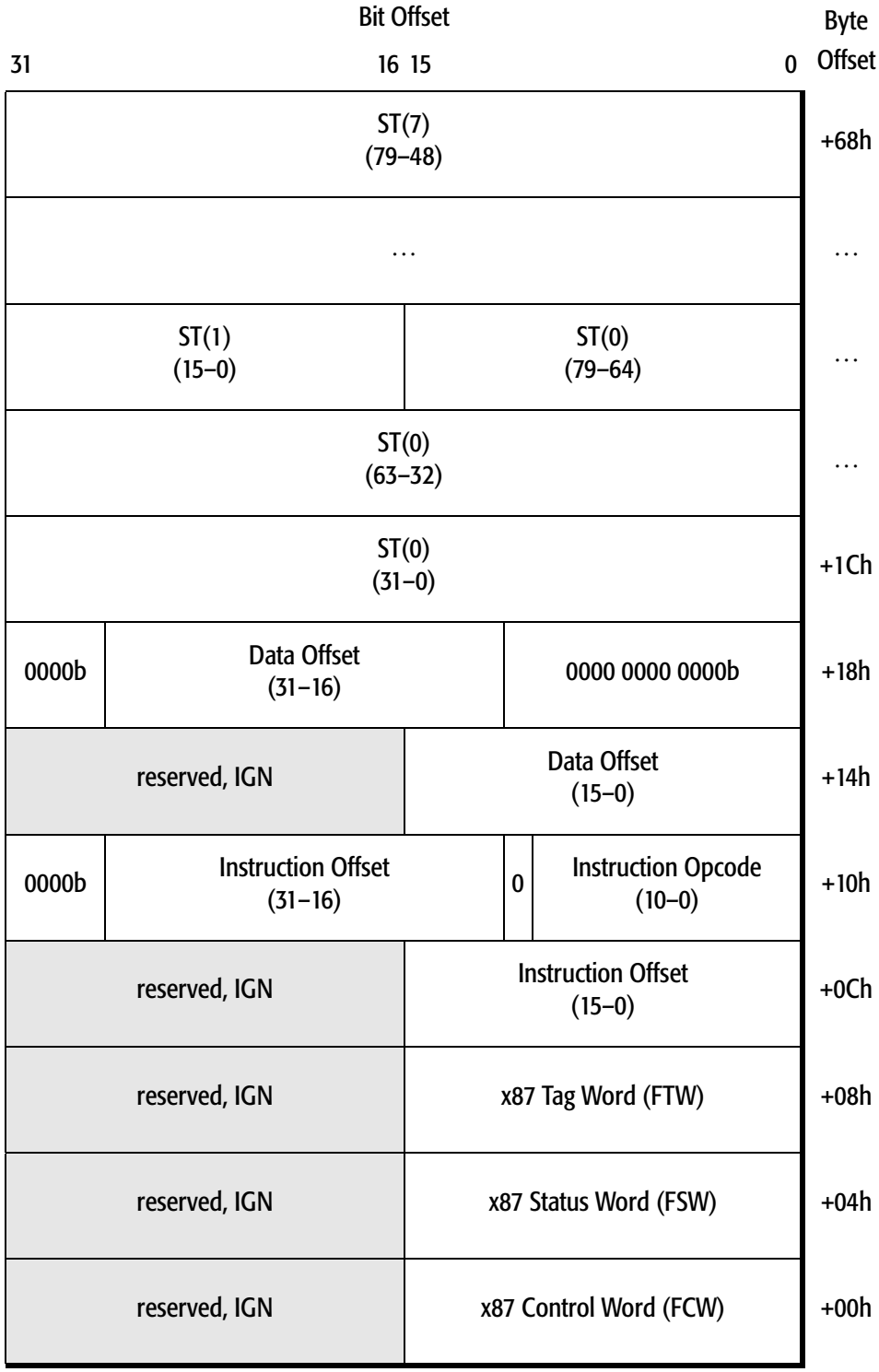


Figure 11-5. FSAVE/FNSAVE Image (32-Bit, Real/Virtual-8086 Modes)

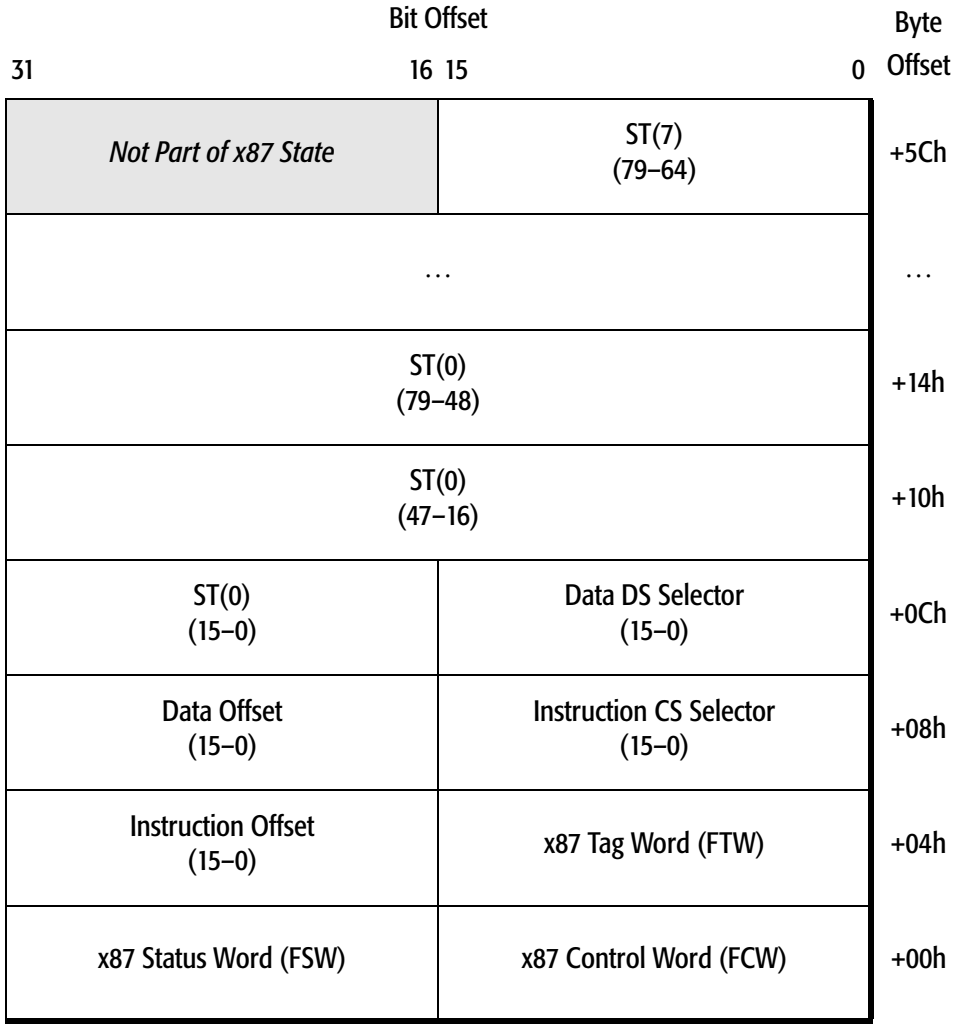


Figure 11-6. FSAVE/FNSAVE Image (16-Bit, Protected Mode)

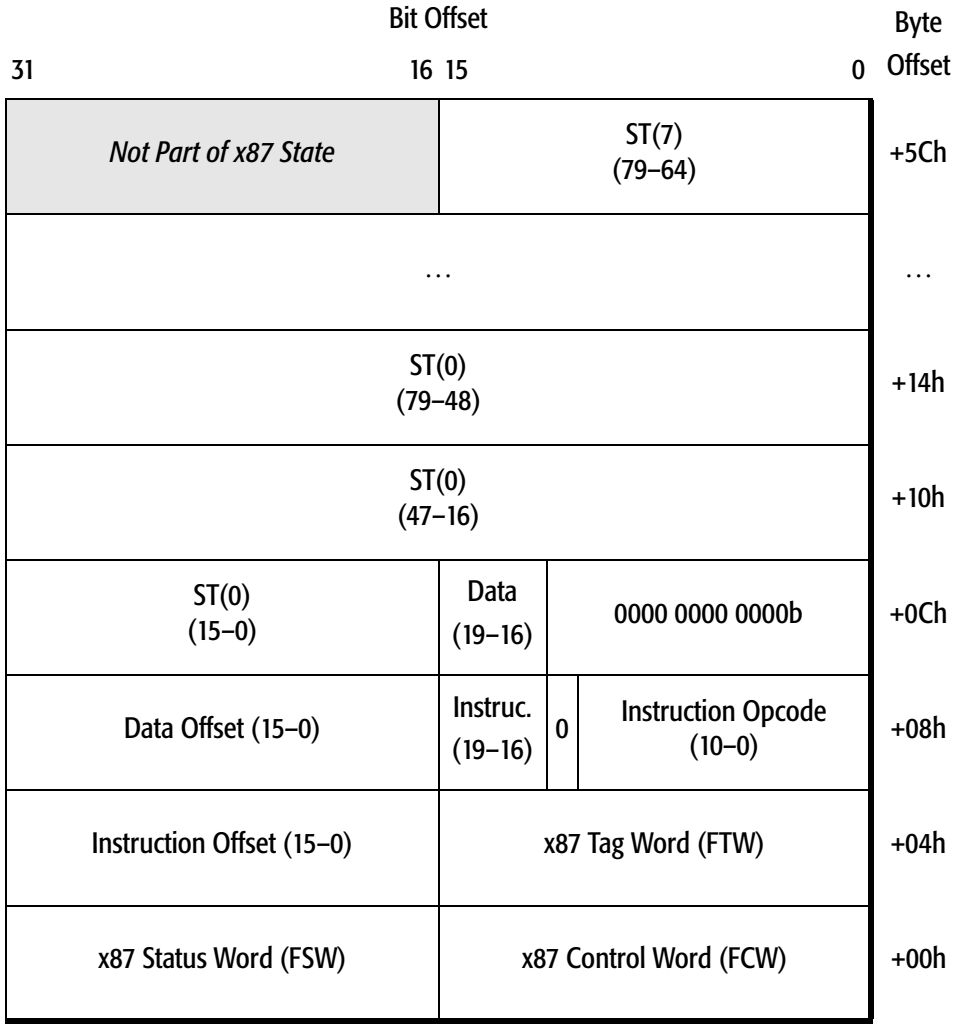


Figure 11-7. FSAVE/FNSAVE Image (16-Bit, Real/Virtual-8086 Modes)

FLDENV/FNLDENV and FSTENV Instructions. The FLDENV/FNLDENV and FSTENV instructions load and store only the x87 floating-point environment. These instructions, unlike the FSAVE/FNSAVE and FRSTOR instructions, do not save or restore the x87 data registers.

FXSAVE and FXRSTOR Instructions. The FXSAVE and FXRSTOR instructions save and restore the entire 128-bit media, 64-bit media, and x87 state. These instructions usually execute faster than FSAVE/FNSAVE and FRSTOR because they do not normally save and restore the x87 exception pointers (last-instruction pointer, last data-operand pointer, and last opcode).

The only case in which they do save the exception pointers is the relatively rare case in which the exception-summary bit in the x87 status word (FSW.ES) is set to 1, indicating that an unmasked exception has occurred. The FXSAVE and FXRSTOR memory format contains fields for storing these values.

Unlike FSAVE and FNSAVE, the FXSAVE instruction does not alter the x87 tag word. Therefore, the contents of the shared 64-bit MMX and 80-bit FPR registers can remain valid after an FXSAVE instruction (or any other value the tag bits indicated before the save). Also, FXSAVE (like FNSAVE) does not check for pending unmasked-x87 floating-point exceptions.

Figure 11-8 on page 351 shows the memory format of the media x87 state in long mode. When in long mode using a 64-bit operand size, the format shown in Figure 11-8 is used. If a 32-bit operand size is used (in long mode), the memory format is the same, except that RIP and RDS are stored as sel:offset pointers, as shown in Figure 11-9 on page 352.

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
reserved, IGN																+1F0h
…																…
reserved, IGN																+1A0h
XMM15																+190h
XMM14																+180h
XMM13																+170h
XMM12																+160h
XMM11																+150h
XMM10																+140h
XMM9																+130h
XMM8																+120h
XMM7																+110h
XMM6																+100h
XMM5																+F0h
XMM4																+E0h
XMM3																+D0h
XMM2																+C0h
XMM1																+B0h
XMM0																+A0h
reserved, IGN								MMX7/ST(7)								+90h
reserved, IGN								MMX6/ST(6)								+80h
reserved, IGN								MMX5/ST(5)								+70h
reserved, IGN								MMX4/ST(4)								+60h
reserved, IGN								MMX3/ST(3)								+50h
reserved, IGN								MMX2/ST(2)								+40h
reserved, IGN								MMX1/ST(1)								+30h
reserved, IGN								MMX0/ST(0)								+20h
MXCSR_MASK						MXCSR				RDP[1]						+10h
RIP[1]								FOP		0		FTW		FSW	FCW	+00h

1. Stored as sel:offset if operand size is 32 bits.

Figure 11-8. FXSAVE and FXRSTOR Image (Long Mode)

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0	Byte
reserved, IGN																+1F0h
…																…
reserved, IGN																+120h
XMM7																+110h
XMM6																+100h
XMM5																+F0h
XMM4																+E0h
XMM3																+D0h
XMM2																+C0h
XMM1																+B0h
XMM0																+A0h
reserved, IGN								MMX7/ST(7)								+90h
reserved, IGN								MMX6/ST(6)								+80h
reserved, IGN								MMX5/ST(5)								+70h
reserved, IGN								MMX4/ST(4)								+60h
reserved, IGN								MMX3/ST(3)								+50h
reserved, IGN								MMX2/ST(2)								+40h
reserved, IGN								MMX1/ST(1)								+30h
reserved, IGN								MMX0/ST(0)								+20h
MXCSR_MASK				MXCSR				rsrvd, IGN		DS			DP			+10h
rsrvd, IGN		CS		EIP				FOP		0	FTW		FSW		FCW	+00h

Figure 11-9. FXSAVE and FXRSTOR Image (Legacy Mode)

Software can read and write all fields within the FXSAVE and FXRSTOR memory image. These fields include:

- *FCW*—Bytes 01h–00h. x87 control word.
- *FSW*—Bytes 03h–02h. x87 status word.
- *FTW*—Byte 04h. x87 tag word. See "FXSAVE Format for x87 Tag Word" on page 354 for additional information on the FTW format saved by the FXSAVE instruction.
- (Byte 05h contains the value 00h.)

- *FOP*—Bytes 07h–06h. last x87 opcode.
- *Last x87 Instruction Pointer*—A pointer to the last non-control x87 floating-point instruction executed by the processor:
  - *RIP (64-bit format)*—Bytes 0Fh–08h. 64-bit offset into the code segment (used without a CS selector).
  - *EIP (32-bit format)*—Bytes 0Bh–08h. 32-bit offset into the code segment.
  - *CS (32-bit format)*—Bytes 0Dh–0Ch. Segment selector portion of the pointer.
- *Last x87 Data Pointer*—If the last non-control x87 floating point instruction referenced memory, this value is a pointer to the data operand referenced by the last non-control x87 floating-point instruction executed by the processor:
  - *RDP (64-bit format)*—Bytes 17h–10h. 64-bit offset into the data segment (used without a DS selector).
  - *DP (32-bit format)*—Bytes 13h–10h. 32-bit offset into the data segment.
  - *DS (32-bit format)*—Bytes 15h–14h. Segment selector portion of the pointer.

  If the last non-control x87 instruction did not reference memory, then the value in the pointer is implementation dependent.
- *MXCSR*—Bytes 1Bh–18h. 128-bit media-instruction control and status register. This register is saved only if CR4.OSFXSR is set to 1.
- *MXCSR_MASK*—Bytes 1Fh–1Ch. Set bits in MXCSR_MASK indicate supported feature bits in MXCSR. For example, if bit 6 (the DAZ bit) in the returned MXCSR_MASK field is set to 1, the DAZ mode and the DAZ flag in MXCSR *are* supported. Cleared bits in MXCSR_MASK indicate reserved bits in MXCSR. If software attempts to set a reserved bit in the MXCSR register, a #GP exception will occur. To avoid this exception, after software clears the FXSAVE memory image and executes the FXSAVE instruction, software should use the value returned by the processor in the MXCSR_MASK field when writing a value to the MXCSR register, as follows:
  - *MXCSR_MASK = 0:* If the processor writes a zero value into the MXCSR_MASK field, the denormals-are-zeros (DAZ) mode and the DAZ flag in MXCSR are *not*

supported. Software should use the default mask value, 0000_FFBFh (bit 6, the DAZ bit, and bits 31–16 cleared to 0), to mask any value it writes to the MXCSR register to ensure that all reserved bits in MXCSR are written with 0, thus avoiding a #GP exception.

. *MXCSR_MASK ≠ 0:* If the processor writes a non-zero value into the MXCSR_MASK field, software should AND this value with any value it writes to the MXCSR register.

- *MMXn/FPRn*—Bytes 9Fh–20h. Shared 64-bit media and x87 floating-point registers. As in the case of the x87 FSAVE instruction, these registers are stored in stack order ST(0)/MMX0–ST(7)/MMX7. The upper six bytes in the memory image for each register are reserved.
- *XMMn*—Bytes 11Fh–A0h. 128-bit media registers. These registers are saved only if CR4.OSFXSR is set to 1.

FXSAVE Format for x87 Tag Word. Rather than saving the entire x87 tag word, FXSAVE saves a single-byte encoded version. FXSAVE encodes each of the eight two-bit fields in the x87 tag word as follows:

- Two-bit values of 00, 01, and 10 are encoded as a 1, indicating the corresponding x87 FPR*n* register holds a value.
- A two-bit value of 11 is encoded as a 0, indicating the corresponding x87 FPR*n* is empty.

For example, assume an FSAVE instruction saves an x87 tag word with the value 83F1h. This tag-word value describes the x87 FPRn contents as follows:

x87 Register	FPR7	FPR6	FPR5	FPR4	FPR3	FPR2	FPR1	FPR0
Tag Word Value (hex)	8		3		F		1	
Tag Value (binary)	10	00	00	11	11	11	00	01
Meaning	Special	Valid	Valid	Empty	Empty	Empty	Valid	Zero

When an FXSAVE is used to write the x87 tag word to memory, it encodes the value as E3h. This encoded version describes the x87 FPRn contents as follows:

x87 Register	FPR7	FPR6	FPR5	FPR4	FPR3	FPR2	FPR1	FPR0
Encoded Tag Byte (hex)	E				3			
Tag Value (binary)	1	1	1	0	0	0	1	1
Meaning	Valid	Valid	Valid	Empty	Empty	Empty	Valid	Valid

If necessary, software can decode the single-bit FXSAVE tag-word fields into the two-bit field FSAVE uses by examining the contents of the corresponding FPR registers saved by FXSAVE. Table 11-1 shows how the FPR contents are used to find the equivalent FSAVE tag-field value. The *fraction* column refers to fraction portion of the extended-precision significand (bits 62–0). The *integer bit* column refers to the integer-portion of the significand (bit 63). See "x87 Floating-Point Programming" in Volume 1 for more information on floating-point numbering formats.

Table 11-1. Deriving FSAVE Tag Field from FXSAVE Tag Field

Encoded FXSAVE Tag Field	Exponent	Integer Bit[2]	Fraction[1]	Type of Value	Equivalent FSAVE Tag Field
1 (Valid)	All 0s	0	All 0s	Zero	01 (Zero)
		0	Not all 0s	Denormal	10 (Special)
		1	All 0s	Pseudo Denormal	
		1	Not all 0s		
	Neither all 0s nor all 1s	0	don't care	Unnormal	
		1		Normal	00 (Valid)
	All 1s	0		Pseudo Infinity or Pseudo NaN	10 (Special)
		1	All 0s	Infinity	
			Not all 0s	NaN	
0 (Empty)	don't care			Empty	11 (Empty)

Note:

1. *Bits 62–0 of the significand. Bit 62, the most-significant bit of the fraction, is also called the M bit.*
2. *Bit 63 of the significand, also called the J bit,*

Performance Considerations. When system software supports multi-tasking, it must be able to save the processor state for one task and load the state for another. For performance reasons, the media and/or x87 processor state is usually saved and loaded only when necessary. System software can save and load this state at the time a task switch occurs. However, if the new task does not use the state, loading the state is unnecessary and reduces performance.

The task-switch bit (CR0.TS) is provided as a *lazy* context-switch mechanism that allows system software to save and load state only when necessary. When CR0.TS=1, a device-not-available exception (#NM) occurs when an attempt is made to execute a 128-bit media, 64-bit media, or x87 instruction. System software can use the #NM exception handler to save the state of the previous task, and restore the state of the current task. Before returning from the exception handler to the media or x87 instruction, system software must clear CR0.TS to 0 to allow the instruction to be executed. Using this approach, state is saved only when the registers are used.

In legacy mode, the hardware task-switch mechanism sets CR0.TS=1 during a task switch (see "Task Switched (TS) Bit" on page 54 for more information). In long mode, the hardware task-switching is not supported, and the CR0.TS bit is not set by the processor. Instead, the architecture assumes that system software handles all task-switching and state-saving functions. If CR0.TS is to be used in long mode for controlling the save and restore of media or x87 state, system software must set and clear it explicitly.

12 Task Management

This chapter describes the hardware task-management features. All of the legacy x86 task-management features are supported by the AMD64 architecture in legacy mode, but most features are not available in long mode. Long mode, however, requires system software to initialize and maintain certain task-management *resources*. The details of these resource-initialization requirements for long mode are discussed in "Task-Management Resources" on page 358.

12.1 Hardware Multitasking Overview

A task (also called a *process*) is a 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:

- Code segment and instruction pointer.
- Data segments.
- Stack segments for each privilege level.
- General-purpose registers.
- rFLAGS register.
- Local-descriptor table.
- Task register, and a link to the previously-executed task.
- I/O-permission and interrupt-permission bitmaps.
- Pointer to the page-translation tables (CR3).

The state information defining this execution space is stored in the task-state segment (TSS) maintained for each task.

Full support for hardware multitasking is provided 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:

- Suspends execution of the task, allowing any executing instructions to complete and save their results.
- Saves the task execution state in the task TSS.

- Loads the execution state for the new task from its TSS.
- Begins executing the new task at the location specified in the new task TSS.

Software can switch tasks by branching to a new task using the CALL or JMP instructions. Exceptions and interrupts can also switch tasks if the exception or interrupt handlers are themselves separate tasks. IRET can be used to return to an earlier task.

12.2 Task-Management Resources

The hardware-multitasking features are available when protected mode is enabled (CR0.PE=1). Protected-mode software execution, by definition, occurs as part of a task. While system software is not required to use the hardware-multitasking features, it is required to initialize certain task-management resources for at least one task (the current task) when running in protected mode. This single task is needed to establish the protected-mode execution environment. The resources that must be initialized are:

- *Task-State Segment (TSS)*—A segment that holds the processor state associated with a task.
- *TSS Descriptor*—A segment descriptor that defines the task-state segment.
- *TSS Selector*—A segment selector that references the TSS descriptor located in the GDT.
- *Task Register*—A register that holds the TSS selector and TSS descriptor for the current task.

Figure 12-1 on page 359 shows the relationship of these resources to each other in both 64-bit and 32-bit operating environments.

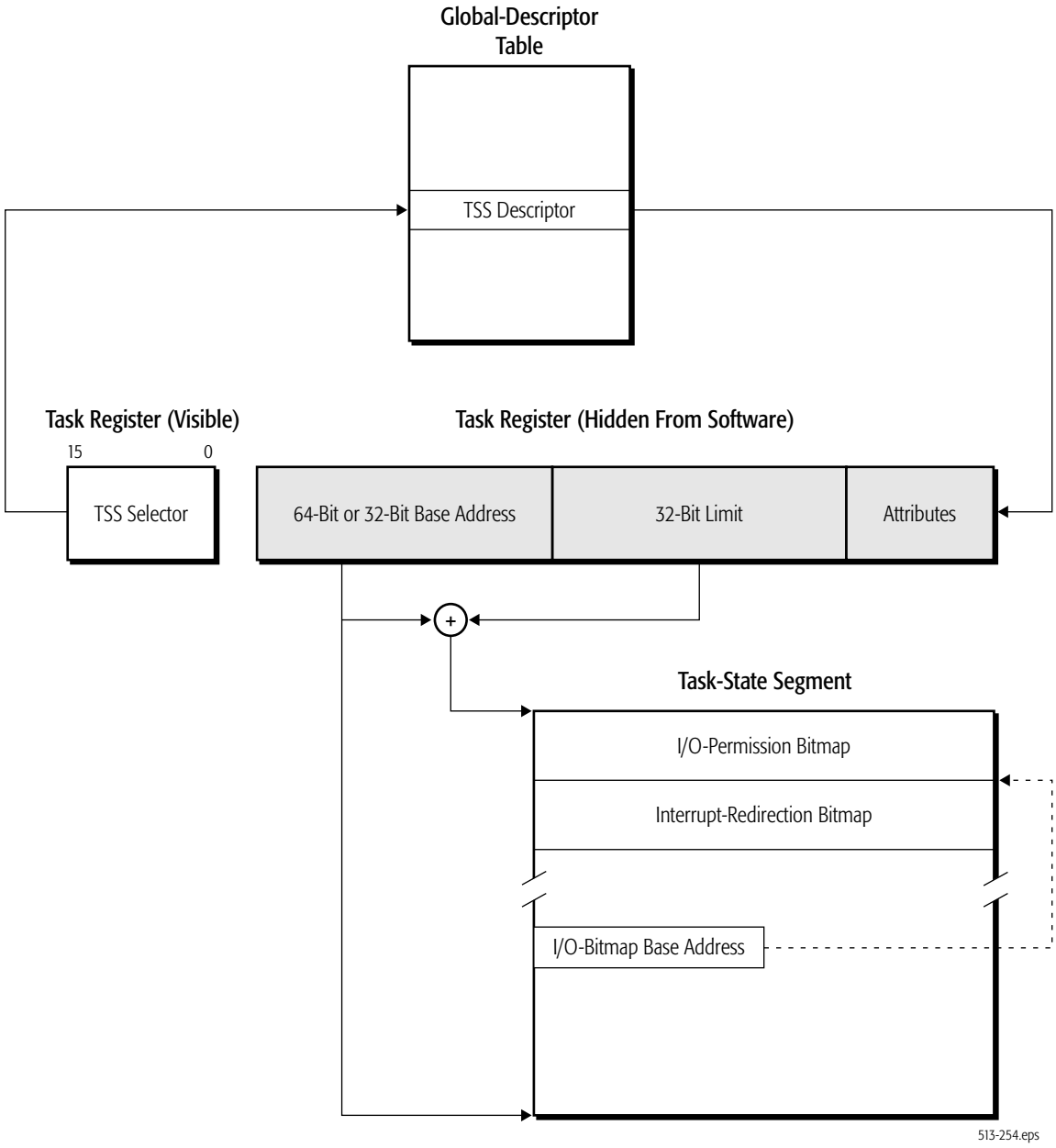


Figure 12-1. Task-Management Resources

A fifth resource is available in legacy mode for use by system software that uses the hardware-multitasking mechanism to manage more than one task:

- *Task-Gate Descriptor*—This form of gate descriptor holds a reference to a TSS descriptor and is used to control access between tasks.

The task-management resources are described in the following sections.

12.2.1 TSS Selector

TSS selectors are selectors that point to task-state segment descriptors in the GDT. Their format is identical to all other segment selectors, as shown in Figure 12-2.

Bits	Mnemonic	Description
15-3		Selector Index
2	TI	Table Indicator
1-0	RPL	Requestor Privilege Level

Figure 12-2. Task-Segment Selector

The selector format consists of the following fields:

Selector Index. Bits 15–3. The selector-index field locates the TSS descriptor in the global-descriptor table.

Table Indicator (TI) Bit. Bit 2. The TI bit must be cleared to 0, which indicates that the GDT is used. TSS descriptors cannot be located in the LDT. If a reference is made to a TSS descriptor in the LDT, a general-protection exception (#GP) occurs.

Requestor Privilege-Level (RPL) Field. Bits 1–0. RPL represents the privilege level (CPL) the processor is operating under at the time the TSS selector is loaded into the task register.

12.2.2 TSS Descriptor

The TSS descriptor is a system-segment descriptor, and it can be located only in the GDT. The format for an 8-byte, legacy-mode and compatibility-mode TSS descriptor can be found in “System Descriptors” on page 102. The format for a 16-byte, 64-bit mode TSS descriptor can be found in “System Descriptors” on page 109.

The fields within a TSS descriptor (all modes) are described in “Descriptor Format” on page 95. The following additional information applies to TSS descriptors:

- *Segment Limit*—A TSS descriptor must have a segment limit value of at least 67h, which defines a minimum TSS size of 68h (104 decimal) bytes. If the limit is less than 67h, an invalid-TSS exception (#TS) occurs during the task switch. When an I/O-permission bitmap, interrupt-redirection bitmap, or additional state information is included in the TSS, the limit must be set to a value large enough to enclose that information. In this case, if the TSS limit is not large enough to hold the additional information, a #GP exception occurs when an attempt is made to access beyond the TSS limit. No check for the larger limit is performed during the task switch.
- *Type*—Four system-descriptor types are defined as TSS types, as shown in Table 4-5 on page 103. Bit 9 is used as the descriptor busy bit (B). This bit indicates that the task is busy when set to 1, and available when cleared to 0. Busy tasks are the currently running task and any previous (outer) tasks in a nested-task hierarchy. Task recursion is not supported, and a #GP exception occurs if an attempt is made to transfer control to a busy task. See "Nesting Tasks" on page 377 for additional information.

  In long mode, the 32-bit TSS types (available and busy) are redefined as 64-bit TSS types, and only 64-bit TSS descriptors can be used. Loading the task register with an available 64-bit TSS causes the processor to change the TSS descriptor type to indicate a busy 64-bit TSS. Because long mode does not support task switching, the TSS-descriptor busy bit is never cleared by the processor to indicate an available 64-bit TSS.

  16-bit TSS types are illegal in long mode. A general-protection exception (#GP) occurs if a reference is made to a 16-bit TSS.

12.2.3 Task Register

The *task register* (TR) points to the TSS location in memory, defines its size, and specifies its attributes. As with the other descriptor-table registers, the TR has two portions. A *visible* portion holds the TSS selector, and a *hidden* portion holds the TSS descriptor. When the TSS selector is loaded into the TR, the processor automatically loads the TSS descriptor from the GDT into the hidden portion of the TR.

The TR is loaded with a new selector using the LTR instruction. The TR is also loaded during a task switch, as described in "Switching Tasks" on page 373.

Figure 12-3 shows the format of the TR in legacy mode.

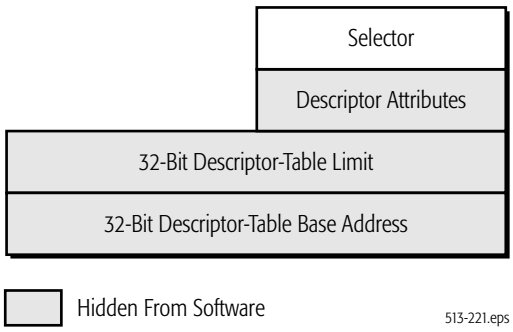


Figure 12-3. TR Format, Legacy Mode

Figure 12-4 shows the format of the TR in long mode (both compatibility mode and 64-bit mode).

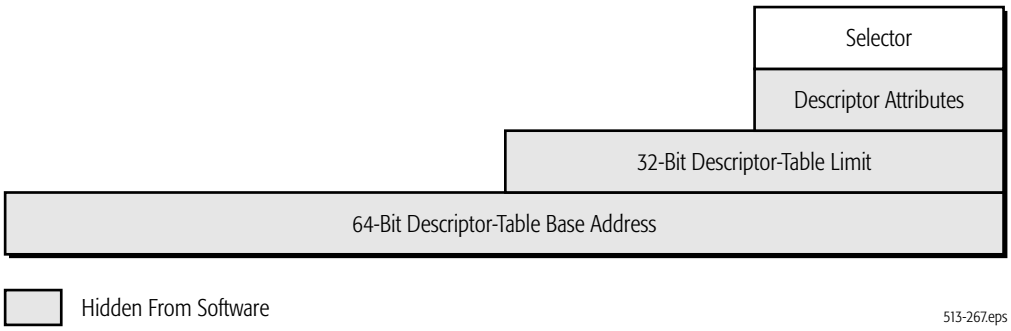


Figure 12-4. TR Format, Long Mode

The AMD64 architecture expands the TSS-descriptor base-address field to 64 bits so that system software running in long mode can access a TSS located 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 TR is loaded from the GDT, the system-segment descriptor format has been expanded to 16 bytes by the AMD64 architecture in support of 64-bit mode. See “System Descriptors” on page 109 for more information on this expanded format. The high-order base-address bits are only loaded from 64-bit mode using the

LTR instruction. Figure 12-5 shows the relationship between the TSS and GDT.

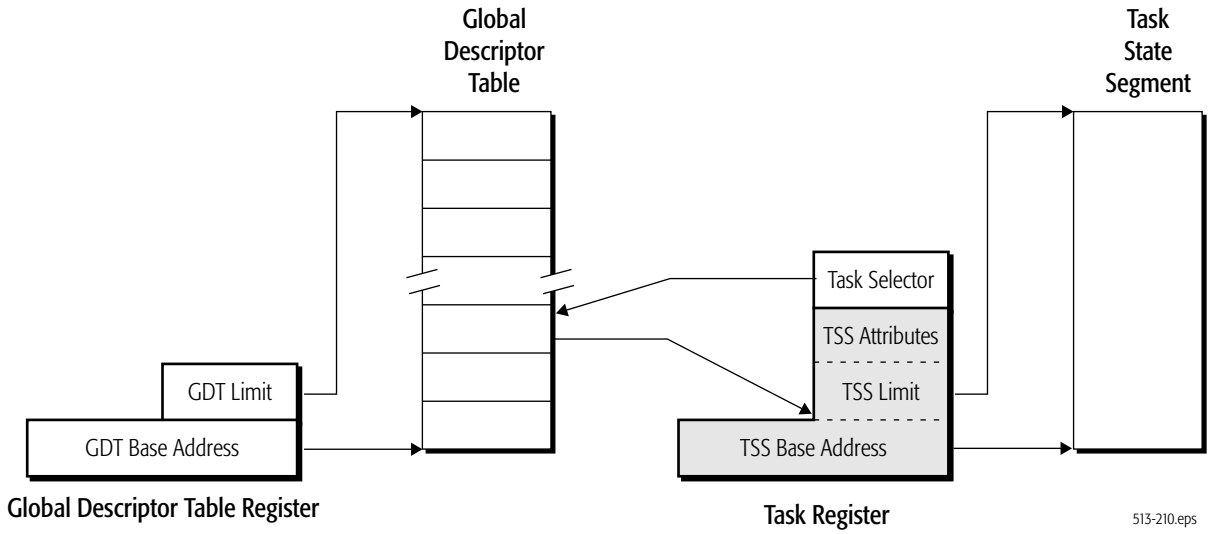


Figure 12-5. Relationship between the TSS and GDT

Long mode requires the use of a 64-bit TSS type, and this type must be loaded into the TR by executing the LTR instruction in *64-bit mode*. Executing the LTR instruction in 64-bit mode loads the TR with the full 64-bit TSS base address from the 16-byte TSS descriptor format (compatibility mode can only load 8-byte system descriptors). A processor running in either compatibility mode or 64-bit mode uses the full 64-bit TR.base address.

12.2.4 Legacy Task-State Segment

The task-state segment (TSS) is a data structure in memory that the processor uses to save and restore the execution state for a task when a task switch occurs. Figure 12-6 on page 364 shows the format of a legacy 32-bit TSS.

Bit Offset 31	16	15	0	Byte Offset
I/O-Permission Bitmap (IOPB) (Up to 8 Kbytes)				IOPB Base
Interrupt-Redirection Bitmap (IRB) (Eight 32-Bit Locations)				
↑ ↓ Operating-System Data Structure			↑ ↓	
I/O-Permission Bitmap Base Address		reserved, IGN	T	+64h
reserved, IGN		LDT Selector		+60h
reserved, IGN		GS		+5Ch
reserved, IGN		FS		+58h
reserved, IGN		DS		+54h
reserved, IGN		SS		+50h
reserved, IGN		CS		+4Ch
reserved, IGN		ES		+48h
EDI				+44h
ESI				+40h
EBP				+3Ch
ESP				+38h
EBX				+34h
EDX				+30h
ECX				+2Ch
EAX				+28h
EFLAGS				+24h
EIP				+20h
CR3				+1Ch
reserved, IGN		SS2		+18h
ESP2				+14h
reserved, IGN		SS1		+10h
ESP1				+0Ch
reserved, IGN		SS0		+08h
ESP0				+04h
reserved, IGN		Link (Prior TSS Selector)		+00h

Figure 12-6. Legacy 32-bit TSS

The 32-bit TSS contains three types of fields:

- *Static fields* are read by the processor during a task switch when a new task is loaded, but are not written by the processor when a task is suspended.
- *Dynamic fields* are read by the processor during a task switch when a new task is loaded, and are written by the processor when a task is suspended.
- *Software-defined fields* are read and written by software, but are not read or written by the processor. All but the first 104 bytes of a TSS can be defined for software purposes, minus any additional space required for the optional I/O-permission bitmap and interrupt-redirection bitmap.

TSS fields are not read or written by the processor when the LTR instruction is executed. The LTR instruction loads the TSS descriptor into the TR and marks the task as busy, but it does not cause a task switch.

The TSS fields used by the processor in legacy mode are:

- *Link*—Bytes 01h–00h, dynamic field. Contains a copy of the task selector from the previously-executed task. See "Nesting Tasks" on page 377 for additional information.
- *Stack Pointers*—Bytes 1Bh–04h, static field. Contains the privilege 0, 1, and 2 stack pointers for the task. These consist of the stack-segment selector (SS*n)*, and the stack-segment offset (ESP*n*).
- *CR3*—Bytes 1Fh–1Ch, static field. Contains the page-translation-table base-address (CR3) register for the task.
- *EIP*—Bytes 23h–20h, dynamic field. Contains the instruction pointer (EIP) for the next instruction to be executed when the task is restored.
- *EFLAGS*—Bytes 27h–24h, dynamic field. Contains a copy of the EFLAGS image at the point the task is suspended.
- *General-Purpose Registers*—Bytes 47h–28h, dynamic field. Contains a copy of the EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI values at the point the task is suspended.
- *Segment-Selector Registers*—Bytes 59h–48h, dynamic field. Contains a copy of the ES, CS, SS, DS, FS, and GS, values at the point the task is suspended.

- *LDT Segment-Selector Register*—Bytes 63h–60h, static field. Contains the local-descriptor-table segment selector for the task.
- *T (Trap) Bit*—Bit 0 of byte 64h, static field. This bit, when set to 1, causes a debug exception (#DB) to occur on a task switch. See "Task-Switch Breakpoints" on page 398 for additional information.
- *I/O-Permission Bitmap Base Address*—Bytes 67h–66h, static field. This field represents a 16-bit offset into the TSS. This offset points to the beginning of the I/O-permission bitmap, and the end of the interrupt-redirection bitmap.
- *I/O-Permission Bitmap*—Static field. This field specifies protection for I/O-port addresses (up to the 64K ports supported by the processor), as follows:
  - Whether the port can be accessed at any privilege level.
  - Whether the port can be accessed outside the privilege level established by EFLAGS.IOPL.
  - Whether the port can be accessed when the processor is running in virtual-8086 mode.

  Because one bit is used per 8-byte I/O-port, this bitmap can take up to 8 Kbytes of TSS space. The bitmap can be located anywhere within the first 64 Kbytes of the TSS, as long as it is above byte 103. The last byte of the bitmap must contain all ones (0FFh). See "I/O-Permission Bitmap" on page 367 for more information.
- *Interrupt-Redirection Bitmap*—Static field. This field defines how each of the 256-possible software interrupts is directed in a virtual-8086 environment. One bit is used for each interrupt, for a total bitmap size of 32 bytes. The bitmap can be located anywhere above byte 103 within the first 64 Kbytes of the TSS. See "Interrupt Redirection of Software Interrupts" on page 297 for information on using this field.

The TSS can be paged by system software. System software that uses the hardware task-switch mechanism must guarantee that a page fault does not occur during a task switch. Because the processor only reads and writes the first 104 TSS bytes during a task switch, this restriction only applies to those bytes. The simplest approach is to align the TSS on a page boundary so that all critical bytes are either present or not present. Then, if a page fault occurs when the TSS is accessed, it occurs before

the first byte is read. If the page fault occurs after a portion of the TSS is read, the fault is unrecoverable.

I/O-Permission Bitmap. The I/O-permission bitmap (IOPB) allows system software to grant less-privileged programs access to individual I/O ports, overriding the effect of RFLAGS.IOPL for those devices. When an I/O instruction is executed, the processor checks the IOPB only if the processor is in virtual x86 mode or the CPL is greater than the RFLAGS.IOPL field. Each bit in the IOPB corresponds to a byte I/O port. A word I/O port corresponds to two consecutive IOPB bits, and a doubleword I/O port corresponds to four consecutive IOPB bits. Access is granted to an I/O port of a given size when *all* IOPB bits corresponding to that port are clear. If any bits are set, a #GP occurs.

The IOPB is located in the TSS, as shown by the example in Figure 12-7 on page 368. Each TSS can have a different copy of the IOPB, so access to individual I/O devices can be granted on a task-by-task basis. The I/O-permission bitmap base-address field located at byte 66h in the TSS is an offset into the TSS locating the start of the IOPB. If all 64K IO ports are supported, the IOPB base address must not be greater than 0DFFFh, otherwise accesses to the bitmap cause a #GP to occur. An extra byte must be present after the last IOPB byte. This byte must have all bits set to 1 (0FFh). This allows the processor to read two IOPB bytes each time an I/O port is accessed. By reading two IOPB bytes, the processor can check all bits when unaligned, multi-byte I/O ports are accessed.

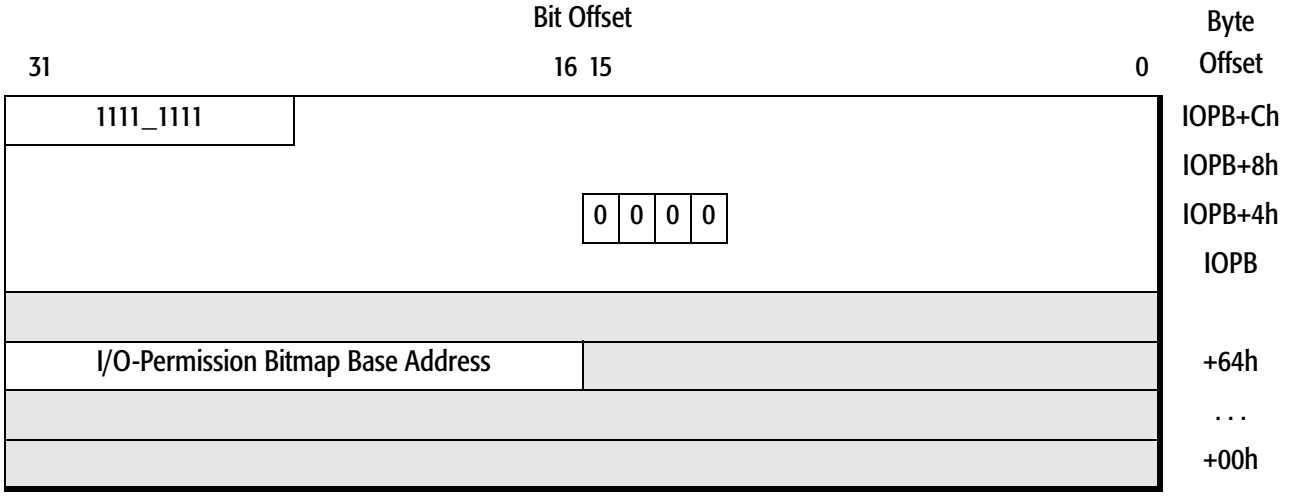


Figure 12-7. I/O-Permission Bitmap Example

Bits in the IOPB sequentially correspond to I/O port addresses. The example in Figure 12-7 shows bits 12 through 15 in the second doubleword of the IOPB cleared to 0. Those bit positions correspond to byte I/O ports 44h through 47h, or alternatively, doubleword I/O port 44h. Because the bits are cleared to zero, software running at any privilege level can access those I/O ports.

By adjusting the TSS limit, it may happen that some ports in the I/O-address space have no corresponding IOPB entry. Ports not represented by the IOPB will cause a #GP exception. Referring again to Figure 12-7, the last IOPB entry is at bit 23 in the fourth IOPB doubleword, which corresponds to I/O port 77h. In this example, all ports from 78h and above will cause a #GP exception, as if their permission bit was set to 1.

12.2.5 64-Bit Task State Segment

Although the hardware task-switching mechanism is not supported in long mode, a 64-bit task state segment (TSS) must still exist. System software must create at least one 64-bit TSS for use after activating long mode, and it must execute the LTR instruction, *in 64-bit mode*, to load the TR register with a pointer to the 64-bit TSS that serves both 64-bit-mode programs and compatibility-mode programs.

The legacy TSS contains several fields used for saving and restoring processor-state information. The legacy fields include general-purpose register, EFLAGS, CR3 and segment-selector

register state, among others. Those legacy fields are not supported by the 64-bit TSS. System software must save and restore the necessary processor-state information required by the software-multitasking implementation (if multitasking is supported). Figure 12-8 on page 370 shows the format of a 64-bit TSS.

The 64-bit TSS holds several pieces of information important to long mode that are not directly related to the task-switch mechanism:

- *RSPn*—Bytes 1Bh–04h. The full 64-bit canonical forms of the stack pointers (RSP) for privilege levels 0 through 2.
- *ISTn*—Bytes 5Bh–24h. The full 64-bit canonical forms of the interrupt-stack-table (IST) pointers. See "Interrupt-Stack Table" on page 290 for a description of the IST mechanism.
- *I/O Map Base Address*—Bytes 67h–66h. The 16-bit offset to the I/O-permission bit map from the 64-bit TSS base. The function of this field is identical to that in a legacy 32-bit TSS. See "I/O-Permission Bitmap" on page 367 for more information.

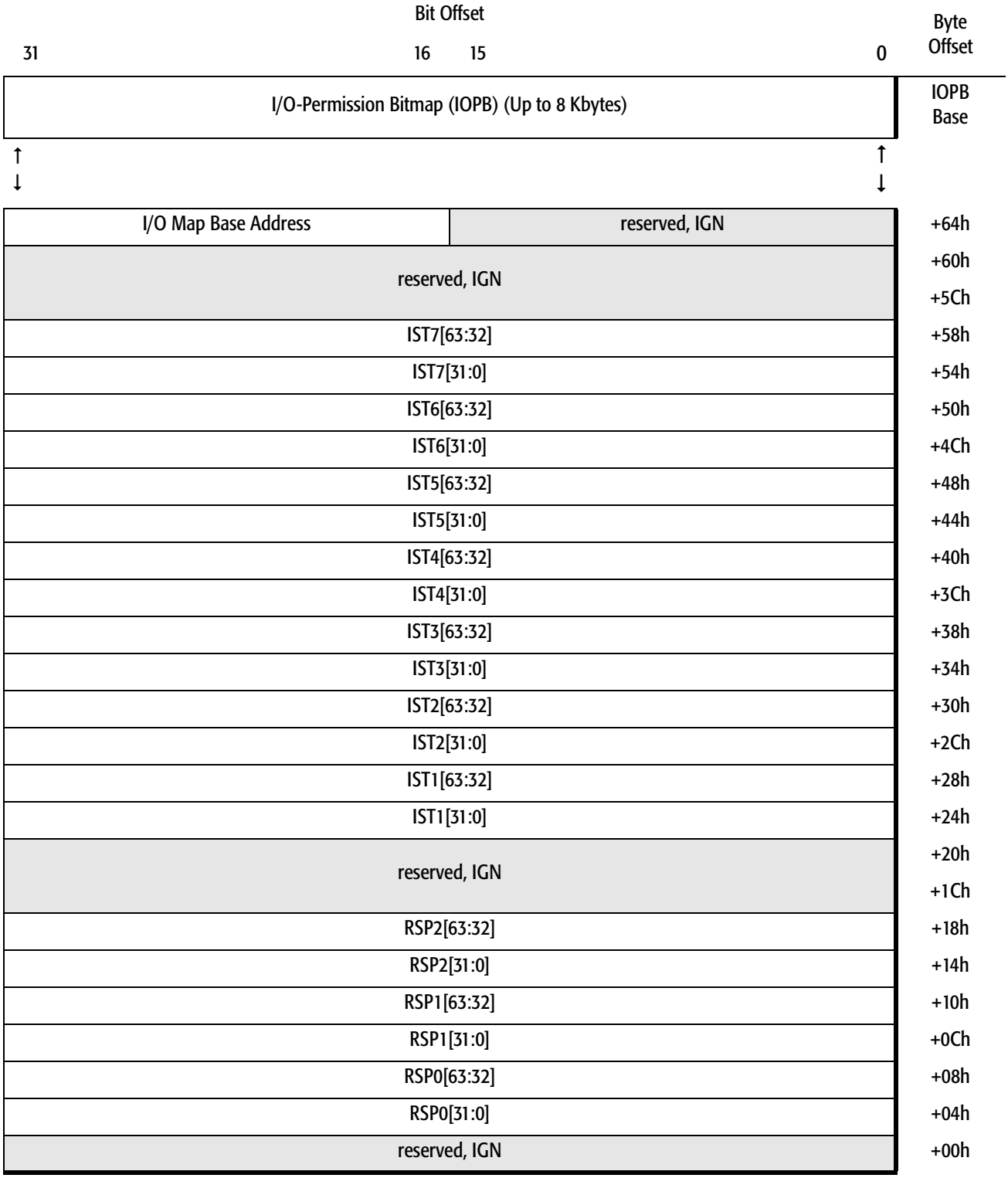


Figure 12-8. Long Mode TSS Format

12.2.6 Task Gate Descriptor (Legacy Mode Only)

Task-gate descriptors hold a selector reference to a TSS and are used to control access between tasks. Unlike a TSS descriptor or other gate descriptors, a task gate can be located in any of the three descriptor tables (GDT, LDT, and IDT). Figure 12-9 shows the format of a task-gate descriptor.

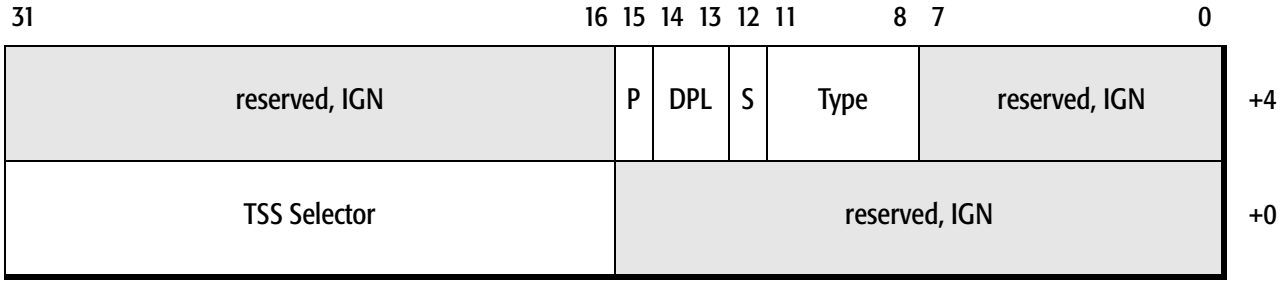


Figure 12-9. Task-Gate Descriptor, Legacy Mode Only

The task-gate descriptor fields are:

- *System (S) and Type*—Bits 12 and 11–8 (respectively) of byte +4. These bits are encoded by software as 00101b to indicate a task-gate descriptor type.
- *Present (P)*—Bit 15 of byte +4. The segment-present bit indicates the segment referenced by the gate descriptor is loaded in memory. If a reference is made to a segment 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.
- *Descriptor Privilege-Level (DPL)*—Bits 14–13 of byte +4. The DPL field indicates the gate-descriptor privilege level. DPL can be set to any value from 0 to 3, with 0 specifying the most privilege and 3 the least privilege.

12.3 Hardware Task-Management in Legacy Mode

This section describes the operation of the task-switch mechanism when the processor is running in legacy mode. None of these features are supported in long mode (either compatibility mode or 64-bit mode).

12.3.1 Task Memory-Mapping

The hardware task-switch mechanism gives system software a great deal of flexibility in managing the sharing and isolation of memory—both virtual (linear) and physical—between tasks.

Segmented Memory. The segmented memory for a task consists of the segments that are loaded during a task switch and any segments that are later accessed by the task code. The hardware task-switch mechanism allows tasks to either share segments with other tasks, or to access segments in isolation from one another. Tasks that share segments actually share a virtual-address (linear-address) space, but they do not necessarily share a physical-address space. When paging is enabled, the virtual-to-physical mapping for each task can differ, as is described in the following section. Shared segments *do share* physical memory when paging is disabled, because virtual addresses are used as physical addresses.

A number of options are available to system software that shares segments between tasks:

- Sharing segment descriptors using the GDT. All tasks have access to the GDT, so it is possible for segments loaded in the GDT to be shared among tasks.
- Sharing segment descriptors using a single LDT. Each task has its own LDT, and that LDT selector is automatically saved and restored in the TSS by the processor during task switches. Tasks, however, can share LDTs simply by storing the same LDT selector in multiple TSSs. Using the LDT to manage segment sharing and segment isolation provides more flexibility to system software than using the GDT for the same purpose.
- Copying shared segment descriptors into multiple LDTs. Segment descriptors can be copied by system software into multiple LDTs that are otherwise not shared between tasks. Allowing segment sharing at the segment-descriptor level, rather than the LDT level or GDT level, provides the greatest flexibility to system software.

In all three cases listed above, the actual data and instructions are shared between tasks only when the tasks' virtual-to-physical address mappings are identical.

Paged Memory. Each task has its own page-translation table base-address (CR3) register, and that register is automatically saved and restored in the TSS by the processor during task switches. This allows each task to point to its own set of page-translation tables, so that each task can translate virtual addresses to physical addresses independently. Page translation must be enabled for changes in CR3 values to have an effect on virtual-

to-physical address mapping. When page translation is disabled, the tables referenced by CR3 are ignored, and virtual addresses are equivalent to physical addresses.

12.3.2 Switching Tasks

The hardware task-switch mechanism transfers program control to a new task when any of the following occur:

- A CALL or JMP instruction with a selector operand that references a task gate is executed. The task gate can be located in either the LDT or GDT.
- A CALL or JMP instruction with a selector operand that references a TSS descriptor is executed. The TSS descriptor must be located in the GDT.
- A software-interrupt instruction (INT*n*) is executed that references a task gate located in the IDT.
- An exception or external interrupt occurs, and the vector references a task gate located in the IDT.
- An IRET is executed while the EFLAGS.NT bit is set to 1, indicating that a return is being performed from an inner-level task to an outer-level task. The new task is referenced using the selector stored in the current-task link field. See "Nesting Tasks" on page 377 for additional information. The RET instruction *cannot* be used to switch tasks.

When a task switch occurs, the following operations are performed automatically by the processor:

- The processor performs privilege-checking to determine whether the currently-executing program is allowed to access the target task. If this check fails, the task switch is aborted without modifying the processor state, and a general-protection exception (#GP) occurs. The privilege checks performed depend on the cause of the task switch:
  - If the task switch is initiated by a CALL or JMP instruction through a TSS descriptor, the processor checks that both the currently-executing program CPL and the TSS-selector RPL are numerically less-than or equal-to the TSS-descriptor DPL.
  - If the task switch takes place through a task gate, the CPL and task-gate RPL are compared with the task-gate DPL, and no comparison is made using the TSS-descriptor DPL. See "Task Switches Using Task Gates" on page 375.

  - Software interrupts, hardware interrupts, and exceptions all transfer control without checking the task-gate DPL.
  - The IRET instruction transfers control without checking the TSS-descriptor DPL.
- The processor performs limit-checking on the target TSS descriptor to verify that the TSS limit is greater than or equal to 67h (at least 104 bytes). If this check fails, the task switch is aborted without modifying the processor state, and an invalid-TSS exception (#TS) occurs.
- The current-task state is saved in the TSS. This includes the next-instruction pointer (EIP), EFLAGS, the general-purpose registers, and the segment-selector registers.

  Up to this point, any exception that occurs aborts the task switch without changing the processor state. From this point forward, any exception that occurs does so in the context of the new task. If an exception occurs in the context of the new task during a task switch, the processor finishes loading the new-task state without performing additional checks. The processor transfers control to the #TS handler after this state is loaded, but before the first instruction is executed in the new task. When a #TS occurs, it is possible that some of the state loaded by the processor did not participate in segment access checks. The #TS handler must verify that all segments are accessible before returning to the interrupted task.
- The task register (TR) is loaded with the new-task TSS selector, and the hidden portion of the TR is loaded with the new-task descriptor. The TSS now referenced by the processor is that of the new task.
- The current task is marked as busy. The previous task is marked as available or remains busy, based on the type of linkage. See “Nesting Tasks” on page 377 for more information.
- CR0.TS is set to 1. This bit can be used to save other processor state only when it becomes necessary. For more information, see the next section, “Saving Other Processor State.”
- The new-task state is loaded from the TSS. This includes the next-instruction pointer (EIP), EFLAGS, the general-purpose registers, and the segment-selector registers. The processor clears the segment-descriptor present (P) bits (in the hidden portion of the segment registers) to prevent

access into the new segments, until the task switch completes successfully.

- The LDTR and CR3 registers are loaded from the TSS, changing the virtual-to-physical mapping from that of the old task to the new task. Because this is done in the middle of accessing the new TSS, system software must guarantee that TSS addresses are translated identically in all tasks.
- The descriptors for all previously-loaded segment selectors are loaded into the hidden portion of the segment registers. This sets or clears the P bits for the segments as specified by the new descriptor values.

If the above steps complete successfully, the processor begins executing instructions in the new task beginning with the instruction referenced by the CS:EIP far pointer loaded from the new TSS. The privilege level of the new task is taken from the new CS segment selector's RPL.

Saving Other Processor State. The processor does not automatically save the registers used by the media or x87 instructions. Instead, the processor sets CR0.TS to 1 during a task switch. Later, when an attempt is made to execute any of the media or x87 instructions while TS=1, a device-not-available exception (#NM) occurs. System software can then save the previous state of the media and x87 registers and clear the CR0.TS bit to 0 before executing the next media/x87 instruction. As a result, the media and x87 registers are saved only when necessary after a task switch.

12.3.3 Task Switches Using Task Gates

When a control transfer to a new task occurs through a task gate, the processor reads the task-gate DPL (DPL_G) from the task-gate descriptor. Two privilege checks, both of which must pass, are performed on DPL_G before the task switch can occur successfully:

- The processor compares the CPL with DPL_G. The CPL must be numerically *less than or equal to* DPL_G for this check to pass. In other words, the following expression must be true: $CPL \leq DPL_G$.
- The processor compares the RPL in the task-gate selector with DPL_G. The RPL must be numerically *less than or equal to* DPL_G for this check to pass. In other words, the following expression must be true: $RPL \leq DPL_G$.

Unlike call-gate control transfers, the processor does not read the DPL from the target TSS descriptor (DPL_S) and compare it with the CPL when a task gate is used.

Figure 12-10 on page 377 shows two examples of task-gate privilege checks. In Example 1, the privilege checks pass:

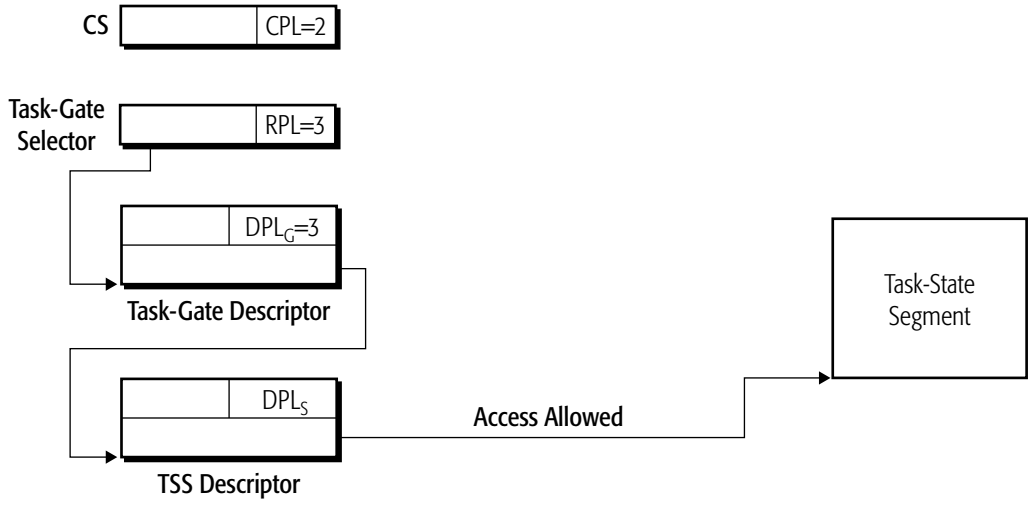
- The task-gate DPL (DPL_G) is at the lowest privilege (3), specifying that software running at any privilege level (CPL) can access the gate.
- The selector referencing the task gate passes its privilege check because the RPL is numerically less than or equal to DPL_G.

In Example 2, both privilege checks fail:

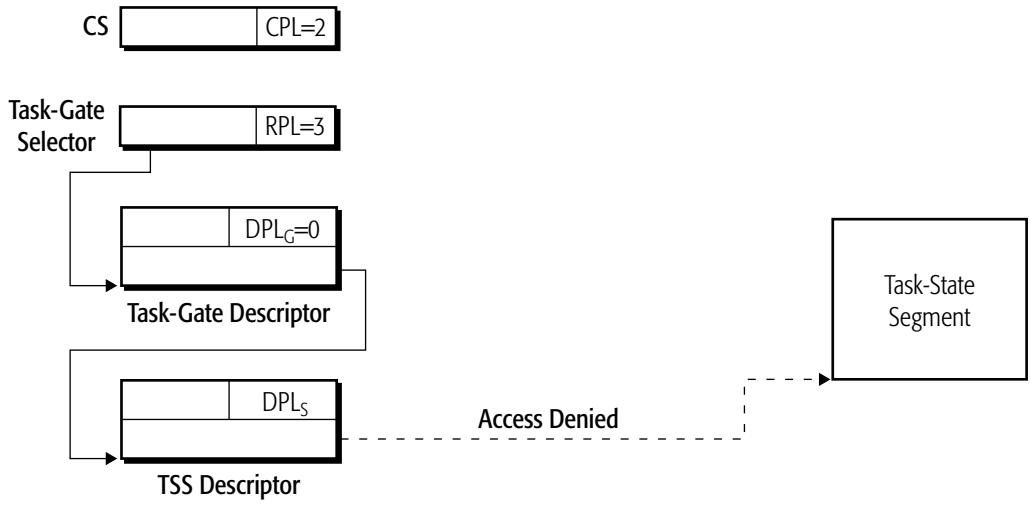
- The task-gate DPL (DPL_G) specifies that only software at privilege-level 0 can access the gate. The current program does not have enough privilege to access the task gate, because its CPL is 2.
- The selector referencing the task-gate descriptor does not have a high enough privilege to complete the reference. Its RPL is numerically greater than DPL_G.

Although both privilege checks failed in the example, if only one check fails, access into the target task is denied.

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 using a task gate in long mode causes a general-protection exception (#GP) to occur.



Example 1: Privilege Check Passes



Example 2: Privilege Check Fails

513-255.eps

Figure 12-10. Privilege-Check Examples for Task Gates

12.3.4 **Nesting Tasks**

The hardware task-switch mechanism supports task nesting through the use of EFLAGS *nested-task* (NT) bit and the TSS link-field. The manner in which these fields are updated and used during a task switch depends on how the task switch is initiated:

- The JMP instruction does not update EFLAGS.NT or the TSS link-field. Task nesting is not supported by the JMP instruction.
- The CALL instruction, INT*n* instructions, interrupts, and exceptions can only be performed from outer-level tasks to inner-level tasks. All of these operations set the EFLAGS.NT bit for the new task to 1 during a task switch, and copy the selector for the previous task into the new-task link field.
- An IRET instruction which returns to another task only occurs when the EFLAGS.NT bit for the current task is set to 1, and only can be performed from an inner-level task to an outer-level task. When an IRET results in a task switch, the new task is referenced using the selector stored in the current-TSS link field. The EFLAGS.NT bit for the current task is cleared to 0 during the task switch.

Table 12-1 summarizes the effect various task-switch initiators have on EFLAGS.NT, the TSS link-field, and the TSS-busy bit. (For more information on the busy bit, see the next section, "Preventing Recursion.")

Table 12-1. Effects of Task Nesting

<table>
<tr><th rowspan="2">Task-Switch Initiator</th><th colspan="3">Old Task</th><th colspan="3">New Task</th></tr>
<tr><th>EFLAGS.NT</th><th>Link (Selector)</th><th>Busy</th><th>EFLAGS.NT</th><th>Link (Selector)</th><th>Busy</th></tr>
<tr><td>JMP</td><td>–</td><td>–</td><td>Clear to 0 (was 1)</td><td>–</td><td>–</td><td>Set to 1</td></tr>
<tr><td>CALL
INTn
Interrupt
Exception</td><td colspan="2">–</td><td>–
(Was 1)</td><td>Set to 1</td><td>Old Task</td><td>Set to 1</td></tr>
<tr><td>IRET</td><td>Clear to 0 (was 1)</td><td>–</td><td>Clear to 0 (was 1)</td><td colspan="3">–</td></tr>
<tr><td colspan="7">Note:
"–" indicates no change is made.</td></tr>
</table>

Programs running at any privilege level can set EFLAGS.NT to 1 and execute the IRET instruction to transfer control to another task. System software can keep control over improperly nested-task switches by initializing the link field of all TSSs

that it creates. That way, improperly nested-task switches always transfer control to a known task.

Preventing Recursion. Task recursion is not allowed by the hardware task-switch mechanism. If recursive-task switches were allowed, they would replace a previous task-state image with a newer image, discarding the previous information. To prevent recursion from occurring, the processor uses the busy bit located in the TSS-descriptor type field (bit 9 of byte +4). Use of this bit depends on how the task switch is initiated:

- The JMP instruction clears the busy bit in the old task to 0 and sets the busy bit in the new task to 1. A general-protection exception (#GP) occurs if an attempt is made to JMP to a task with a set busy bit.
- The CALL instruction, INT*n* instructions, interrupts, and exceptions set the busy bit in the new task to 1. The busy bit in the old task remains set to 1, preventing recursion through task-nesting levels. A general-protection exception (#GP) occurs if an attempt is made to switch to a task with a set busy bit.
- An IRET to another task (EFLAGS.NT must be 1) clears the busy bit in the old task to 0. The busy bit in the new task is not altered, because it was already set to 1.

Table 12-1 on page 378 summarizes the effect various task-switch initiators have on the TSS-busy bit.

13 Debug and Performance Resources

Testing, debug, and performance optimization consume a significant portion of the time needed to develop a new computer or software product and move it successfully into production. To stay competitive, product developers need tools that allow them to rapidly detect, isolate, and correct problems before a product is shipped. The goal of the debug and performance features incorporated into processor implementations of the AMD64 architecture is to support the tool chain solutions used in software-product and hardware-product development.

The debug and performance resources that can be supported by AMD64 architecture implementations include:

- *Software Debug*—The AMD64 architecture supports the legacy software-debug facilities, including the debug registers (DR0–DR7), debug exception, and breakpoint exception. Additional features are provided using model-specific registers (MSRs). These registers are used to set breakpoints on branches, interrupts, and exceptions and to single step from one branch to the next. The software-debug capability is described in "Software-Debug Resources" on page 382.
- *Performance Monitoring*—Model-specific registers (MSRs) are provided to monitor events within an implementation of the AMD64 architecture. A set of control registers allow the selection of events to be monitored and a corresponding set of counter registers track the frequency of monitored events. These features are used to support software-performance and hardware-performance tuning. Performance monitoring is described in "Performance Optimization" on page 401.

A given processor implementation may include additional debug capabilities that allow monitoring program execution and manipulation of processor state to be manipulated. These resources are typically intended for BIOS and platform development and, if available, are described in the *BIOS and Kernel Developer's Guide for AMD Athlon™ 64 and AMD Opteron™ Processors* (order# 26094) for information specific to a particular processor.

13.1 Software-Debug Resources

Software can program breakpoints into the debug registers, causing a #DB exception when matches occur on instruction-memory addresses, data-memory addresses, or I/O addresses. The breakpoint exception (#BP) is also supported to allow software to set breakpoints by placing INT3 instructions in the instruction memory for a program. Program control is transferred to the breakpoint exception (#BP) handler when an INT3 instruction is executed.

In addition to the debug features supported by the debug registers (DR0–DR7), the processor also supports features supported by model-specific registers (MSRs). Together, these capabilities provide a rich set of breakpoint conditions, including:

- *Breakpoint On Address Match*—Breakpoints occur when the address stored in a address-breakpoint register matches the address of an instruction or data reference. Up to four address-match breakpoint conditions can be set by software.
- *Single Step All Instructions*—Breakpoints can be set to occur on every instruction, allowing a debugger to examine the contents of registers as a program executes.
- *Single Step Control Transfers*—Breakpoints can be set to occur on control transfers, such as calls, jumps, interrupts, and exceptions. This can allow a debugger to narrow a problem search to a specific section of code before enabling single stepping of all instructions.
- *Breakpoint On Any Instruction*—Breakpoints can be set on any specific instruction using either the address-match breakpoint condition or using the INT3 instruction to force a breakpoint when the instruction is executed.
- *Breakpoint On Task Switch*—Software forces a #DB exception to occur when a task switch is performed to a task with the T bit in the TSS set to 1. Debuggers can use this capability to enable or disable debug conditions for a specific task.

Problem areas can be identified rapidly using the information supplied by the debug registers when breakpoint conditions occur:

- Special conditions that cause a #DB exception are recorded in the DR6 debug-status register, including breakpoints due to task switches and single stepping. The DR6 register also

identifies which address-breakpoint register (DR0–DR3) caused a #DB exception due to an address match. When combined with the DR7 debug-control register settings, the cause of a #DB exception can be identified.

- To assist in analyzing the instruction sequence a processor follows in reaching its current state, the source and destination addresses of control-transfer events are saved by the processor. These include branches (calls and jumps), interrupts, and exceptions. Debuggers can use this information to narrow a problem search to a specific section of code before single stepping all instructions.

13.1.1 Debug Registers

The AMD64 architecture supports the legacy debug registers, DR0–DR7. These registers are expanded to 64 bits by the AMD64 architecture. In legacy mode and in compatibility mode, only the lower 32 bits are used. In these modes, writes to a debug register fill the upper 32 bits with zeros, and reads from a debug register return only the lower 32 bits. In 64-bit mode, all 64 bits of the debug registers are read and written. Operand-size prefixes are ignored.

The debug registers can be read and written only when the current-protection level (CPL) is 0 (most privileged). Attempts to read or write the registers at a lower-privilege level (CPL>0) cause a general-protection exception (#GP).

Several debug registers described below are model-specific registers (MSRs). See "Software-Debug MSRs" on page 442 for a listing of the debug-MSR numbers and their reset values. Some processor implementations include additional MSRs used to support implementation-specific software debug features. For more information on these registers and their capabilities, refer to the BIOS writer's guide for the implementation.

Address-Breakpoint Registers (DR0-DR3). Figure 13-1 on page 384 shows the format of the four address-breakpoint registers, DR0-DR3. Software can load a virtual (linear) address into any of the four registers, and enable breakpoints to occur when the address matches an instruction or data reference. The MOV DR*n* instructions *do not* check that the virtual addresses loaded into DR0–DR3 are in canonical form. Breakpoint conditions are enabled using the debug-control register, DR7 (see "Debug-Control Register (DR7)" on page 386).

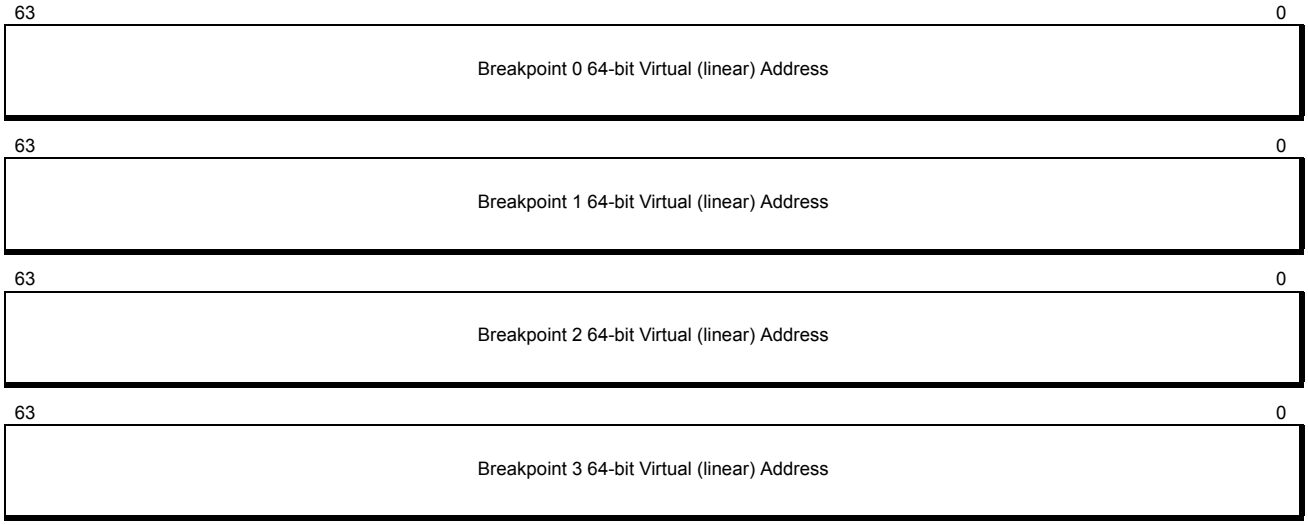
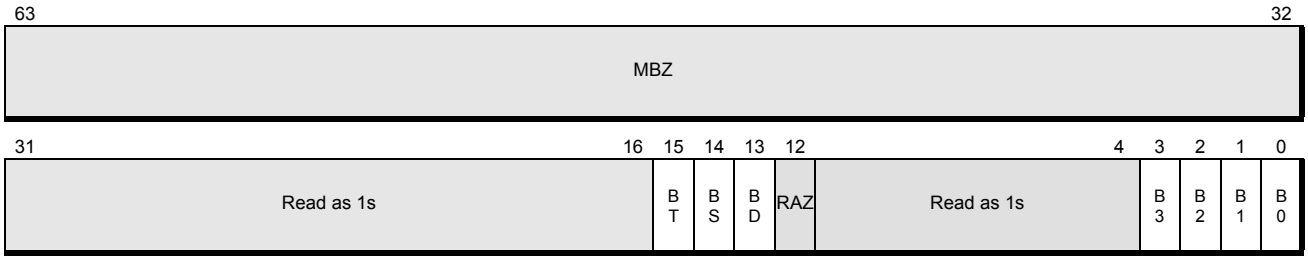


Figure 13-1. Address-Breakpoint Registers (DR0–DR3)

Reserved Debug Registers (DR4, DR5). The DR4 and DR5 registers are reserved and should not be used by software. These registers are aliased to the DR6 and DR7 registers, respectively. When the debug extensions are enabled (CR4.DE=1) attempts to access these registers cause an invalid-opcode exception (#UD).

Debug-Status Register (DR6). Figure 13-2 on page 385 shows the format of the debug-status register, DR6. Debug status is loaded into DR6 when an enabled debug condition is encountered that causes a #DB exception. This register is never cleared by the processor and must be cleared by software after the contents have been read.



Bits	Mnemonic	Description	R/W
63–16	reserved	MBZ	
15	BT	Breakpoint Task Switch	R/W
14	BS	Breakpoint Single Step	R/W
13	BD	Breakpoint Debug Access Detected	R/W
12	reserved	Read as Zero	
11-4	reserved	Read as 1s	
3	B3	Breakpoint #3 Condition Detected	R/W
2	B2	Breakpoint #2 Condition Detected	R/W
1	B1	Breakpoint #1 Condition Detected	R/W
0	B0	Breakpoint #0 Condition Detected	R/W

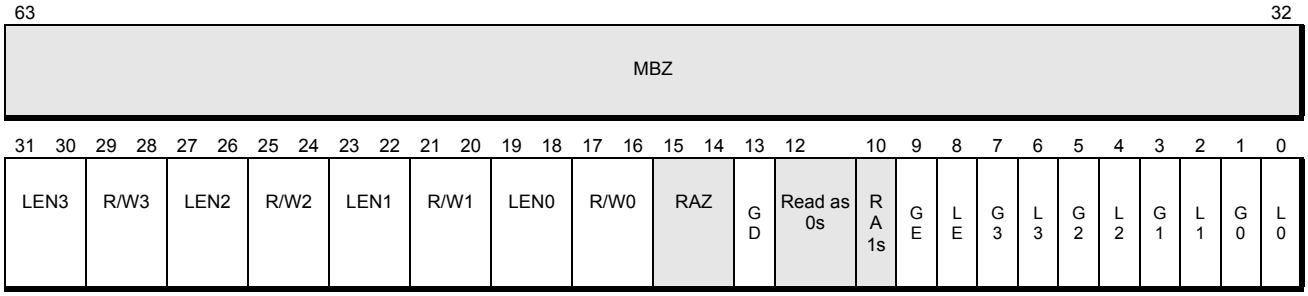
Figure 13-2. Debug-Status Register (DR6)

The fields within the DR6 register are set by the processor and must be cleared by software. These fields are:

- *Breakpoint-Condition Detected (B3–B0)*—Bits 3–0. The processor individually sets these bits to 1 when the corresponding address-breakpoint register detects a breakpoint condition, as specified by the DR7 R/W*n* and LEN*n* controls. For example, B1 (bit 1) is set to 1 if an address-breakpoint condition is detected by DR1. These bits are only set if the #DB exception is enabled by the L*n* and G*n* control bits in DR7.
- *Debug-Register-Access Detected (BD)*—Bit 13. The processor sets this bit to 1 if software accesses any debug register (DR0–DR7) while the general-detect condition is enabled (DR7.GD=1).
- *Single Step (BS)*—Bit 14. The processor sets this bit to 1 if the #DB exception occurs as a result of single-step mode (rFLAGS.TF=1). Single-step mode has the highest-priority among debug exceptions. Other status bits within the DR6 register can be set by the processor along with the BS bit.
- *Task-Switch (BT)*—Bit 15. The processor sets this bit to 1 if the #DB exception occurred as a result of task switch to a task with a TSS T-bit set to 1.

All remaining bits in the DR6 register are reserved. Reserved bits 31–16 and 11–4 must all be set to 1, while reserved bit 12 must be cleared to 0. In 64-bit mode, the upper 32 bits of DR6 are reserved and must be written with zeros. Writing a 1 to any of the upper 32 bits results in a general-protection exception, #GP(0).

Debug-Control Register (DR7). Figure 13-3 shows the format of the debug-control register, DR7. DR7 is used to establish the breakpoint conditions for the address-breakpoint registers (DR0–DR3) and to enable debug exceptions for each address-breakpoint register individually. DR7 is also used to enable the general-detect breakpoint condition.



Bits	Mnemonic	Description	R/W
63–32	reserved	MBZ	
31-30	LEN3	Length of Breakpoint #3	R/W
29-28	R/W3	Type of Transaction(s) to Trap	R/W
27-26	LEN2	Length of Breakpoint #2	R/W
25-24	R/W2	Type of Transaction(s) to Trap	R/W
23-22	LEN1	Length of Breakpoint #1	R/W
21-20	R/W1	Type of Transaction(s) to Trap	R/W
19-18	LEN0	Length of Breakpoint #0	R/W
17-16	R/W0	Type of Transaction(s) to Trap	R/W
15-14	reserved	Read as 0s	
13	GD	General Detect Enabled	R/W
12-11	reserved	Read as 0s	
10	reserved	Read as 1	
9	GE	Global Exact Breakpoint Enabled	R/W
8	LE	Local Exact Breakpoint Enabled	R/W
7	G3	Global Exact Breakpoint #3 Enabled	R/W
6	L3	Local Exact Breakpoint #3 Enabled	R/W
5	G2	Global Exact Breakpoint #2 Enabled	R/W
4	L2	Local Exact Breakpoint #2 Enabled	R/W
3	G1	Global Exact Breakpoint #1 Enabled	R/W
2	L1	Local Exact Breakpoint #1 Enabled	R/W
1	G0	Global Exact Breakpoint #0 Enabled	R/W
0	L0	Local Exact Breakpoint #0 Enabled	R/W

Figure 13-3. Debug-Control Register (DR7)

The fields within the DR7 register are all read/write. These fields are:

- *Local-Breakpoint Enable (L3–L0)*—Bits 6, 4, 2, and 0 (respectively). Software individually sets these bits to 1 to enable debug exceptions to occur when the corresponding address-breakpoint register (DR*n*) detects a breakpoint condition while executing the *current* task. For example, if L1 (bit 2) is set to 1 and an address-breakpoint condition is detected by DR1, a #DB exception occurs. These bits are cleared to 0 by the processor when a hardware task-switch occurs.
- *Global-Breakpoint Enable (G3–G0)*—Bits 7, 5, 3, and 1 (respectively). Software sets these bits to 1 to enable debug exceptions to occur when the corresponding address-breakpoint register (DR*n*) detects a breakpoint condition while executing *any* task. For example, if G1 (bit 3) is set to 1 and an address-breakpoint condition is detected by DR1, a #DB exception occurs. These bits are never cleared to 0 by the processor.
- *Local-Enable (LE)*—Bit 8. Software sets this bit to 1 in legacy implementations to enable exact breakpoints while executing the *current* task. This bit is ignored by implementations of the AMD64 architecture. All breakpoint conditions, except certain string operations preceded by a repeat prefix, are exact.
- *Global-Enable (GE)*—Bit 9. Software sets this bit to 1 in legacy implementations to enable exact breakpoints while executing *any* task. This bit is ignored by implementations of the AMD64 architecture. All breakpoint conditions, except certain string operations preceded by a repeat prefix, are exact.
- *General-Detect Enable (GD)*—Bit 13. Software sets this bit to 1 to cause a debug exception to occur when an attempt is made to execute a MOV DR*n* instruction to any debug register (DR0–DR7). This bit is cleared to 0 by the processor when the #DB handler is entered, allowing the handler to read and write the DR*n* registers. The #DB exception occurs before executing the instruction, and DR6.BD is set by the processor. Software debuggers can use this bit to prevent the currently-executing program from interfering with the debug operation.

- *Read/Write (R/W3–R/W0)*—Bits 29–28, 25–24, 21–20, and 17–16 (respectively). Software sets these fields to control the breakpoint conditions used by the corresponding address-breakpoint registers (DR*n*). For example, control-field R/W1 (bits 21–20) controls the breakpoint conditions for the DR1 register. The R/W*n* control-field encodings specify the following conditions for an address-breakpoint to occur:
  - 00—Only on instruction execution.
  - 01—Only on data write.
  - 10—This encoding is further qualified by CR4.DE as follows:
    - CR4.DE=0—Condition is undefined.
    - CR4.DE=1—Only on I/O read or I/O write.
  - 11—Only on data read or data write.
- *Length (LEN3–LEN0)*—Bits 31–30, 27–26, 23–22, and 19–18 (respectively). Software sets these fields to control the range used in comparing a memory address with the corresponding address-breakpoint register (DR*n*). For example, control-field LEN1 (bits 23–22) controls the breakpoint-comparison range for the DR1 register.

  The value in DR*n* defines the low-end of the address range used in the comparison. LEN*n* is used to mask the low-order address bits in the corresponding DR*n* register so that they are not used in the address comparison. To work properly, breakpoint boundaries must be aligned on an address corresponding to the range size specified by LEN*n*. The LEN*n* control-field encodings specify the following address-breakpoint-comparison ranges:
  - 00—1 byte.
  - 01—2 byte, must be aligned on a word boundary.
  - 10—8 byte, must be aligned on a quadword boundary. (Long mode only; otherwise undefined.)
  - 11—4 byte, must be aligned on a doubleword boundary.

  If the R/W*n* field is used to specify instruction breakpoints (R/W*n*=00), the corresponding LEN*n* field must be set to 00. Setting LEN*n* to any other value produces undefined results.

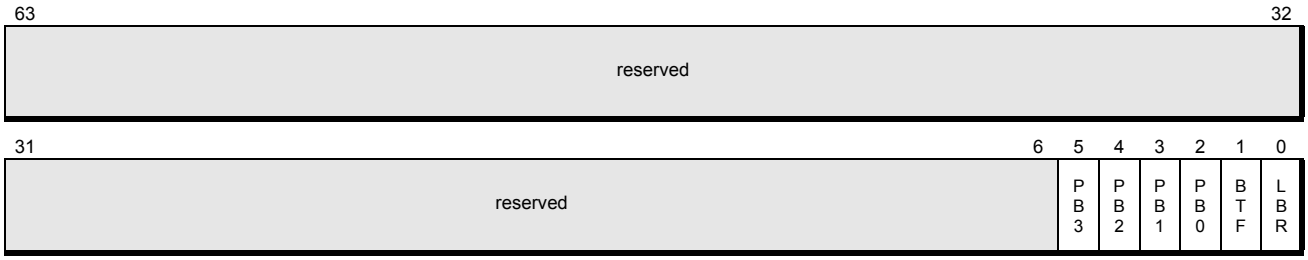
All remaining bits in the DR7 register are reserved. Reserved bits 15–14 and 12–11 must all be cleared to 0, while reserved bit 10 must be set to 1. In 64-bit mode, the upper 32 bits of DR7 are reserved and must be written with zeros. Writing a 1 to any of

the upper 32 bits results in a general-protection #GP(0) exception.

64-Bit-Mode Extended Debug Registers. In 64-bit mode, additional encodings for debug registers are available. 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 Prefix-Byte Fields" in Volume 3. These additional encodings enable the processor to address DR8–DR15.

Access to the DR8–DR15 registers is implementation-dependent. The architecture does not require any of these extended debug registers to be implemented. Any attempt to access an unimplemented register results in an invalid-opcode exception (#UD).

Debug-Control MSR (DebugCtlMSR). Figure 13-4 shows the format of the debug-control MSR, DebugCtlMSR. DebugCtlMSR provides additional debug controls over control-transfer recording and single stepping, and external-breakpoint reporting and trace messages. DebugCtlMSR is an MSR and is read and written using the RDMSR and WRMSR instructions.



Bits	Mnemonic	Description	R/W
63-6	reserved		
5	PB3	Performance Monitoring Pin Control	R/W
4	PB2	Performance Monitoring Pin Control	R/W
3	PB1	Performance Monitoring Pin Control	R/W
2	PB0	Performance Monitoring Pin Control	R/W
1	BTF	Branch Single Step	R/W
0	LBR	Last-Branch Record	R/W

Figure 13-4. Debug-Control MSR (DebugCtlMSR)

The fields within the DebugCtlMSR register are:

- *Last-Branch Record (LBR)*—Bit 0, read/write. Software sets this bit to 1 to cause the processor to record the source and target addresses of the last control transfer taken before a debug exception occurs. The recorded control transfers

include branch instructions, interrupts, and exceptions. When a debug exception occurs, the processor clears this bit before transferring control to the debug-exception handler. This prevents control transfers into the debug-exception handler from recording the source and target addresses.

- *Branch Single Step (BTF)*—Bit 1, read/write. Software uses this bit to change the behavior of the rFLAGS.TF bit. When this bit is cleared to 0, the rFLAGS.TF bit controls instruction single stepping, (normal behavior). When this bit is set to 1, the rFLAGS.TF bit controls single stepping on control transfers. The single-stepped control transfers include branch instructions, interrupts, and exceptions. Control-transfer single stepping requires both BTF=1 and rFLAGS.TF=1.
- *Performance-Monitoring/Breakpoint Pin-Control (PBi)*—Bits 5–2, read/write. Software uses these bits to control the type of information reported by the four external performance-monitoring/breakpoint pins on the processor. When a PB*i* bit is cleared to 0, the corresponding external pin (BP*i*) reports performance-monitor information. When a PB*i* bit is set to 1, the corresponding external pin (BP*i*) reports breakpoint information.

All remaining bits in the DebugCtlMSR register are reserved.

Control-Transfer Recording MSRs. Figure 13-5 on page 391 shows the format of the 64-bit control-transfer recording MSRs: LastBranchToIP, LastBranchFromIP, LastExceptionToIP, and LastExceptionFromIP. These registers are loaded automatically by the processor when the DebugCtlMSR.LBR bit is set to 1. These MSRs are read-only.

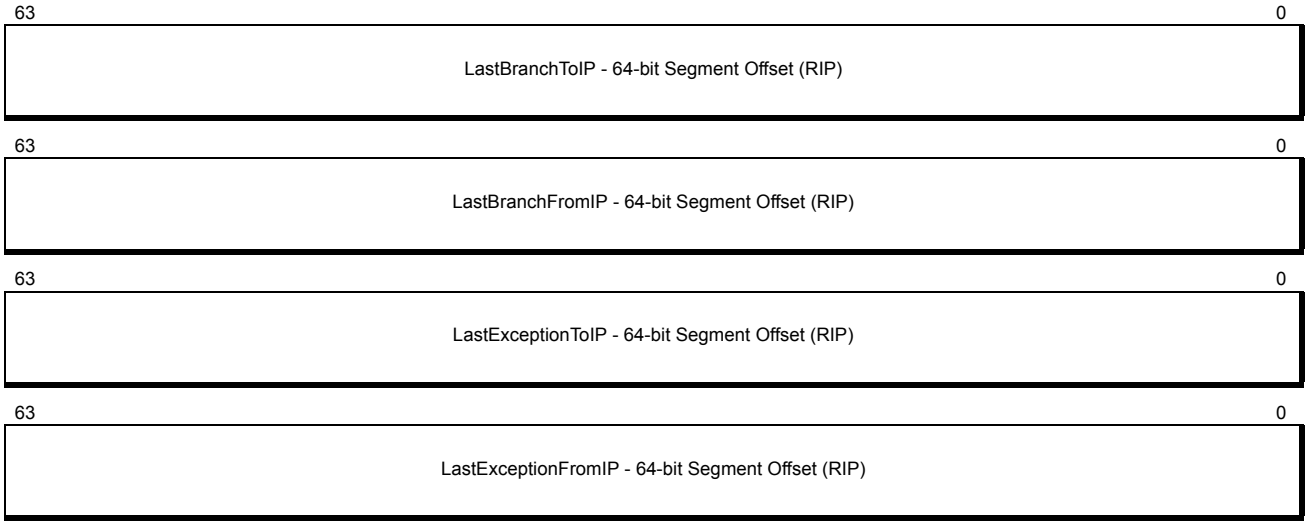


Figure 13-5. Control-Transfer Recording MSRs

When DebugCtlMSR.LBR=1, the control-transfer recording MSRs are updated as follows:

- *LastBranchToIP Register*—The processor loads this register with the segment offset of the first instruction to be executed after a control transfer occurs (also called the *target rIP)*. The control transfers that cause this register to be written are branch instructions, interrupts, and exceptions. In the case of INTn and INTO instructions, interrupts, and exceptions, this register is loaded with the offset of the interrupt or exception handler.
- *LastBranchFromIP Register*—The processor loads this register during a control transfer (branch instructions, interrupt, or exception). In the case of a branch instruction, this register is loaded with the segment offset of the branch instruction. In the case of interrupts and exceptions, this register is loaded with the segment offset of the interrupted instruction. In both cases, the offset loaded is referred to as the *source rIP.*
- *LastExceptionToIP Register*—This register holds the target rIP of the last branch that occurred before the exception or interrupt. The processor loads this register with the contents of the LastBranchToIP register when an interrupt or exception occurs, before updating the LastBranchToIP register as described above.

- *LastExceptionFromIP Register*—This register holds the source rIP of the last branch that occurred before the exception or interrupt. The processor loads this register with the contents of the LastBranchFromIP register when an interrupt or exception occurs, before updating the LastBranchFromIP register as described above.

13.2 Breakpoints

13.2.1 Setting Breakpoints

Breakpoints can be set to occur on either instruction addresses or data addresses using the breakpoint-address registers, DR0–DR3 (DR*n*). The values loaded into these registers represent the breakpoint-location virtual address. The debug-control register, DR7, is used to enable the breakpoint registers and to specify the type of access and the range of addresses that can trigger a breakpoint.

Software enables the DR*n* registers using the corresponding local-breakpoint enable (L*n*) or global-breakpoint enable (G*n*) found in the DR7 register. L*n* is used to enable breakpoints only while the current task is active, and it is cleared by the processor when a task switch occurs. G*n* is used to enable breakpoints for all tasks, and it is never cleared by the processor.

The R/W*n* fields in DR7, along with the CR4.DE bit, specify the type of access required to trigger a breakpoint when an address match occurs on the corresponding DR*n* register. Breakpoints can be set to occur on instruction execution, data reads and writes, and I/O reads and writes. The R/W*n* and CR4.DE encodings used to specify the access type are described on page 388 of “Debug-Control Register (DR7).”

The LEN*n* fields in DR7 specify the size of the address range used in comparison with data or instruction addresses. LEN*n* is used to mask the low-order address bits in the corresponding DR*n* register so that they are not used in the address comparison. Breakpoint boundaries must be aligned on an address corresponding to the range size specified by LEN*n*. Assuming the access type matches the type specified by R/W*n*, a breakpoint occurs if any accessed byte falls within the range specified by LEN*n*. For instruction breakpoints, LEN*n* must specify a single-byte range. The LEN*n* encodings used to

specify the address range are described on page 388 of "Debug-Control Register (DR7)."

Table 13-1 shows several examples of data accesses, and whether or not they cause a #DB exception to occur based on the breakpoint address in DR*n* and the breakpoint-address range specified by LEN*n*. In this table, R/W*n* always specifies read/write access.

Table 13-1. Breakpoint-Setting Examples

Data-Access Address (hexadecimal)	Access Size (bytes)	Byte-Addresses in Data-Access (hexadecimal)	Breakpoint-Address Range (hexadecimal)	Result
DR*n*=F000, LEN*n*=00 (1 Byte)				
EFFB	8	EFFB, EFFC, EFFD, EFFE, EFFF, F000, F001	F000	#DB
EFFE	2	EFFE, EFFF		–
	4	EFFE, EFFF, F000, F001		#DB
F000	1	F000		
F001	2	F001, F002		–
F005	4	F005, F006, F007, F008		
DR*n*=F004, LEN*n*=11 (4 Bytes)				
EFFB	8	EFFB, EFFC, EFFD, EFFE, EFFF, F000, F001	F004–F007	–
EFFE	2	EFFE, EFFF		
	4	EFFE, EFFF, F000, F001		
F000	1	F000		
F001	2	F001, F002		
F005	4	F005, F006, F007, F008		#DB

Note:
"–" indicates no #DB occurs.
R/Wn specifies read/write access.

Table 13-1. Breakpoint-Setting Examples *(continued)*

Data-Access Address (hexadecimal)	Access Size (bytes)	Byte-Addresses in Data-Access (hexadecimal)	Breakpoint-Address Range (hexadecimal)	Result
DR*n*=F005, LEN*n*=10 (8 Bytes)				
EFFB	8	EFFB, EFFC, EFFD, EFFE, EFFF, F000, F001	F000–F007	#DB
EFFE	2	EFFE, EFFF		–
	4	EFFE, EFFF, F000, F001		#DB
F000	1	F000		
F001	2	F001, F002		
F005	4	F005, F006, F007, F008		

Note:
"–" indicates no #DB occurs.
R/Wn specifies read/write access.

13.2.2 Using Breakpoints

A debug exception (#DB) occurs when an enabled-breakpoint condition is encountered during program execution. The debug-handler must check the debug-status register (DR6), the conditions enabled by the debug-control register (DR7), and the debug-control MSR (DebugCtlMSR), to determine the #DB cause. The #DB exception corresponds to interrupt-vector 1. See "#DB—Debug Exception (Vector 1)" on page 248.

Instruction breakpoints and general-detect conditions cause the #DB exception to occur *before* the instruction is executed, while all other breakpoint and single-stepping conditions cause the #DB exception to occur *after* the instruction is executed. Table 13-2 on page 395 summarizes where the #DB exception occurs based on the breakpoint condition.

Table 13-2. Breakpoint Location by Condition

Breakpoint Condition	Breakpoint Location
Instruction	Before Instruction is Executed
General Detect	
Data Write Only	After Instruction is Executed[1]
Data Read or Data Write	
I/O Read or I/O Write	
Single Step[1]	After Instruction is Executed
Task Switch	

Note:
1. *Repeated operations (REP prefix) can breakpoint between iterations.*

Instruction breakpoints and general-detect conditions have a lower interrupt-priority than the other breakpoint and single-stepping conditions (see "Priorities" on page 267). Data-breakpoint conditions on the *previous* instruction occur before an instruction-breakpoint condition on the *next* instruction. However, if instruction and data breakpoints can occur as a result of executing a *single* instruction, the instruction breakpoint occurs first (before the instruction is executed), followed by the data breakpoint (after the instruction is executed).

Instruction Breakpoints. Instruction breakpoints are set by loading a breakpoint-address register (DR*n*) with the desired instruction virtual-address, and then setting the corresponding DR7 fields as follows:

- L*n* or G*n* is set to 1 to enable the breakpoint for either the local task or all tasks, respectively.
- R/W*n* is set to 00b to specify that the contents of DR*n* are to be compared only with the virtual address of the next instruction to be executed.
- LEN*n* must be set to 00b.

When a #DB exception occurs due to an instruction breakpoint-address in DR*n*, the corresponding B*n* field in DR6 is set to 1 to indicate that a breakpoint condition occurred. The breakpoint occurs before the instruction is executed, and the breakpoint-

instruction address is pushed onto the debug-handler stack. If multiple instruction breakpoints are set, the debug handler can use the B*n* field to identify which register caused the breakpoint.

Returning from the debug handler causes the breakpoint instruction to be executed. Before returning from the debug handler, the rFLAGS.RF bit should be set to 1 to prevent a reoccurrence of the #DB exception due to the instruction-breakpoint condition. The processor ignores instruction-breakpoint conditions when rFLAGS.RF=1, until after the next instruction (in this case, the breakpoint instruction) is executed. After the next instruction is executed, the processor clears rFLAGS.RF to 0.

Data Breakpoints. Data breakpoints are set by loading a breakpoint-address register (DR*n*) with the desired data virtual-address, and then setting the corresponding DR7 fields as follows:

- L*n* or G*n* is set to 1 to enable the breakpoint for either the local task or all tasks, respectively.
- R/W*n* is set to 01b to specify that the data virtual-address is compared with the contents of DR*n* only during a memory-write. Setting this field to 11b specifies that the comparison takes place during both memory reads and memory writes.
- LEN*n* is set to 00b, 01b, 11b, or 10b to specify an address-match range of one, two, four, or eight bytes, respectively. Long mode must be active to set LEN*n* to 10b.

When a #DB exception occurs due to a data breakpoint address in DR*n*, the corresponding B*n* field in DR6 is set to 1 to indicate that a breakpoint condition occurred. The breakpoint occurs after the data-access instruction is executed, which means that the original data is overwritten by the data-access instruction. If the debug handler needs to report the previous data value, it must save that value before setting the breakpoint.

Because the breakpoint occurs after the data-access instruction is executed, the address of the instruction following the data-access instruction is pushed onto the debug-handler stack. Repeated string instructions, however, can trigger a breakpoint before all iterations of the repeat loop have completed. When this happens, the address of the string instruction is pushed onto the stack during a #DB exception if the repeat loop is not

complete. A subsequent IRET from the #DB handler returns to the string instruction, causing the remaining iterations to be executed. Most implementations cannot report breakpoints exactly for repeated string instructions, but instead report the breakpoint on an iteration later than the iteration where the breakpoint occurred.

I/O Breakpoints. I/O breakpoints are set by loading a breakpoint-address register (DR*n*) with the I/O-port address to be trapped, and then setting the corresponding DR7 fields as follows:

- L*n* or G*n* is set to 1 to enable the breakpoint for either the local task or all tasks, respectively.
- R/W*n* is set to 10b to specify that the I/O-port address is compared with the contents of DR*n* only during execution of an I/O instruction. This encoding of R/W*n* is valid only when debug extensions are enabled (CR4.DE=1).
- LEN*n* is set to 00b, 01b, or 11b to specify the breakpoint occurs on a byte, word, or doubleword I/O operation, respectively.

The I/O-port address specified by the I/O instruction is zero extended by the processor to 64 bits before comparing it with the DR*n* registers.

When a #DB exception occurs due to an I/O breakpoint in DR*n*, the corresponding B*n* field in DR6 is set to 1 to indicate that a breakpoint condition occurred. The breakpoint occurs after the instruction is executed, which means that the original data is overwritten by the breakpoint instruction. If the debug handler needs to report the previous data value, it must save that value before setting the breakpoint.

Because the breakpoint occurs after the instruction is executed, the address of the instruction following the I/O instruction is pushed onto the debug-handler stack, in most cases. In the case of INS and OUTS instructions that use the repeat prefix, however, the breakpoint occurs after the first iteration of the repeat loop. When this happens, the I/O-instruction address can be pushed onto the stack during a #DB exception if the repeat loop is not complete. A subsequent return from the debug handler causes the next I/O iteration to be executed. If the breakpoint condition is still set, the #DB exception reoccurs after that iteration is complete.

Single Stepping. Single-step breakpoints are enabled by setting the rFLAGS.TF bit to 1. When single stepping is enabled, a #DB exception occurs after every instruction is executed until it is disabled by clearing rFLAGS.TF to 0. However, the instruction that sets the TF bit, and the instruction that follows it, is *not* single stepped.

When a #DB exception occurs due to single stepping, the processor clears rFLAGS.TF to 0 before entering the debug handler, so that the debug handler itself is not single stepped. The processor also sets DR6.BS to 1, which indicates that the #DB exception occurred as a result of single stepping. The rFLAGS image pushed onto the debug-handler stack has the TF bit set, and single stepping resumes when a subsequent IRET pops the stack image into the rFLAGS register.

Single-step breakpoints have a higher priority than external interrupts. If an external interrupt occurs during single stepping, control is transferred to the #DB handler first, causing the rFLAGS.TF bit to be cleared to 0. Next, before the first instruction in the debug handler is executed, the processor transfers control to the pending-interrupt handler. This allows external interrupts to be handled outside of single-step mode.

The INT*n*, INT3, and INTO instructions clear the rFLAGS.TF bit to 0 when they are executed. If a debugger is used to single-step software that contains these instructions, it must emulate them instead of executing them.

The single-step mechanism can also be set to single step only control transfers, rather than single step every instruction. See “Single Stepping Control Transfers” on page 400 for additional information.

Task-Switch Breakpoints. Breakpoints can be set in a task TSS to raise a #DB exception after a task switch. Software enables a task breakpoint by setting the T bit in the TSS to 1. When a task switch occurs into a task with the T bit set, the processor completes loading the new task state. Before the first instruction is executed, the #DB exception occurs, and the processor sets DR6.BT to 1, indicating that the #DB exception occurred as a result of task breakpoint.

The processor does not clear the T bit in the TSS to 0 when the #DB exception occurs. Software must explicitly clear this bit to disable the task breakpoint. Software should never set the T-bit

in the debug-handler TSS if a separate task is used for #DB exception handling, otherwise the processor loops on the debug handler.

General-Detect Condition. General-detect is a special debug-exception condition that occurs when software running at any privilege level attempts to access any of the DR*n* registers while DR7.GD is set to 1. When a #DB exception occurs due to the general-detect condition, the processor clears DR7.GD to 0 and sets DR6.BD to 1. Clearing DR7.GD to 0 allows the debug handler to access the DR*n* registers without causing infinite #DB exceptions.

A debugger enables general detection to prevent other software from accessing and interfering with the debug registers while they are in use by the debugger. The exception is taken before executing the MOV DR*n* instruction so that the DR*n* contents are not altered.

13.2.3 Breakpoint Instruction (INT3)

The INT3 instruction, or the INT*n* instruction with an operand of 3, can be used to set breakpoints that transfer control to the breakpoint-exception (#BP) handler rather than the debug-exception handler. When a debugger uses the breakpoint instructions to set breakpoints, it does so by replacing the first bytes of an instruction with the breakpoint instruction. The debugger replaces the breakpoint instructions with the original-instruction bytes to clear the breakpoint.

INT3 is a single-byte instruction while INT*n* with an operand of 3 is a two-byte instruction. The instructions have slightly different effects on the breakpoint exception-handler stack. See "#BP—Breakpoint Exception (Vector 3)" on page 250 for additional information on this exception.

13.2.4 Control-Transfer Breakpoint Features

A control transfers is accomplished by using one of following instructions:

- Jumps (JMP, J*cc*, JCXZ, JECXZ).
- LOOP*x*.
- Software interrupts (INT*n*, INT 3, INTO).
- CALL.
- RET, IRET.
- SYSCALL, SYSRET.
- SYSENTER, SYSEXIT.

Recording Control Transfers. Software enables control-transfer recording by setting DebugCtlMSR.LBR to 1. Before setting this bit, software can initialize the control-transfer recording MSRs (LastBranchToIP, LastBranchFromIP, LastExceptionToIP, LastExceptionFromIP) to a known value (such as all zeros) using the WRMSR instruction.

After the control-transfer recording mechanism is initialized, the processor updates the recording MSRs automatically when control transfers occur. Every branch instruction, interrupt, and exception causes the LastBranchToIP register to be loaded with the target rIP, and the LastBranchFromIP register to be loaded with the source rIP. Interrupts and exceptions cause the processor to copy the contents of LastBranchToIP into LastExceptionToIP, and LastBranchFromIP into LastExceptionFromIP, before updating LastBranchToIP and LastBranchFromIP.

The processor automatically disables control-transfer recording when a debug exception (#DB) occurs by clearing DebugCtlMSR.LBR to 0. The contents of the control-transfer recording MSRs are not altered by the processor when the #DB occurs. Before exiting the debug-exception handler, software can set DebugCtlMSR.LBR to 1 to re-enable the recording mechanism.

Debuggers can trace a control transfer backward from a bug to its source using the recording MSRs and the breakpoint-address registers. The debug handler does this by updating the breakpoint registers from the recording MSRs after a #DB exception occurs, and restarting the program. The program takes a #DB exception on the previous control transfer, and this process can be repeated. The debug handler cannot simply copy the contents of the recording MSR into the breakpoint-address register. The recording MSRs hold segment offsets, while the debug registers hold virtual (linear) addresses. The debug handler must calculate the virtual address by reading the code-segment selector (CS) from the interrupt-handler stack, then reading the segment-base address from the CS descriptor, and adding that base address to the offset in the recording MSR. The calculated virtual-address can then be used as a breakpoint address.

Single Stepping Control Transfers. Software can enable control-transfer single stepping by setting DebugCtlMSR.BTF to 1 and

rFLAGS.TF to 1. The processor automatically disables control-transfer single stepping when a debug exception (#DB) occurs by clearing DebugCtlMSR.BTF to 0. rFLAGS.TF is also cleared when a #DB exception occurs. Before exiting the debug-exception handler, software must set both DebugCtlMSR.BTF and rFLAGS.TF to 1 to restart single stepping.

When enabled, this single-step mechanism causes a #DB exception to occur on every branch instruction, interrupt, or exception. Debuggers can use this capability to perform a "coarse" single step across blocks of code (bound by control transfers), and then, as the problem search is narrowed, switch into a "fine" single-step mode on every instruction (DebugCtlMSR.BTF=0, rFLAGS.TF=1).

Debuggers can use both the single-step mechanism and recording mechanism to support full backward and forward tracing of control transfers.

13.3 Performance Optimization

The AMD64 architecture supports the performance-monitoring features introduced in earlier processor implementations. These features allow the selection of events to be monitored, and include a set of corresponding counter registers that track the frequency of monitored events. Software tools can use these features to identify performance bottlenecks, such as sections of code that have high cache-miss rates or frequently mis-predicted branches. This information can then be used as a guide for improving or eliminating performance problems through software optimizations or hardware-design improvements.

The performance-monitoring features include:

- A set of performance-counter registers that count the frequency or duration of specific processor events.
- A set of performance-event-select registers used to specify the events that are be tracked by the performance-counter registers.
- A time-stamp counter that can be used to count processor-clock cycles over time.

Implementations are not required to support the performance counters and the event-select registers, or the time-stamp

counter. The presence of these features can be determined by executing the CPUID instruction, and checking the returned feature bits to determine their availability. See “Processor Feature Identification” on page 75 for more information on using the CPUID instruction.

The registers used in support of performance monitoring are model-specific registers (MSRs). See “Performance-Monitoring MSRs” on page 443 for a listing of the performance-monitoring MSR numbers and their reset values.

Each feature, and its use, is described in the following sections.

13.3.1 Performance Counters

Performance counters are used to count specific processor events, such as data-cache misses, or the duration of events, such as the number of clocks it takes to return data from memory after a cache miss. During event counting, the processor increments the counter when it detects an occurrence of the event. During duration measurement, the processor counts the number of processor clocks it takes to complete an event. Each performance counter can be used to count one event, or measure the duration of one event, at a time.

Implementations of the AMD64 architecture can support any number of performance counters, PerfCtr*n*. The length, in bits, of the performance counters is also implementation dependent, but the maximum length supported is 64 bits. Figure 13-6 shows the format of the PerfCtr*n* register.

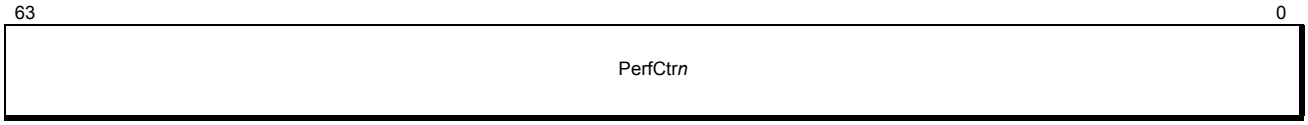


Figure 13-6. Performance Counter (PerfCtr*n*)

The PerfCtr*n* registers are model-specific registers that can be read using a special *read performance-monitoring counter* instruction, RDPMC. The RDPMC instruction loads the contents of the PerfCtr*n* register specified by the ECX register, into the EDX register and the EAX register. The high 32 bits are loaded into EDX, and the low 32 bits are loaded into EAX. RDPMC can be executed only at CPL=0, unless system software enables use of the instruction at all privilege levels. RDPMC can be enabled for use at all privilege levels by setting CR4.PCE (the *performance-monitor counter-enable* bit) to 1.

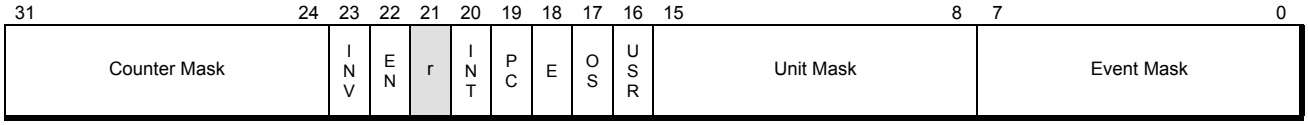
When CR4.PCE = 0 and CPL > 0, attempts to execute RDPMC result in a general-protection exception (#GP).

The performance counters can also be read and written by system software running at CPL=0 using the RDMSR and WRMSR instructions, respectively. Writing the performance counters can be useful if software wants to count a specific number of events, and then trigger an interrupt when that count is reached. An interrupt can be triggered when a performance counter overflows (see "Counter Overflow" on page 406 for additional information). Software should use the WRMSR instruction to load the count as a two's-complement negative number into the performance counter. This causes the counter to overflow after counting the appropriate number of times.

The performance counters are not guaranteed to produce identical measurements each time they are used to measure a particular instruction sequence, and they should not be used to take measurements of very small instruction sequences. The RDPMC instruction is not serializing, and it can be executed out-of-order with respect to other instructions around it. Even when bound by serializing instructions, the system environment at the time the instruction is executed can cause events to be counted before the counter value is loaded into EDX:EAX.

13.3.2 Performance Event-Select Registers

Performance event-select registers (PerfEvtSel*n*) are 32-bit registers used to specify the events counted by the performance counters, and to control other aspects of their operation. Each performance counter supported by the implementation has a corresponding event-select register that controls its operation. Figure 13-7 on page 404 shows the format of the PerfEvtSel*n* register.



Bits	Mnemonic	Description	R/W
31-24		Counter Mask	R/W
23	INV	Invert Mask	R/W
22	EN	Counter Enable	R/W
21	reserved		
20	INT	Interrupt Enable	R/W
19	PC	Pin Control	R/W
18	E	Edge Detect	R/W
17	OS	Operating-System Mode	R/W
16	USR	User Mode	R/W
15-8		Unit Mask	R/W
7-0		Event Mask	R/W

Figure 13-7. Performance Event-Select Register (PerfEvtSel*n*)

The fields within the PerfEvtSel*n* register are:

- *Event Mask*—Bits 7–0, read/write. This field specifies both the event or event duration to be counted by the corresponding PerfCtr*n* register. The events that can be counted are implementation dependent. For more information, refer to the BIOS writer's guide for the implementation.
- *Unit Mask*—Bits 15–8, read/write. This field can be used to specify a particular processor unit to be monitored, if the event counted can be produced by multiple units within the processor. Implementations can also use this field to further specify or qualify a monitored event.
- *Operating-System Mode (OS) and User Mode (USR)*—Bits 17–16 (respectively), read/write. Software uses these bits to control the privilege level at which event counting is performed according to Table 13-3.

Table 13-3. Operating-System Mode and User Mode Bits

OS Mode (Bit 17)	USR Mode (Bit 16)	Event Counting
0	0	No counting.
0	1	Only at CPL > 0.
1	0	Only at CPL = 0.
1	1	At all privilege levels.

- *Edge Detect (E)*—Bit 18, read/write. Software sets this bit to 1 to count the number of edge transitions from the negated to asserted state. This feature is useful when coupled with event-duration monitoring, as it can be used to calculate the average time spent in an event. Clearing this bit to 0 disables edge detection.
- *Pin Control (PC)*—Bit 19, read/write. Software sets this bit to 1 to cause the external PM*i* pins on the processor to toggle when the counter overflows. When this bit is cleared to 0, the processor toggles the PM*i* pins each time it increments the performance counter.
- *Interrupt Enable (INT)*—Bit 20, read/write. Software sets this bit to 1 to enable an interrupt to occur when the performance counter overflows (see "Counter Overflow" on page 406 for additional information). Clearing this bit to 0 disables the triggering of the interrupt.
- *Counter Enable (EN)*—Bit 22, read/write. Software sets this bit to 1 to enable the PerfEvtSel*n* register, and counting in the corresponding PerfCtr*n* register. Clearing this bit to 0 disables the register pair.
- *Invert Mask (INV)*—Bit 23, read/write. Software sets this bit to 1 to invert the comparison *result* performed on the counter-mask field, so that a less-than or equal-to comparison can be performed. Clearing this bit to 0 leaves the comparison result alone, so that a greater-than or equal-to comparison is reported.
- *Counter Mask*—Bits 31–24, read/write. This field is used to set a threshold for counting multiple events that can occur in a single clock. If the number of events occurring in the single clock is greater than or equal to this field, the corresponding PerfCtr*n* register is incremented. PerfCtr*n* is not incremented if the number of events is less than the count mask.

  The INV bit, when set, causes the PerfCtr*n* register to be incremented when the comparison is less than or equal to the count mask. In this case, PerfCtr*n* is not incremented if the number of events is greater than the count mask.

The performance event-select registers can be read and written only by system software running at CPL=0 using the RDMSR and WRMSR instructions, respectively. Any attempt to read or write these registers at CPL>0 causes a general-protection exception to occur.

13.3.3 Using Performance Counters

Starting and Stopping. Performance counting in a PerfCtr*n* register is initiated by setting the corresponding PerfEvtSel*n*.EN bit to 1. Counting is stopped by clearing PerfEvtSel*n*.EN to 0. Software must initialize the remaining PerfEvtSel*n* fields with the appropriate setup information before or at the same time EN is set. Counting begins when the WRMSR instruction that sets PerfEvtSel*n*.EN to 1 completes execution. Counting stops when the WRMSR instruction that clears PerfEvtSel*n*.EN to 0 completes execution.

Counter Overflow. Some processor implementations support an interrupt-on-overflow capability that allows an interrupt to occur when one of the PerfCtr*n* registers overflows. The source and type of interrupt is implementation dependent. Some implementations cause a debug interrupt to occur, while others make use of the local APIC to specify the interrupt vector and trigger the interrupt when an overflow occurs. Software controls the triggering of an interrupt by setting or clearing the PerfEvtSel*n*.INT bit.

If system software makes use of the interrupt-on-overflow capability, an interrupt handler must be provided that can record information relevant to the counter overflow. Before returning from the interrupt handler, the performance counter can be re-initialized to its previous state so that another interrupt occurs when the appropriate number of events are counted.

13.3.4 Time-Stamp Counter

The time-stamp counter (TSC) is used to count processor-clock cycles. The TSC is cleared to 0 after a processor reset. After a reset, the TSC is incremented by one for every processor clock cycle. Each time the TSC is read, it returns a monotonically-larger value than the previous value read from the TSC. When the TSC contains all ones, it wraps to zero. The TSC in a 1-GHz processor counts for almost 600 years before it wraps. Figure 13-8 shows the format of the 64-bit time-stamp counter (TSC).

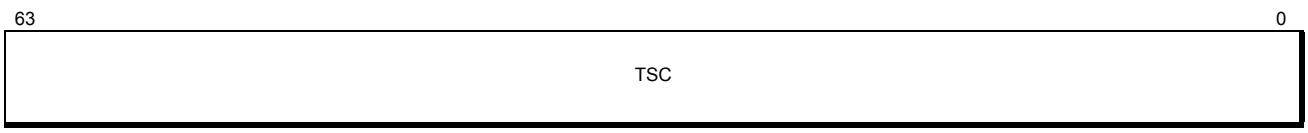


Figure 13-8. Time-Stamp Counter (TSC)

The TSC is a model-specific register that can also be read using a special *read time-stamp counter* instruction, RDTSC. RDTSC loads the contents of the TSC into the EDX register and the EAX register. The high 32 bits are loaded into EDX, and the low 32 bits are loaded into EAX. RDTSC can be executed at any privilege level and from any processor mode. However, system software can disable the RDTSC instruction for programs that run at CPL > 0 by setting CR4.TSD (the *time-stamp disable* bit) to 1. When CR4.TSD = 1 and CPL > 0, attempts to execute RDSTC result in a general-protection exception (#GP).

Some implementations allow the TSC register to be read and written using the RDMSR and WRMSR instructions, respectively. Support of this capability, however, is not required by the architecture, and software should avoid using these instructions to access the TSC. The programmer should use the CPUID instruction to determine whether these features are supported. If bit 4 returned by CPUID standard function 1 is set, then the processor supports TSC, the RDTSC instruction and CR4.TSD.

The TSC register can be used by performance-analysis applications, along with the performance-monitoring registers, to help determine the relative frequency of an event or its duration. Software can also use the TSC to time software routines to help identify candidates for optimization. In general, the TSC should not be used to take very short time measurements, because the resulting measurement is not guaranteed to be identical each time it is made. The RDTSC instruction is not serializing, and it can be executed out-of-order with respect to other instructions around it. Even when bound by serializing instructions, the system environment at the time the instruction is executed can cause additional cycles to be counted before the TSC value is loaded into EDX:EAX.

14 Processor Initialization and Long-Mode Activation

This chapter describes the hardware actions taken following a processor reset and the steps that must be taken to initialize processor resources and activate long mode. In some cases the actions required are implementation-specific with references made to the appropriate implementation-specific documentation.

14.1 Reset and Initialization

System logic initializes the processor in one of two ways. One method is to assert an external reset signal (typically designated RESET#). The other method, referred to here as INIT, is performed using implementation-dependent external signalling. Both initialization techniques place the processor in real mode and initialize processor resources to a known, consistent state from which software can begin execution. The differences between the two methods are:

- *RESET#*—This method provides an optional *built-in self test* (BIST) that can be performed as part of the RESET# process.
- *INIT*—This method does not modify the following state:
  - Memory hierarchy, including internal and external caches.
  - 128-bit media, 64-bit media, or x87 resources.
  - Memory-type range registers (MTRRs).
  - Machine-check registers.

  Some, but not all, model-specific registers (MSRs) are modified by an INIT.

  BIST cannot be performed as part of the INIT process.

The processor always performs RESET# after it is powered up, but RESET# can be performed at any time. An INIT can be performed at any time after the processor is powered up.

14.1.1 Built-In Self Test (BIST)

An optional built-in self-test can be performed after RESET# is asserted. The mechanism for triggering the BIST is implementation-specific, and can be found in the hardware documentation for the implementation. The number of processor cycles BIST can consume before completing is also

implementation-specific but typically consumes several million cycles.

BIST can be used by system implementations to assist in verifying system integrity, thereby improving system reliability, availability, and serviceability. The internal BIST hardware generally tests all internal array structures for errors. These structures can include (but are not limited to):

- All internal caches, including the tag arrays as well as the data arrays.
- All TLBs.
- Internal ROMs, such as the microcode ROM and floating-point constant ROM.
- Branch-prediction structures.

EAX is loaded with zero if BIST completes without detecting errors. If any hardware faults are detected during BIST, a non-zero value is loaded into EAX.

14.1.2 Clock Multiplier Selection

The internal processor clock runs at some multiple of the system clock. The processor-to-system clock multiple does not have to be fixed by a processor implementation but instead can be programmable through hardware or software, or some combination of the two. For information on selecting the processor-clock multiplier, refer to the BIOS writer's guide for the implementation.

14.1.3 Processor Initialization State

Table 14-1 shows the initial processor state following either RESET# or INIT. Except as indicated, processor resources generally are set to the same value after either RESET# or INIT.

Table 14-1. Initial Processor State

Processor Resource	Reset (RESET#) Value	Initialization (INIT) Value
CR0	0000_0000_6000_0010h	CD and NW are unchanged Bit 4 (reserved) = 1 All others = 0
CR2, CR3, CR4	0	
CR8	0	Not modified
RFLAGS	0000_0000_0000_0002h	

Table 14-1. Initial Processor State (continued)

Processor Resource	Reset (RESET#) Value	Initialization (INIT) Value
EFER	0	
RIP	0000_0000_0000_FFF0h	
CS	Selector = F000h Base = 0000_0000_FFFF_0000h Limit = FFFFh Attributes = See Table 14-2	
DS, ES, FS, GS, SS	Selector = 0000h Base = 0 Limit = FFFFh Attributes = See Table 14-2	
GDTR, IDTR	Base = 0 Limit = FFFFh	
LDTR, TR	Selector = 0000h Base = 0 Limit = FFFFh Attributes = See Table 14-2	
RAX	0 (non-zero if BIST is run and fails)	0
RDX	Family/Model/Stepping, including extended family and extended model–see "Processor Implementation Information" on page 414	
RBX, RCX, RBP, RSP, RDI, RSI, R8, R9, R10, R11, R12, R13, R14, R15	0	
x87 Floating-Point State	FPR0–FPR7 = 0 Control Word = 0040h Status Word = 0000h Tag Word = 5555h Instruction CS = 0000h Instruction Offset = 0 x87 Instruction Opcode = 0 Data-Operand DS = 0000h Data-Operand Offset = 0	Not modified
64-Bit Media State	MMX0–MMX7 = 0	Not modified

Table 14-1. Initial Processor State (continued)

Processor Resource	Reset (RESET#) Value	Initialization (INIT) Value
128-Bit Media State	XMM0–XMM15 = 0 MXCSR = 1F80h	Not modified
Memory-Type Range Registers	See "Memory-Typing MSRs" on page 438	Not modified
Machine-Check Registers	See "Machine-Check MSRs" on page 441	Not modified
DR0, DR1, DR2, DR3	0	
DR6	0000_0000_FFFF_0FF0h	
DR7	0000_0000_0000_0400h	
Time-Stamp Counter	0	Not modified
Performance-Monitor Resources	See "Performance-Monitoring MSRs" on page 443	Not modified
Other Model-Specific Registers	See "MSR Cross-Reference" on page 433	Not modified
Instruction and Data Caches	Invalidated	Not modified
Instruction and Data TLBs		
APIC	Enabled	Not modified
SMRAM Base Address (SMBASE)	0003_0000h	Not modified

Table 14-2 on page 413 shows the initial state of the segment-register attributes (located in the hidden portion of the segment registers) following either RESET# or INIT.

Table 14-2. Initial State of Segment-Register Attributes

Attribute		Value (Binary)	Description
G		0	Byte Granularity
D/B		0	16-Bit Segment
L (CS Only)		0	Legacy-Mode Segment
P		1	Segment is Present
DPL		00	Privilege-Level 0
S and Type	Code Segment	S = 1 Type = 1010	Executable/Readable Code Segment
	Data Segment	S = 1 Type = 0010	Read/Write Data Segment
	LDTR	S = 0 Type = 0010	LDT
	TR	S = 0 Type = 0011	Busy 16-Bit TSS

14.1.4 Multiple Processor Initialization

Following reset in multiprocessor configurations, the processors use a multiple-processor initialization protocol to negotiate which processor becomes the *bootstrap* processor. This bootstrap processor then executes the system initialization code while the remaining processors wait for software initialization to complete. For further information, see the documentation for particular implementations of the architecture.

14.1.5 Fetching the First Instruction

After a RESET# or INIT, the processor is operating in 16-bit real mode. Normally within real mode, the code-segment base-address is formed by shifting the CS-selector value left four bits. The base address is then added to the value in EIP to form the physical address into memory. As a result, the processor can only address the first 1 Mbyte of memory when in real mode.

However, immediately following RESET# or INIT, the CS-selector register is loaded with F000h, but the CS base-address is *not* formed by left-shifting the selector. Instead, the CS base-address is initialized to FFFF_0000h. EIP is initialized to

FFF0h. Therefore, the first instruction fetched from memory is located at physical-address FFFF_FFF0h (FFFF_0000h + 0000_FFF0h).

The CS base-address remains at this initial value until the CS-selector register is loaded by software. This can occur as a result of executing a far jump instruction or call instruction, for example. When CS is loaded by software, the new base-address value is established as defined for real mode (by left shifting the selector value four bits).

14.2 Hardware Configuration

14.2.1 Processor Implementation Information

Software can read processor-identification information from the EDX register immediately following RESET# or INIT. This information can be used to initialize software to perform processor-specific functions. The information stored in EDX is defined as follows:

- *Stepping ID (bits 3–0)*—This field identifies the processor-revision level.
- *Extended Model (bits 19–16) and Model (bits 7–4)*—These fields combine to differentiate processor models within a instruction family. For example, two processors may share the same microarchitecture but differ in their feature set. Such processors are considered different models within the same instruction family. This is a split field, comprising an extended-model portion in bits 19–16 with a legacy portion in bits 7–4
- *Extended Family (bits 27–20) and Family (bits 11–8)*—These fields combine to differentiate processors by their microarchitecture.

The CPUID instruction can be used to obtain the same information. This is done by executing CPUID with either standard-function code 1 or extended-function code 8000_0001h. Additional information about the processor and the features supported can be gathered using CPUID with other feature codes. See “Processor Feature Identification” on page 75 for additional information.

14.2.2 Enabling Internal Caches

Following a RESET# (but not an INIT), all instruction and data caches are disabled, and their contents are invalidated (the MOESI state is set to the invalid state). Software can enable

these caches by clearing the cache-disable bit (CR0.CD) to zero (RESET# sets this bit to 1). Software can further refine caching based on individual pages and memory regions. Refer to "Cache Control Mechanisms" on page 210 for more information on cache control.

Memory-Type Range Registers (MTRRs). Following a RESET# (but not an INIT), the MTRRdefType register is cleared to 0, which disables the MTRR mechanism. The variable-range and fixed-range MTRR registers are not initialized and are therefore in an undefined state. Before enabling the MTRR mechanism, the initialization software (usually BIOS) must load these registers with a known value to prevent unexpected results. Clearing these registers, for example, sets memory to the uncacheable (UC) type.

14.2.3 Initializing Media and x87 Processor State

Some resources used by x87 floating-point instructions and 128-bit media instructions must be initialized by software before being used. Initialization software can use the CPUID instruction to determine whether the processor supports these instructions, and then initialize their resources as appropriate.

x87 Floating-Point State Initialization. Table 14-3 on page 416 shows the differences between the initial x87 floating-point state following a RESET# and the state established by the FINIT/FNINIT instruction. An INIT does not modify the x87 floating-point state. The initialization software can execute an FINIT or FNINIT instruction to prepare the x87 floating-point unit for use by application software. The FINIT and FNINIT instructions have no effect on the 64-bit media state.

Table 14-3. x87 Floating-Point State Initialization

x87 Floating-Point Resource	RESET#	FINIT/FNINIT Instructions
FPR0–FPR7	0	Not modified
Control Word	0040h • Round to nearest • Single precision • Unmask all exceptions	037Fh • Round to nearest • Extended precision • Mask all exceptions
Status Word	0000h	
Tag Word	5555h (FPR*n* contain zero)	FFFFh (FPR*n* are empty)
Instruction CS	0000h	
Instruction Offset	0	
x87 Instruction Opcode	0	
Data-Operand DS	0000h	
Data-Operand Offset	0	

Initialization software should also load the MP, EM, and NE bits in the CR0 register as appropriate for the operating system. The recommended settings for implementations of the AMD64 architecture are:

- *MP=1*—Setting MP to 1 causes a device-not-available exception (#NM) to occur when the FWAIT/WAIT instruction is executed and the task-switched bit (CR0.TS) is set to 1. This supports operating systems that perform lazy context-switching of x87 floating-point state.
- *EM=0*—Clearing EM to 0 allows the x87 floating-point unit to execute instructions rather than causing a #NM exception (CR0.EM=1). System software sets EM to 1 only when software emulation of x87 instructions is desired.
- *NE=1*—Setting NE to 1 causes x87 floating-point exceptions to be handled by the floating-point exception-pending exception (#MF) handler. Clearing this bit causes the processor to externally indicate the exception occurred, and an external device can then cause an external interrupt to occur in response.

Refer to "CR0 Register" on page 53 for additional information on these control bits.

64-Bit Media State Initialization. There are no special requirements placed on software to initialize the processor state used by 64-bit media instructions. This state is initialized completely by the processor following a RESET#. System software should leave CR0.EM cleared to 0 to allow execution of the 64-bit media instructions. If CR0.EM is set to 1, attempted execution of the 64-bit media instructions causes an invalid-opcode exception (#UD).

The 64-bit media state is not modified by an INIT.

128-Bit Media State Initialization. BIOS or system software must also prepare the processor to allow execution of 128-bit media instructions. The required preparations include:

- Leaving CR0.EM cleared to 0 to allow execution of the 128-bit media instructions. If CR0.EM is set to 1, attempted execution of the 128-bit media instructions causes an invalid-opcode exception (#UD).
- Enabling the 128-bit media instructions by setting CR4.OSFXSR to 1. Software cannot execute the 128-bit media instructions unless this bit is set. Setting this bit also indicates that system software uses the FXSAVE and FXRSTOR instructions to save and restore, respectively, the 128-bit media state. These instructions also save and restore the 64-bit media state and x87 floating-point state.
- Indicating that system software uses the SIMD floating-point exception (#XF) for handling 128-bit media floating-point exceptions. This is done by setting CR4.OSXMMEXCPT to 1.
- Setting (optionally) the MXCSR mask bits to mask or unmask 128-bit media floating-point exceptions as desired. Because this register can be read and written by application software, it is not absolutely necessary for system software to initialize it.

Refer to "CR4 Register" on page 58 for additional information on these CR4 control bits.

14.2.4 Model-Specific Initialization

Implementations of the AMD64 architecture can contain model-specific features and registers that are not initialized by the processor and therefore require system-software

initialization. System software must use the CPUID instruction to determine which features are supported. Model-specific features are generally configured using model-specific registers (MSRs), which can be read and written using the RDMSR and WRMSR instructions, respectively.

Some of the model-specific features are pervasive across many processor implementations of the AMD64 architecture and are therefore described within this volume. These include:

- System-call extensions, which must be enabled in the EFER register before using the SYSCALL and SYSRET instructions. See "System-Call Extension (SCE) Bit" on page 68 for information on enabling these instructions.
- Memory-typing MSRs. See "Memory-Type Range Registers (MTRRs)" on page 415 for information on initializing and using these registers.
- The machine-check mechanism. See "Initializing the Machine-Check Mechanism" on page 313 for information on enabling and using this capability.
- Extensions to the debug mechanism. See "Software-Debug Resources" on page 382 for information on initializing and using these extensions.
- The performance-monitoring resources. See "Performance Optimization" on page 401 for information on initializing and using these resources.

Initialization of other model-specific features used by the page-translation mechanism and long mode are described throughout the remainder of this section.

Some model-specific features are not pervasive across processor implementations and are therefore not described in this volume. For more information on these features and their initialization requirements, refer to the BIOS writer's guide for the implementation.

14.3 Initializing Real Mode

A basic real-mode (real-address-mode) operating environment must be initialized so that system software can initialize the protected-mode operating environment. This real-mode environment must include:

- A real-mode IDT for vectoring interrupts and exceptions to the appropriate handlers while in real mode. The IDT base-address value in the IDTR initialized by the processor can be used, or system software can relocate the IDT by loading a new base-address into the IDTR.
- The real-mode interrupt and exception handlers. These must be loaded before enabling external interrupts.

  Because the processor can always accept a non-maskable interrupt (NMI), it is possible an NMI can occur before initializing the IDT or the NMI handler. System hardware must provide a mechanism for disabling NMIs to allow time for the IDT and NMI handler to be properly initialized. Alternatively, the IDT and NMI handler can be stored in non-volatile memory that is referenced by the initial values loaded into the IDTR.

  Maskable interrupts can be enabled by setting EFLAGS.IF after the real-mode IDT and interrupt handlers are initialized.
- A valid stack pointer (SS:SP) to be used by the interrupt mechanism should interrupts or exceptions occur. The values of SS:SP initialized by the processor can be used.
- One or more data-segment selectors for storing the protected-mode data structures that are created in real mode.

Once the real-mode environment is established, software can begin initializing the protected-mode environment.

14.4 Initializing Protected Mode

Protected mode must be entered before activating long mode. A minimal protected-mode environment must be established to allow long-mode initialization to take place. This environment must include the following:

- A protected-mode IDT for vectoring interrupts and exceptions to the appropriate handlers while in protected mode.
- The protected-mode interrupt and exception handlers referenced by the IDT. Gate descriptors for each handler must be loaded in the IDT.

- A GDT which contains:
  - A code descriptor for the code segment that is executed in protected mode.
  - A read/write data segment that can be used as a protected-mode stack. This stack can be used by the interrupt mechanism if interrupts or exceptions occur.

Software can optionally load the GDT with one or more data segment descriptors, a TSS descriptor, and an LDT descriptor for use by long-mode initialization software.

After the protected-mode data structures are initialized, system software must load the IDTR and GDTR (and optionally, the LDTR and TR) with pointers to those data structures. Once these registers are initialized, protected mode can be enabled by setting CR0.PE to 1.

If legacy paging is used during the long-mode initialization process, the page-translation tables must be initialized before enabling paging. At a minimum, one page directory and one page table are required to support page translation. The CR3 register must be loaded with the starting physical address of the highest-level table supported in the page-translation hierarchy. After these structures are initialized and protected mode is enabled, paging can be enabled by setting CR0.PG to 1.

14.5 Initializing Long Mode

From protected mode, system software can initialize the data structures required by long mode and store them anywhere in the first 4 Gbytes of physical memory. These data structures can be relocated above 4 Gbytes once long mode is activated. The data structures required by long mode include the following:

- An IDT with 64-bit interrupt-gate descriptors. Long-mode interrupts are always taken in 64-bit mode, and the 64-bit gate descriptors are used to transfer control to interrupt handlers running in 64-bit mode. See “Long-Mode Interrupt Control Transfers” on page 285 for more information.
- The 64-bit mode interrupt and exception handlers to be used in 64-bit mode. Gate descriptors for each handler must be loaded in the 64-bit IDT.

- A GDT containing segment descriptors for software running in 64-bit mode and compatibility mode, including:
  - Any LDT descriptors required by the operating system or application software.
  - A TSS descriptor for the single 64-bit TSS required by long mode.
  - Code descriptors for the code segments that are executed in long mode. The code-segment descriptors are used to specify whether the processor is operating in 64-bit mode or compatibility mode. See "Code-Segment Descriptors" on page 106, "Long (L) Attribute Bit" on page 107, and "CS Register" on page 85 for more information.
  - Data-segment descriptors for software running in compatibility mode. The DS, ES, and SS segments are ignored in 64-bit mode. See "Data-Segment Descriptors" on page 108 for more information.
  - FS and GS data-segment descriptors for 64-bit mode, if required by the operating system. If these segments are used in 64-bit mode, system software can also initialize the full 64-bit base addresses using the WRMSR instruction. See "FS and GS Registers in 64-Bit Mode" on page 86 for more information.

  The existing protected-mode GDT can be used to hold the long-mode descriptors described above.
- A single 64-bit TSS for holding the privilege-level 0, 1, and 2 stack pointers, the interrupt-stack-table pointers, and the I/O-redirection-bitmap base address (if required). This is the only TSS required, because hardware task-switching is not supported in long mode. See "64-Bit Task State Segment" on page 368 for more information.
- The 4-level page-translation tables required by long mode. Long mode also requires the use of physical-address extensions (PAE) to support physical-address sizes greater than 32 bits. See "Long-Mode Page Translation" on page 158 for more information.

If paging is enabled during the initialization process, it *must* be disabled before enabling long mode. After the long-mode data structures are initialized, and paging is disabled, software can enable and activate long mode.

14.6 Enabling and Activating Long Mode

Long mode is *enabled* by setting the long-mode enable control bit (EFER.LME) to 1. However, long mode is not *activated* until software also enables paging. When software enables paging while long mode is enabled, the processor activates long mode, which the processor indicates by setting the long-mode-active status bit (EFER.LMA) to 1. The processor behaves as a 32-bit x86 processor in all respects until long mode is activated, even if long mode is enabled. None of the new 64-bit data sizes, addressing, or system aspects available in long mode can be used until EFER.LMA=1.

Table 14-4 shows the control-bit settings for enabling and activating the various operating modes of the AMD64 architecture. The default address and data sizes are shown for each mode. For the methods of overriding these default address and data sizes, see "Instruction Prefixes" in Volume 3.

Table 14-4. Processor Operating Modes

Mode		Encoding			Default Address Size (bits)[2]	Default Data Size (bits)[2]
		EFER.LMA[1]	CS.L	CS.D		
Long Mode	64-Bit Mode	1	1	0	64	32
	Compatibility Mode		0	1	32	32
				0	16	16
Legacy Mode		0	x	1	32	32
				0	16	16

Note:

1. *EFER.LMA is set by the processor when software sets EFER.LME and CR0.PG according to the sequence described in "Activating Long Mode" on page 423.*
2. *See "Instruction Prefixes" in Volume 1 for overrides to default sizes.*

Long mode uses two code-segment-descriptor bits, CS.L and CS.D, to control the operating submodes. If long mode is active, CS.L = 1, and CS.D = 0, the processor is running in 64-bit mode, as shown in Table 14-4. With this encoding (CS.L=1, CS.D=0),

default operand size is 32 bits and default address size is 64 bits. Using instruction prefixes, the default operand size can be overridden to 64 bits or 16 bits, and the default address size can be overridden to 32 bits.

The final encoding of CS.L and CS.D in long mode (CS.L=1, CS.D=1) is reserved for future use.

When long mode is active and CS.L is cleared to 0, the processor is in compatibility mode, as shown in Table 14-4. In compatibility mode, CS.D controls default operand and address sizes exactly as it does in the legacy x86 architecture. Setting CS.D to 1 specifies default operand and address sizes as 32 bits. Clearing CS.D to 0 specifies default operand and address sizes as 16 bits.

14.6.1 Activating Long Mode

Switching the processor to long mode requires several steps. In general, the sequence involves disabling paging (CR0.PG=0), enabling physical-address extensions (CR4.PAE=1), loading CR3, enabling long mode (EFER.LME=1), and finally enabling paging (CR0.PG=1).

Specifically, software must follow this sequence to activate long mode:

1. If starting from page-enabled protected mode, disable paging by clearing CR0.PG to 0. This requires that the MOV CR0 instruction used to disable paging be located in an identity-mapped page (virtual address equals physical address).
2. In any order:
   - Enable physical-address extensions by setting CR4.PAE to 1. Long mode requires the use of physical-address extensions (PAE) in order to support physical-address sizes greater than 32 bits. Physical-address extensions must be enabled before enabling paging.
   - Load CR3 with the physical base-address of the level-4 page-map-table (PML4). See “Long-Mode Page Translation” on page 158 for details on creating the 4-level page translation tables required by long mode.
   - Enable long mode by setting EFER.LME to 1.

3. Enable paging by setting CR0.PG to 1. This causes the processor to set the EFER.LMA bit to 1. The instruction following the MOV CR0 that enables paging must be a branch, and both the MOV CR0 and the following branch instruction must be located in an identity-mapped page.

14.6.2 Consistency Checks

The processor performs long-mode consistency checks whenever software attempts to modify any of the control bits directly involved in activating long mode (EFER.LME, CR0.PG, and CR4.PAE). A general-protection exception (#GP) occurs when a consistency check fails. Long-mode consistency checks ensure that the processor does not enter an undefined mode or state that results in unpredictable behavior.

Long-mode consistency checks cause a general-protection exception (#GP) to occur if:

- An attempt is made to enable or disable long mode while paging is enabled.
- Long mode is enabled, and an attempt is made to enable paging before enabling physical-address extensions (PAE).
- Long mode is enabled, and an attempt is made to enable paging while CS.L=1.
- Long mode is active and an attempt is made to disable physical-address extensions (PAE).

Table 14-5 summarizes the long-mode consistency checks made during control-bit transitions.

Table 14-5. Long-Mode Consistency Checks

Control Bit	Transition	Check
EFER.LME	0 → 1	If (CR0.PG=1) then #GP(0)
	1 → 0	If (CR0.PG=1) then #GP(0)
CR0.PG	0 → 1	If ((EFER.LME=1) & (CR4.PAE=0) then #GP(0) If ((EFER.LME=1) & (CS.L=1)) then #GP(0)
CR4.PAE	1 → 0	If (EFER.LMA=1) then #GP(0)

14.6.3 Updating System Descriptor Table References

Immediately after activating long mode, the system-descriptor-table registers (GDTR, LDTR, IDTR, TR) continue to reference legacy descriptor tables. The tables referenced by these descriptors all reside in the lower 4 Gbytes of virtual-address

space. After activating long mode, 64-bit operating-system software should use the LGDT, LLDT, LIDT, and LTR instructions to load the system descriptor-table registers with references to the 64-bit versions of the descriptor tables. See "Descriptor Tables" on page 87 for details on descriptor tables in long mode.

Long mode requires 64-bit interrupt-gate descriptors to be stored in the interrupt-descriptor table (IDT). Software must not allow exceptions or interrupts to occur between the time long mode is activated and the subsequent update of the interrupt-descriptor-table register (IDTR) that establishes a reference to the 64-bit IDT. This is because the IDTR continues to reference a 32-bit IDT immediately after long mode is activated. If an interrupt or exception occurred before updating the IDTR, a legacy 32-bit interrupt gate would be referenced and interpreted as a 64-bit interrupt gate, with unpredictable results.

External interrupts can be disabled using the CLI instruction. Non-maskable interrupts (NMI) and system-management interrupts (SMI) must be disabled using external hardware. See "Long-Mode Interrupt Control Transfers" on page 285 for more information on long mode interrupts.

14.6.4 Relocating Page-Translation Tables

The long-mode page-translation tables must be located in the first 4 Gbytes of physical-address space before activating long mode. This is necessary because the MOV CR3 instruction used to initialize the page-map level-4 base address must be executed in legacy mode before activating long mode. Because the MOV CR3 is executed in legacy mode, only the low 32 bits of the register are written, which limits the location of the page-map level-4 translation table to the low 4 Gbytes of memory. Software can relocate the page tables anywhere in physical memory, and re-initialize the CR3 register, after long mode is activated.

14.7 Leaving Long Mode

To return from long mode to legacy protected mode with paging enabled, software must deactivate and disable long mode using the following sequence:

1. Switch to compatibility mode and place the processor at the highest privilege level (CPL=0).

2. Deactivate long mode by clearing CR0.PG to 0. This causes the processor to clear the LMA bit to 0. The MOV CR0 instruction used to disable paging must be located in an identity-mapped page. Once paging is disabled, the processor behaves as a standard 32-bit x86 processor.
3. Load CR3 with the physical base-address of the legacy page tables.
4. Disable long mode by clearing EFER.LME to 0.
5. Enable legacy page-translation by setting CR0.PG to 1. The instruction following the MOV CR0 that enables paging must be a branch, and both the MOV CR0 and the following branch instruction must be located in an identity-mapped page.

14.8 Long-Mode Initialization Example

Following is sample code that outlines the steps required to place the processor in long mode.

```
mydata  segment para

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; This generic data-segment holds pseudo-descriptors used
; by the LGDT and LIDT instructions.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;
; Establish a temporary 32-bit GDT and IDT.
;

pGDT32  label   fword                   ; Used by LGDT.
                dw      gdt32_limit     ; GDT limit ...
                dd      gdt32_base      ; and 32-bit GDT base

pIDT32  label   fword                   ; Used by LIDT.
                dw      idt32_limit     ; IDT limit ...
                dd      idt32_base      ; and 32-bit IDT base


;
; Establish a 64-bit GDT and IDT (64-bit linear base-
; address)
;
```

```
pGDT64  label   tbyte                   ; Used by LGDT.
                dw      gdt64_limit     ; GDT limit ...
                dq      gdt64_base      ; and 64-bit GDT base

pIDT64  label   tbyte                   ; Used by LIDT.
                dw      idt64_limit     ; IDT limit ...
                dq      idt64_base      ; and 64-bit GDT base

mydata  ends              ; end of data segment


code16  segment para use16 ; 16-bit code segment

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; 16-bit code, real mode
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;
; Initialize DS to point to the data segment containing
; pGDT32 and PIDT32. Set up a real-mode stack pointer, SS:SP,
; in case of interrupts and exceptions.
;
   cli
   mov   ax, seg mydata
   mov   ds, ax
   mov   ax, seg mystack
   mov   ss, ax
   mov   sp, esp0


;
; Use CPUID to determine if the processor supports long mode.
;

   mov   eax, 80000000h ; Extended-function code 8000000h.
   cpuid                ; Is largest extended function
   cmp   eax, 80000000h ; any function > 80000000h?
   jbe   no_long_mode   ; If not, no long mode.
   mov   eax, 80000001h ; Extended-function code 8000001h.
   cpuid                ; Now EDX = extended-features flags.
   bt    edx, 29        ; Test if long mode is supported.
   jnc   no_long_mode   ; Exit if not supported.


;
; Load the 32-bit GDT before entering protected mode.
; This GDT must contain, at a minimum, the following
; descriptors:
```

```
;  1) a CPL=0 16-bit code descriptor for this code segment.
;  2) a CPL=0 32/64-bit code descriptor for the 64-bit code.
;  3) a CPL=0 read/write data segment, usable as a stack
;  (referenced by SS).
;
; Load the 32-bit IDT, in case any interrupts or exceptions
; occur after entering protected mode, but before enabling
; long mode).
;
; Initialize the GDTR and IDTR to point to the temporary
; 32-bit GDT and IDT, respectively.
;

   lgdt  ds:[pGDT32]
   lidt  ds:[pIDT32]


;
; Enable protected mode (CR0.PE=1).
;

   mov   eax, 000000011h
   mov   cr0, eax


;
; Execute a far jump to turn protected mode on.
; code16_sel must point to the previously-established 16-bit
; code descriptor located in the GDT (for the code currently
; being executed).
;

   db    0eah              ;Far jump...
   dw    offset now_in_prot;to offset...
   dw    code16_sel        ;in current code segment.


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; At this point we are in 16-bit protected mode, but long
; mode is still disabled.
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

now_in_prot:


;
; Set up the protected-mode stack pointer, SS:ESP.
; Stack_sel must point to the previously-established stack
; descriptor (read/write data segment), located in the GDT.
```

```
; Skip setting DS/ES/FS/GS, because we are jumping right to
; 64-bit code.
;

   mov   ax, stack_sel
   mov   ss, ax
   mov   esp, esp0


;
; Enable the 64-bit page-translation-table entries by
; setting CR4.PAE=1 (this is _required_ before activating
; long mode). Paging is not enabled until after long mode
; is enabled.
;

   mov   eax, cr4
   bts   eax, 5
   mov   cr4, eax


;
; Create the long-mode page tables, and initialize the
; 64-bit CR3 (page-table base address) to point to the base
; of the PML4 page table. The PML4 page table must be located
; below 4 Gbytes because only 32 bits of CR3 are loaded when
; the processor is not in 64-bit mode.
;

   mov   eax, pml4_base ; Pointer to PML4 table (<4GB).
   mov   cr3, eax       ; Initialize CR3 with PML4 base.


;
; Enable long mode (set EFER.LME=1).
;

   mov   ecx, 0c0000080h   ; EFER MSR number.
   rdmsr                   ; Read EFER.
   bts   eax, 8            ; Set LME=1.
   wrmsr                   ; Write EFER.


;
; Enable paging to activate long mode (set CR0.PG=1)
;

   mov   eax, cr0          ; Read CR0.
   bts   eax, 31           ; Set PE=1.
   mov   cr0, eax          ; Write CR0.
```

```
;
; At this point, we are in 16-bit compatibility mode
; ( LMA=1, CS.L=0, CS.D=0 ).
; Now, jump to the 64-bit code segment. The offset must be
; equal to the linear address of the 64-bit entry point,
; because 64-bit code is in an unsegmented address space.
; The selector points to the 32/64-bit code selector in the
; current GDT.
;

   db    066h
   db    0eah
   dd    start64_linear
   dw    code64_sel

code16ends     ; End of the 16-bit code segment


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;;
;;;    Start of 64-bit code
;;
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

code64  para use64

start64:       ; At this point, we're using 64-bit code


;
; Point the 64-bit RSP register to the stack's _linear_
; address. There is no need to set SS here, because the SS
; register is not used in 64-bit mode.
;

   mov   rsp, stack0_linear


;
; This LGDT is only needed if the long-mode GDT is to be
; located at a linear address above 4 Gbytes.  If the long
; mode GDT is located at a 32-bit linear address, putting
; 64-bit descriptors in the GDT pointed to by [pGDT32] is
; just fine. pGDT64_linear is the _linear_ address of the
; 10-byte GDT pseudo-descriptor.
;
; The new GDT should have a valid CPL0 64-bit code segment
; descriptor at the entry-point corresponding to the current
```

```
; CS selector. Alternatively, a far transfer to a valid CPLO
; 64-bit code segment descriptor in the new GDT must be done
; before enabling interrupts.
;

   lgdt  [pGDT64_linear]


;
; Load the 64-bit IDT. This is _required_, because the 64-bit
; IDT uses 64-bit interrupt descriptors, while the 32-bit
; IDT used 32-bit interrupt descriptors. pIDT64_linear is
; the _linear_ address of the 10-byte IDT pseudo-descriptor.
;

   lidt  [pIDT64_linear]


;
; Set the current TSS. tss_sel should point to a 64-bit TSS
; descriptor in the current GDT. The TSS is used for
; inner-level stack pointers and the IO bit-map.
;

   mov   ax, tss_sel
   ltr   ax


;
; Set the current LDT. ldt_sel should point to a 64-bit LDT
; descriptor in the current GDT.
;

   mov   ax, ldt_sel
   lldt  ax


;
; Using fs: and gs: prefixes on memory accesses still uses
; the 32-bit fs.base and gs.base.  Reload these 2 registers
; before using the fs: and gs: prefixes. FS and GS can be
; loaded from the GDT using a normal “mov fs,foo” type
; instructions, which loads a 32-bit base into FS or GS.
; Alternatively, use WRMSR to assign 64-bit base values to
; MSR_FS_base or MSR_GS_base.
;

   mov   ecx, MSR_FS_base
   mov   eax, FsbaseLow
   mov   edx, FsbaseHi
   wrmsr
```

```
;
; Reload CR3 if long-mode page tables are to be located above
; 4 Gbytes. Because the original CR3 load was done in 32-bit
; legacy mode, it could only load 32 bits into CR3. Thus, the
; current page tables are located in the lower 4 Gbytes of
; physical memory. This MOV to CR3 is only needed if the
; actual long-mode page tables should be located at a linear
; address above 4 Gbytes.
;

   mov   rax, final_pml4_base  ; Point to PML4
   mov   cr3, rax              ; Load 64-bit CR3


;
; Enable interrupts.
;

   sti                         ; Enabled INTR

   <insert 64-bit code here>
```

Appendix A MSR Cross-Reference

This appendix lists the MSRs that are defined in the AMD64 architecture. The AMD64 architecture supports some of the same MSRs as previous versions of the x86 architecture and implementations thereof. Where possible, the AMD64 architecture supports the same MSRs, for the same functions, as these previous architectures and implementations.

The first section lists the MSRs according to their MSR address, and it gives a cross reference for additional information. The remaining sections list the MSRs by their functional group. Those sections also give a brief description of the register and specify the register reset value.

Some MSRs are implementation-specific For information about these MSRs, see the documentation for specific implementations of the AMD64 architecture.

A.1 MSR Cross-Reference by MSR Address

Table A-1 lists the MSRs in the AMD64 architecture in order of MSR address.

Table A-1. MSRs of the AMD64 Architecture

MSR Address	MSR Name	Functional Group	Cross-Reference
0010h	TSC	Performance	"Time-Stamp Counter" on page 406
00FEh	MTRRcap	Memory Typing	"Identifying MTRR Features" on page 224
0174h	SYSENTER_CS	System Software	"SYSENTER and SYSEXIT MSRs" on page 182
0175h	SYSENTER_ESP		
0176h	SYSENTER_EIP		
0179h	MCG_CAP	Machine Check	"Machine-Check Global-Capabilities Register" on page 306
017Ah	MCG_STATUS		"Machine-Check Global-Status Register" on page 306
017Bh	MCG_CTL		"Machine-Check Global-Control Register" on page 307
01D9h	DebugCtlMSR	Software Debug	"Debug-Control MSR (DebugCtlMSR)" on page 389

Table A-1. MSRs of the AMD64 Architecture (continued)

MSR Address	MSR Name	Functional Group	Cross-Reference
01DBh	LastBranchFromIP	Software Debug	"Control-Transfer Recording MSRs" on page 390
01DCh	LastBranchToIP		
01DDh	LastExceptionFromIP		
01DEh	LastExceptionToIP		
0200h	MTRRphysBase0	Memory Typing	"Variable-Range MTRRs" on page 220
0201h	MTRRphysMask0		
0202h	MTRRphysBase1		
0203h	MTRRphysMask1		
0204h	MTRRphysBase2		
0205h	MTRRphysMask2		
0206h	MTRRphysBase3		
0207h	MTRRphysMask3		
0208h	MTRRphysBase4		
0209h	MTRRphysMask4		
020Ah	MTRRphysBase5		
020Bh	MTRRphysMask5		
020Ch	MTRRphysBase6		
020Dh	MTRRphysMask6		
020Eh	MTRRphysBase7		
020Fh	MTRRphysMask7		

Table A-1. MSRs of the AMD64 Architecture (continued)

MSR Address	MSR Name	Functional Group	Cross-Reference
0250h	MTRRfix64K_00000	Memory Typing	"Fixed-Range MTRRs" on page 218
0258h	MTRRfix16K_80000		
0259h	MTRRfix16K_A0000		
0268h	MTRRfix4K_C0000		
0269h	MTRRfix4K_C8000		
026Ah	MTRRfix4K_D0000		
026Bh	MTRRfix4K_D8000		
026Ch	MTRRfix4K_E0000		
026Dh	MTRRfix4K_E8000		
026Eh	MTRRfix4K_F0000		
026Fh	MTRRfix4K_F8000		
0277h	PAT	Memory Typing	"PAT Register" on page 228
02FFh	MTRRdefType		"Default-Range MTRRs" on page 223
0400h	MC0_CTL	Machine Check	See the documentation for particular implementations of the architecture.
0404h	MC1_CTL		
0408h	MC2_CTL		
040Ch	MC3_CTL		
0410h	MC4_CTL		
0401h	MC0_STATUS	Machine Check	"Machine-Check Status Registers" on page 309
0405h	MC1_STATUS		
0409h	MC2_STATUS		
040Dh	MC3_STATUS		
0411h	MC4_STATUS		

Table A-1. MSRs of the AMD64 Architecture (continued)

MSR Address	MSR Name	Functional Group	Cross-Reference
0402h	MC0_ADDR	Machine Check	"Machine-Check Address Registers" on page 312
0406h	MC1_ADDR		
040Ah	MC2_ADDR		
040Eh	MC3_ADDR		
0412h	MC4_ADDR		
0403h	MC0_MISC	Machine Check	"Machine-Check Miscellaneous-Error Information Registers" on page 312
0407h	MC1_MISC		
040Bh	MC2_MISC		
040Fh	MC3_MISC		
0413h	MC4_MISC		
C000_0080h	EFER	System Software	"Extended Feature Enable Register (EFER)" on page 67
C000_0081h	STAR	System Software	"SYSCALL and SYSRET MSRs" on page 181
C000_0082h	LSTAR		
C000_0083h	CSTAR		
C000_0084h	SF_MASK		
C000_0100h	FS.Base	System Software	"FS and GS Registers in 64-Bit Mode" on page 86
C000_0101h	GS.Base		
C000_0102h	KernelGSbase	System Software	"SWAPGS Instruction" on page 183
C001_0000h	PerfEvtSel0	Performance	"Performance Event-Select Registers" on page 403
C001_0001h	PerfEvtSel1		
C001_0002h	PerfEvtSel2		
C001_0003h	PerfEvtSel3		
C001_0004h	PerfCtr0	Performance	"Performance Counters" on page 402
C001_0005h	PerfCtr1		
C001_0006h	PerfCtr2		
C001_0007h	PerfCtr3		
C001_0010h	SYSCFG	Memory Typing	"System Configuration Register (SYSCFG)" on page 70

Table A-1. MSRs of the AMD64 Architecture (continued)

MSR Address	MSR Name	Functional Group	Cross-Reference
C001_0016h	IORRBase0	Memory Typing	"IORRs" on page 235
C001_0017h	IORRMask0		
C001_0018h	IORRBase1		
C001_0019h	IORRMask1		
C001_001Ah	TOP_MEM	Memory Typing	"Top of Memory" on page 237
C001_001Dh	TOP_MEM2		

A.2 System-Software MSRs

Table A-2 lists the MSRs defined for general use by system software in controlling long mode and in allowing fast control transfers between applications and the operating system.

Table A-2. System-Software MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
C000_0080h	EFER	Contains control bits that enable extended features supported by the processor, including long mode.	0000_0000_0000_0000h
C000_0081h	STAR	In legacy mode, used to specify the target address of a SYSCALL instruction, as well as the CS and SS selectors of the called and returned procedures.	undefined
C000_0082h	LSTAR	In 64-bit mode, used to specify the target RIP of a SYSCALL instruction.	undefined
C000_0083h	CSTAR	In compatibility mode, used to specify the target RIP of a SYSCALL instruction.	undefined
C000_0084h	SF_MASK	SYSCALL Flags Mask	undefined
C000_0100h	FS.Base	Contains the 64-bit base address in the hidden portion of the FS register (the base address from the FS descriptor).	undefined

Table A-2. System-Software MSR Cross-Reference (continued)

MSR Address	MSR Name	Description	Reset Value
C000_0101h	GS.Base	Contains the 64-bit base address in the hidden portion of the GS register (the base address from the GS descriptor).	undefined
C000_0102h	KernelGSbase	The SWAPGS instruction exchanges the value in KernelGSbase with the value in GS.base, providing a fast method for system software to load a pointer to system data-structures.	undefined
0174h	SYSENTER_CS	In legacy mode, used to specify the CS selector of the procedure called by SYSENTER.	undefined
0175h	SYSENTER_ESP	In legacy mode, used to specify the stack pointer for the procedure called by SYSENTER.	undefined
0176h	SYSENTER_EIP	In legacy mode, used to specify the EIP of the procedure called by SYSENTER.	undefined

A.3 Memory-Typing MSRs

Table A-3 on page 439 lists the MSRs used to control memory-typing and the page-attribute-table mechanism.

Table A-3. Memory-Typing MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
00FEh	MTRRcap	A *read-only* register containing information describing the level of MTRR support provided by the processor.	0000_0000_0000_0508h
0200h	MTRRphysBase0	Specifies the memory-range base address in physical-address space of a variable-range memory region. These registers also specify the memory type used for the memory region.	undefined
0202h	MTRRphysBase1		
0204h	MTRRphysBase2		
0206h	MTRRphysBase3		
0208h	MTRRphysBase4		
020Ah	MTRRphysBase5		
020Ch	MTRRphysBase6		
020Eh	MTRRphysBase7		
0201h	MTRRphysMask0	Specifies the size of a variable-range memory region.	Valid (bit 11) = 0 All Other Bits Undefined
0203h	MTRRphysMask1		
0205h	MTRRphysMask2		
0207h	MTRRphysMask3		
0209h	MTRRphysMask4		
020Bh	MTRRphysMask5		
020Dh	MTRRphysMask6		
020Fh	MTRRphysMask7		

Table A-3. Memory-Typing MSR Cross-Reference (continued)

MSR Address	MSR Name	Description	Reset Value
0250h	MTRRfix64K_00000	Fixed-range MTRRs used to characterize the first 1 Mbyte of physical memory. Each 64-bit register contains eight type fields for characterizing a total of eight memory ranges. • MTRRfix64K_n characterizes 64 Kbyte ranges. • MTRRfix16K_n characterizes 16 Kbyte ranges. • MTRRfix4K_n characterizes 4 Kbyte ranges.	undefined
0258h	MTRRfix16K_80000		
0259h	MTRRfix16K_A0000		
0268h	MTRRfix4K_C0000		
0269h	MTRRfix4K_C8000		
026Ah	MTRRfix4K_D0000		
026Bh	MTRRfix4K_D8000		
026Ch	MTRRfix4K_E0000		
026Dh	MTRRfix4K_E8000		
026Eh	MTRRfix4K_F0000		
026Fh	MTRRfix4K_F8000		
0277h	PAT	Used to extend the page-table entry format, allowing memory-type characterization on a physical-page basis.	0007_0406_0007_0406h
02FFh	MTRRdefType	Sets the default memory-type for physical addresses not within ranges established by fixed-range and variable-range MTRRs.	0000_0000_0000_0000h
C001_0010h	SYSCFG	Contains control bits for enabling and configuring system bus features.	0000_0000_0002_0601h
C001_0016h	IORRBase0	Specifies the memory-range base address in physical-address space of a variable-range I/O region.	undefined
C001_0018h	IORRBase1		
C001_0017h	IORRMask0	Specifies the size of a variable-range I/O region.	Valid (bit 11) = 0 All Other Bits Undefined
C001_0019h	IORRMask1		
C001_001Ah	TOP_MEM	Sets the boundary between system memory and memory-mapped I/O for addresses below 4 Gbytes.	0000_0000_0400_0000h
C001_001Dh	TOP_MEM2	Sets the boundary between system memory and memory-mapped I/O for addresses above 4 Gbytes.	undefined

A.4 Machine-Check MSRs

Table A-4 lists the MSRs used in support of the machine-check mechanism.

Table A-4. Machine-Check MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
0179h	MCG_CAP	A read-only register that specifies the machine-check mechanism capabilities supported by the processor.	0000_0000_0000_0104h
017Ah	MCG_STATUS	Provides basic information about the processor state immediately after the occurrence of a machine-check error.	undefined
017Bh	MCG_CTL	Controls global reporting of machine-check errors from various sources.	0000_0000_0000_0000h
0400h	MC0_CTL	Controls error reporting for the data-cache-unit register bank.	0000_0000_0000_0000h
0404h	MC1_CTL	Controls error reporting for the instruction-cache-unit register bank.	0000_0000_0000_0000h
0408h	MC2_CTL	Controls error reporting for the bus-unit register bank.	0000_0000_0000_0000h
040Ch	MC3_CTL	Controls error reporting for the load/store-unit register bank.	0000_0000_0000_0000h
0401h	MC0_STATUS	Status registers for each error-reporting register bank, used to report machine-check error information for the specified register bank.	undefined
0405h	MC1_STATUS		
0409h	MC2_STATUS		
040Dh	MC3_STATUS		
0402h	MC0_ADDR	Reports the instruction memory-address or data memory-address responsible for the machine-check error for the specified register bank.	undefined
0406h	MC1_ADDR		
040Ah	MC2_ADDR		
040Eh	MC3_ADDR		

Table A-4. Machine-Check MSR Cross-Reference (continued)

MSR Address	MSR Name	Description	Reset Value
0403h	MC0_MISC	Reports miscellaneous information about the machine-check error for the specified register bank.	undefined
0407h	MC1_MISC		
040Bh	MC2_MISC		
040Fh	MC3_MISC		

A.5 Software-Debug MSRs

Table A-5 lists the MSRs used in support of the software-debug architecture.

Table A-5. Software-Debug MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
01D9h	DebugCtlMSR	Provides debug controls for control-transfer recording and control-transfer single stepping, and external-breakpoint reporting and trace messages.	0000_0000_0000_0000h
01DBh	LastBranchFromIP	During control-transfer recording, this register is loaded with the segment offset of the control-transfer source.	undefined
01DCh	LastBranchToIP	During control-transfer recording, this register is loaded with the segment offset of the control-transfer target.	undefined
01DDh	LastExceptionFromIP	When an interrupt occurs during control-transfer recording, this register is loaded with LastBranchFromIP before LastBranchFromIP is updated.	undefined
01DEh	LastExceptionToIP	When an interrupt occurs during control-transfer recording, this register is loaded with LastBranchToIP before LastBranchToIP is updated.	undefined

A.6 Performance-Monitoring MSRs

Table A-6 lists the MSRs used in support of performance monitoring, including the time-stamp counter.

Table A-6. Performance-Monitoring MSR Cross-Reference

MSR Address	MSR Name	Description	Reset Value
0010h	TSC	Counts processor-clock cycles. It is incremented once for each processor-clock cycle.	0000_0000_0000_0000h
C001_0000h	PerfEvtSel0	For the corresponding performance counter, this register specifies the events counted, and controls other aspects of counter operation.	undefined
C001_0001h	PerfEvtSel1		
C001_0002h	PerfEvtSel2		
C001_0003h	PerfEvtSel3		
C001_0004h	PerfCtr0	Used to count specific processor events, or the duration of events, as specified by the corresponding PerfEvtSel*n* register.	undefined
C001_0005h	PerfCtr1		
C001_0006h	PerfCtr2		
C001_0007h	PerfCtr3		

Index

Symbols

#AC 261
#BP 250
#BR 251
#D 260, 263
#DB 248
#DE 248
#DF 252
#GP 257
#I 260, 263
#IA 260
#IS 260
#MC 262
#MF 260
#NM 252
#NP 255
#O 260, 263
#OF 250
#P 260, 264
#PF 259
#SS 256
#TS 254
#U 260, 264
#UD 251
#XF 263
#Z 260, 263

Numerics

128-bit media instructions
  enabling 338
  feature identification 338
  MXCSR 339
  saving state 344
  XMM registers 36, 339
16-bit mode xxvi
32-bit mode xxvi
64-bit media instructions
  causing #MF exception 342
  feature identification 338
  initializing 417
  MMX registers 341
  saving state 344
64-bit mode ..... xxvi, 15

A

A bit 99, 101, 168
abort 243
AC bit 66
accessed (A)
  code segment 99
  data segment 101
  page-translation tables 168
address-breakpoint registers (DR0-DR3) 383
addressing
  RIP-relative xxxii
address-size prefix 39
ADDRV bit 311
alignment check (rFLAGS.AC) 66, 261
alignment mask (CR0.AM) 55, 261
alignment-check exception (#AC) 55, 66, 261
AM bit 55
architecture differences 29
ARPL instruction 188
attributes 94
available to software (AVL)
  descriptor 97
  page-translation tables 169
AVL bit 97, 169

B

B3–B0 bits 385
base address 90, 93, 97, 150, 159, 166
BD bit 385
benign exception 252
biased exponent xxvi
BIST 409
bootstrap processor 413
BOUND instruction 251
bound-range exception (#BR) 251
branches 40
breakpoint 392
  determining cause 394
  on address match 382, 396
  on any instruction 382
  on I/O 397
  on instruction 395
  on task switch 382, 398
  setting address 392
  specifying address-match length 392
breakpoint exception (#BP) 250
BS bit 385
BT bit 385
BTF bit 390
built-in self test (BIST) 409

C

C bit 99
cache

control mechanisms 210
control precedence 212
enabling 414
index 209
invalidate 214
line 193
line writeback and invalidate 213
offset 209
organization 207
self-modifying code 209
set 208
tag 209
way 208
writeback and invalidate 214
cache disable (CD) bit 55, 210
cache disable (CD), memory type 201
cache-coherency protocol 197
losing coherency 199
CALL
See call gate and control transfer.
call gate 104, 127
count field 106
count field, long mode 114
descriptor, long mode 41
jump through 130
parameters 133
privilege checks 129
stack switch 132
stack switch, long mode 42, 134
canonical address form 5, 159
CD bit 55, 210
CD memory type 201
CLFLUSH instruction 213
CLI instruction 185
clock multiplier 410
CLTS instruction 185
code segment 33, 85, 98
64-bit mode 85
accessed (A) 99
conforming (C) 99
default-operand size (D) 100
ignored fields in 64-bit mode 107
long bit (L) 33, 107
long mode 106
readable (R) 99
type field 99
coherency, cache 194
commit xxvii
commit, instruction results 194
compatibility mode ..... xxvi, 15
conforming (C), code segment 99
consistency checks, long mode 424
contributory exception 253
control registers 36, 51
control transfer 122
See also call gate and interrupt.
call gate 127
direct 123
far, conforming code segment 126
far, nonconforming code segment 124
interrupt to higher privilege 276
interrupt to same privilege 275
parameters 133
stack switch 132
control-transfer recording MSRs 390
coprocessor-segment-overrun exception 254
count field 114, 306
counter mask field 405
CPL 118
definition 118
in call gate protection 129
in data segment protection 119
in interrupt to higher privilege 278
in page protection 173
in protecting conforming CS 126
in protecting nonconforming CS 124
in stack segment protection 121
privileged instructions 177
SYSCALL, SYSRET assumptions 181
CPUID instruction 67, 75, 184
CR0 53
alignment mask (AM) 55, 261
cache disable (CD) 55, 210
emulate coprocessor (EM) 54, 339
extension type (ET) 54
monitor coprocessor (MP) 53
not write-through (NW) 55, 211
numeric error (NE) 55, 261
paging enable (PG) 56, 147
protection enable (PE) 53, 78, 87
task switched (TS) 54, 185
write protect (WP) 55, 174
CR1 62
CR2 ..... 56, 259
CR3 ..... 32, 56, 57, 149, 159, 365
non-PAE paging 149
PAE paging 57, 149
PAE paging, long mode 159
page-level cache disable (PCD) ... 150, 160
page-level write-through (PWT) .. 150, 159
table-base address 150, 159
CR4 58

debugging extensions (DE) 59
machine-check enable (MCE) 60, 262
OS #XF support (OSXMMEXCPT) 263, 339
OS FXSAVE/FXRSTOR support (OSFXSR) 338
page-global enable (PGE) 61, 171
page-size extensions (PSE) 60, 147, 152
performance counter enable (PCE) 61, 186, 402
physical-address extensions (PAE) 60,147, 158
protected-mode virtual interrupts (PVI) 59
time-stamp disable (TSD) 59, 186, 407
virtual-8086 mode extensions (VME) ... 59, 294
CR5–CR7 62
CR8 ..... 62, 271
CR9–CR15 62
CS register 85
CSTAR register 181, 436, 437

D

D bit 100, 108, 168
D/B bit 97, 102
data prefetch, cache 213
data segment 34, 85, 100
64-bit mode 86
accessed (A) 101
default operand size (D) 102
expand down (E) 101
FS and GS 34, 86
ignored fields in 64-bit mode 108
long mode 108
privilege checks 119
type field 101
writable (W) 101
DAZ bit 354
DE bit 59
DE exception 260, 263
debug 26
See breakpoint and single-step.
debug exception (#DB) 65, 248, 394
debug registers 36, 383
address-breakpoint registers (DR0-DR3) 383
control-transfer recording MSRs 390
debug-control MSR (DebugCtlMSR) ... 389
debug-control register (DR7) 386
debug-status register (DR6) 384
reserved (DR4, DR5) 384
debug-control MSR (DebugCtlMSR) 389
debug-control register (DR7) 386
DebugCtlMSR register 389, 433, 442
debugging extensions (CR4.DE) 59
debug-status register (DR6) 384
DEC instruction 43
default operand size
B bit, stack segment 102
D bit, code segment 100
D bit, data segment 102, 138
D/B bit, descriptor 97
with expand down 139
denormalized-operand exception (DE) . 260, 263
denormals-are-zeros (DAZ) mode 353
descriptor 80, 95
available to software (AVL) 97
code segment 33
data segment 34
default operand size (D/B) 97
DPL 97, 118, 371
gate 34
granularity (G) 98
long mode 106
present (P) 97, 371
S field 97, 371
segment base 97
segment limit 96
system segment 34
TSS 360
type field 97, 371
descriptor table 80, 87
global-descriptor table (GDT) 82
interrupt-descriptor table (IDT) 46
local-descriptor table (LDT) 82
descriptor-table registers 33, 82
64-bit mode 114
GDTR 89
IDTR 95
LDTR 91
device-not-available exception (#NM) 53, 54, 252
differences (architectural) 29
direct referencing xxvii
dirty (D), page-translation tables 168
displacement 39
displacements xxvii
divide-by-zero-error exception (#DE) 248
double quadword xxvii
double-fault exception (#DF) 252
doubleword xxvii

DP field 353
DPL 118
  data segment, 64-bit mode 109
  definition 118
  in call gate protection 129
  in data segment protection 119
  in interrupt stack switch 277
  in interrupt to higher privilege 278
  in protecting conforming CS 126
  in protecting nonconforming CS 124
  in stack segment protection 121
  in stack switching 133
DPL field 97, 371
DR0-DR3 registers 383
DR4, DR5 registers 384
DR6 register 384
DR7 register 386
DS field 353
DS register 85, 86

E

E bit 101, 405
eAX–eSP register xxxiii
EFER 37, 67
  long mode active (LMA) 68, 422
  long mode enable (LME) 68, 422
  system-call extension (SCE) 68
EFER register 436, 437
effective address 2, 31
effective address size xxviii
effective operand size xxviii
EFLAGS
  See rFLAGS.
eFLAGS register xxxiv
EIP
  See rIP.
eIP register xxxiv
EIPV bit 307
element xxviii
EM bit 54, 416
emulate coprocessor (CR0.EM) 54, 339
EN bit 311, 405
endian byte-ordering xxxvi
environment 344
error code
  page fault 266
  selector 265
ES register 86
ESP
  See rSP.
ET bit 54
event handler, definition 241
event mask field 404
exception handler, definition 241
exceptions xxviii
  abort 243
  benign 252
  contributory 253
  definition of 241
  definition of vector 245
  differences in long mode 46
  error code, page fault 266
  error code, selector 265
  fault 243
  floating-point priorities 269
  imprecise 242
  maskable 128-bit media floating point 244
  maskable x87 floating point 244
  masking during stack switches 244
  overview 241
  precise 242
  priorities 267
  trap 243
  while in SMM 331
exclusive state, MOESI 197
expand down (E)
  data segment 101
  stack segment 101, 139
exponent xxvi
extended family field 414
extended model field 414
extension type (CR0.ET) 54

F

family field 414
far control transfer 122
far return 42, 136
fault 243
FCW register 342, 344, 352
feature identification 75
fence instructions 196
fill, cache-line 193
first instruction 413
flat segmentation 7, 10, 80
FLDENV, FSTENV instructions 349
floating-point exception pending (#MF) 260
  caused by 64-bit media instructions 342
floating-point exception priorities 269
flush xxviii
FOP register 353
FPR registers 342, 344
FS and GS 34, 86
FS register 86
FS.Base register 436, 437

FSAVE, FRSTOR instructions 344
FSW register 341, 342, 344, 352
FTW register 341, 342, 344, 352
FXSAVE, FXRSTOR instructions 45, 61, 349
  32-bit memory image 352
  64-bit memory image 351
  x87 tag word format 354

G

G bit 98, 169
G3–G0 bits 387
gate descriptors 34
  call gate 104
  DPL 118
  ignored fields in long mode 112
  illegal types in long mode 112
  interrupt gate 104
  long mode 111, 114
  redefined types in long mode 112
  target-segment offset 105
  target-segment selector 106
  task gate 104
  trap gate 104
GD bit 387
GDT 88
GDTR 89
GE bit 387
general detect fault 248, 399
general-protection exception (#GP) 257
general-purpose registers (GPRs) 35
global descriptor table (GDT) 82, 88
  base address, 64-bit mode 90
  first entry 88
  limit check, long mode 90
global descriptor-table register (GDTR) .. 89
  base address 90
  limit 90
  loading 187
  storing 187
global page (G), page-translation tables 169
global pages 61, 171
granularity (G), descriptor 98, 138
GS register 86
GS.Base register 436, 438

H

halt 188
HLT instruction 188

I

I/O privilege level field (rFLAGS.IOPL) .. 64
I/O, memory-mapped 232
I/O-permission bitmap
  in 32-bit TSS 366
  in 64-bit TSS 369
I/O-permission bitmap (IOPB) 367
ID bit 67
IDT 94
IDTR 95
IE exception 260, 263
IF bit 64, 299
IGN xxix
immediate operand 39
imprecise exceptions and interrupts 242
INC instruction 43
indirect xxix
inexact-result exception 260, 264
INIT 409
initialization 409
initialization (INIT) 409
  processor state 410
instructions (system-management) 177
INT bit 405
INT3 instruction 250, 399
integer bit 355
interrupt descriptor table (IDT) 94
  limit check, long mode 95
interrupt descriptor-table register (IDTR) 95
  loading 187
  storing 187
interrupt flag (rFLAGS.IF) 64, 185
interrupt gate 104, 285
  IST field 114
interrupt handler, definition 241
interrupt redirection 282, 297
interrupt-descriptor table (IDT)
  index 241, 274, 286
  protected mode 274
  real-address mode 272
interrupt-redirection bitmap 366
interrupts
  definition of external 241
  definition of software 241
  definition of vector 245
  differences in long mode 46
  external 264
  external maskable 243
  external nonmaskable 244
  external-interrupt priorities 270
  imprecise 242
  long mode summary 285
  overview 241
  precise 242

priorities 267
returning from 64-bit mode 292
returns 281
software 264
stack alignment, long mode 289
stack pointer push, long mode 287
stack switch, long mode 46, 290
to higher privilege 276
to same privilege 275
while in SMM 331
interrupt-stack table (IST)..... 47, 114, 290
in 64-bit TSS 369
interrupt-vector table..... 272
INTn instruction 264, 399
INTO instruction..... 250
INV bit 405
invalid arithmetic-operand exception 260
invalid state, MOESI 197
invalid-opcode exception (#UD)..... 44, 251
invalid-operation exception (IE) 260, 263
invalid-TSS exception (#TS) 254
INVD instruction 189, 214
INVLPG instruction..... 171, 189
IOPB..... 366, 367
IOPL field..... 64, 282
IOPL-sensitive instruction 294
IORRBasen registers 236, 437, 440
IORRMaskn registers..... 236, 440
IORRs, variable-range..... 235
IRET instruction
less privilege 281
long mode 47, 292
same privilege..... 281
IST field..... 114

J

J bit 355
jump
See call gate and control transfer.

K

KernelGSbase register 183, 436, 438

L

L bit..... 107
L1 data cache 193
L1 instruction cache 193
L2 cache..... 193
L3–L0 bits..... 387
LAR instruction 187
LastBranchFromIP register..... 391, 434, 442
LastBranchToIP register 391, 434, 442
LastExceptionFromIP register 392
LastExceptionToIP register 391
LastIntFromIP register..... 434, 442
LastIntToIP register..... 434, 442
LBR bit..... 389
LDT..... 90
LDTR..... 91
LDTselector field 366
LE bit..... 387
legacy mode ..... xxix, 16, 30
legacy x86 xxix
LEN3–LEN0 bits..... 388
LFENCE instruction 213
LGDT instruction 187
LIDT instruction..... 187
limit 90, 93, 96, 361
linear address 3
Link field 365
LLDT instruction..... 187
LMA bit..... 68
LME bit 68
LMSW instruction..... 184
load ordering 213
local descriptor table (LDT) 82, 90
base address, 64-bit mode 93
limit check, long mode..... 93
local descriptor-table register (LDTR)..... 91
attributes 94
base address..... 93
hidden portion..... 91
LDT selector 93
limit 93
loading..... 187
storing 187
locality..... 170
logical address 2
long attribute (L)
code segment..... 107
effect on D bit..... 108
long mode..... xxix, 14, 29
activating 423
consistency checks 424
differences from legacy mode..... 49
enabling 422
enabling versus activating..... 422
GDT requirements 421
IDT requirements..... 420
leaving..... 425
page translation-table requirements... 421
relocating descriptor tables 424
relocating page tables..... 425
TSS requirements..... 421

use of CS.L and CS.D..... 422
long mode active (EFER.LMA)..... 68, 422
long mode enable (EFER.LME)..... 68, 422
LSB..... xxix
lsb..... xxix
LSTAR register..... 181, 436, 437
LTR instruction..... 187

M

M bit..... 355
machine check
error codes..... 310
error sources..... 304
error-reporting address register (MCi_ADDR)..... 312
error-reporting control register (MCi_CTL) 308
error-reporting miscellaneous register (MCi_MISC)..... 312
error-reporting register banks..... 308
error-reporting status register (MCi_STATUS)..... 309
global-capabilities register (MCG_CAP)..... 306
global-control register (MCG_CTL)..... 307
global-status register (MCG_STATUS) 306
initialization..... 313
machine check registers..... 305
machine-check enable (CR4.MCE)... 60, 262
machine-check exception (#MC)..... 262
fatal..... 304
recoverable..... 304
mask..... xxx
masking
definition of interrupt..... 242
MBZ..... xxx
MCA error code field..... 310
MCE bit..... 60
MCG_CAP register..... 306, 433, 441
MCG_CTL register..... 307, 433, 441
MCG_CTL_P bit..... 306
MCG_STATUS register..... 306, 433, 441
MCi_ADDR registers..... 312, 436, 441
MCi_CTL registers..... 308, 309, 435, 441
MCi_MISC registers..... 312, 436, 442
MCi_STATUS registers..... 309, 435, 441
MCIP bit..... 307
memory..... 191
memory addressing
canonical address form..... 5
effective address..... 2
linear address..... 3
logical address..... 2
near pointers..... 3
physical address..... 3
real address..... 11
RIP-relative address..... 39
segment offset..... 2
virtual address..... 3
memory management..... 6
memory serialization..... 213
memory system..... 191
memory type..... 200
determining effective..... 225
memory-access ordering
description..... 194
read ordering..... 195
write ordering..... 195
memory-mapped I/O
directing reads and writes to..... 233, 237
memory-type range register (MTRR)..... 37
combined with PAT..... 231
effect of paging cache controls..... 225
effects with large page sizes..... 226
fixed range..... 218
identifying features..... 224
initial value..... 415
IORRBase..... 236
IORRMask..... 236
MTRRcap..... 224
MTRRdefType..... 224
MTRRfix16K..... 220
MTRRfix4K..... 220
MTRRfix64K..... 220
MTRRphysBase..... 221
MTRRphysMask..... 221
overlapping ranges..... 227
type field, default..... 216
type field, extended..... 233
variable range..... 220
variable range size and alignment..... 222
MFENCE instruction..... 213
MISCV bit..... 311
MMX registers..... 341
model field..... 414
model-specific error code field..... 311
model-specific registers (MSRs). 37, 69, 186
control-transfer recording..... 390
debug extensions..... 73
debug-control MSR (DebugCtlMSR)... 389
FS.base..... 87
GS.base..... 87
initializing..... 417

machine check 74, 305
memory typing 72, 217
PAT 228
performance monitoring 73, 402
SYSCFG 70
system linkage 72, 181
time-stamp counter 73, 406
TOP_MEM 237
TOP_MEM2 237
modes 12
16-bit xxvi
32-bit xxvi
64-bit ..... xxvi, 15
compatibility ..... xxvi, 15
legacy ..... xxix, 16
long ..... xxix, 14
protected ..... xxxi, 16
real ..... xxxi, 4, 16
virtual-8086 ..... xxxiii, 17
modified state, MOESI 197
MOESI 197
moffset xxx
monitor coprocessor (CR0.MP) 53
MOV CRn instruction 184
MOV DRn instruction 185
MOVSXD instruction 43
MP bit 53, 416
MSB xxx
msb xxx
MSR xxxiv
MSRs 69
MTRRcap register 224, 433, 439
MTRRdefType register 224, 435, 440
MTRRfix16K_n registers 220
MTRRfix4K_n registers 220
MTRRfix64K_n registers 220, 435, 440
MtrrFixDramEn bit 71, 234
MtrrFixDramModEn bit 71, 234
MTRRphysBasen registers 221, 434, 439
MTRRphysMaskn registers 221, 439
MTRRs 217
MtrrTom2En bit 71, 239
MtrrVarDramEn bit 71, 239
MXCSR field 353
MXCSR register 340
MXCSR_MASK field 353

N

NE bit 55, 416
near branch
operand size, 64-bit mode 40
near control transfer 122
near pointers 3
near return 135
nested task (rFLAGS.NT) 65, 377
NMI 250
no-execute (NX)
page protection 173
page-translation tables, bit in 169
nonmaskable interrupt exception (NMI) 250
while in SMM 330
non-PAE paging 148
CR3 format 149
NOP instruction 43
not write-through (CR0.NW) 55, 211
NT bit 65
null selector 84
64-bit mode far return 137
interrupt return from 64-bit mode 293
long mode interrupts 290, 291
long mode stack switch 134
numeric error (CR0.NE) 55, 261
NW bit 55
NX bit 169, 173
NXE bit 69

O

octword xxx
OE exception 260, 263
offset ..... xxx, 105
operand-size prefix 38
operating modes 12
OS bit 404
OS FXSAVE/FXRSTOR support
(CR4.OSFXSR) 338
OS unmasked exception support
(CR4.OSXMMEXCPT) 263, 339
OSFXSR bit 61
OSXMMEXCPT bit 61
OVER bit 312
overflow xxxi
overflow exception (#OF) 250
overflow exception (OE) 260, 263
owned state, MOESI 197

P

P bit 97, 167, 371
packed xxxi
PAE bit 60, 147
PAE paging 31, 148
CR3 format 57, 149
CR3 format, long mode 159
legacy mode 154
long mode 160
page directory 149

page size (PS) 148, 152, 154
page directory pointer 149, 154
page size (PS), page-translation tables ... 168
page table 149
page translation 143
page-attribute table (PAT) 228
combined with MTRR 231
effect on memory access 230
identifying support 230
indexing 229
page-translation tables, bit in 169
page-fault exception (#PF) 167, 173, 174, 259
page-fault virtual address 259
page-global enable (CR4.PGE) 61, 171
page-level cache disable (PCD) 211
CR3, bit in 150
page-translation tables, bit in 168
page-level write-through (PWT) 212
CR3, bit in 150
page-translation tables, bit in 168
page-map level-4 159
page-size extensions (CR4.PSE). 32, 60, 147, 152
40-bit physical address support 148, 153
unsupported in long mode 147
page-translation cache 170
page-translation tables 32
accessed (A) 168
available to software (AVL) 169
dirty (D) 168
global page (G) 169
hierarchy 145
no-execute 169
page directory entry (PDE) 149
page size (PS) 168
page table entry (PTE) 149
page-attribute table (PAT) 169
page-directory pointer entry (PDPE) ... 32, 149, 154
page-level cache disable (PCD) 168
page-level write-through (PWT) 168
page-map level-4 entry (PML4E) ... 32, 159
physical-page base address 166
present (P) 167
read/write (R/W) 167
translation-table base address 166
user/supervisor (U/S) 167
paging 8, 31, 143
See also PAE paging and non-PAE paging.
effect of segment protection 176
protection across translation hierarchy 174
protection checks 172
supported translations 145
paging enable (CR0.PG) 56, 147
activating long mode 147, 424
parameter count field 106
PAT
See page-attribute table (PAT).
PAT bit 169
PAT register 228, 435, 440
PBi bits 390
PC bit 405
PCC bit 311
PCD bit 150, 160, 168
PCE bit 61
PDE 149
PDPE 149
PE bit 53
PE exception 260, 264
PerfCtrn registers 402, 436, 443
PerfEvtSeln registers 403, 436, 443
performance counter 186
performance counter enable (CR4.PCE) 61, 186, 402
performance optimization 26, 401
performance-monitoring counter
overflow 406
PerfCtrn 402
PerfEvtSeln 403
starting and stopping 406
PG bit 56, 147
PGE bit 61
physical address 3, 31
as index into cache 209
physical memory 4
physical-address extensions (CR4.PAE) . 31, 32, 60, 147, 158
activating long mode 147, 423
See also PAE paging.
POP instruction 185
precise exceptions and interrupts 242
precision exception (PE) 260, 264
PREFETCH instruction 213
present (P)
descriptor 97, 371
page-translation tables 167
principle of locality 170
priorities, interrupt 267
privilege level 117
probe, cache 194, 199

during cache disable 211
processor feature identification (rFLAGS.ID) 67
processor halt 188
processor state 410
protected mode ..... xxxi, 16
initial operating environment 419
protected-mode virtual interrupts (CR4.PVI) 59
protection checks
adjusting RPL 188
call gate 129
checking access rights 187
data segment 119
direct call, conforming 126
direct call, nonconforming 124
enabling 78
far return 136
interrupt return 281
interrupt to higher privilege 278
limit check, 64-bit mode 138
long mode changes 35
long mode interrupt 290
long mode interrupt return 292
stack segment 121
type check 139
verifying read/write access 188
protection enable (CR0.PE) 53, 78, 87
PS bit 148, 168
PSE bit 60
PSE paging 32
PTE 149
PUSH instruction 185
PVI bit 59
PWT bit 150, 159, 168

Q

quadword xxxi

R

R bit 99
R/W bit 167, 174
R/W3–R/W0 bits 388
r8–r15 xxxiv
rAX–rSP xxxv
RAZ xxxi
RdMem, MTRR type field 71, 233
RDMSR instruction 69, 186
RDP field 353
RDPMC instruction 61, 186, 402
RDTSC instruction 59, 186, 407
read hit 193
read miss 193
read ordering 213
read/write (R/W)
page protection 174
page-translation tables, bit in 167
readable (R), code segment 99
real address 11
real address mode. See real mode
real mode ..... xxxi, 4, 16
initial operating environment 418
registers
See also entries for individual registers.
128-bit media registers (XMM) 36
address-breakpoint registers (DR0-DR3) 383
control registers 36, 51
control-transfer recording MSRs 390
CR0 53
CR2 259
CR3 ..... 32, 57, 149, 159
CR4 58
CSTAR 181
debug registers 36, 383
debug-control MSR (DebugCtlMSR) ... 389
debug-control register (DR7) 386
debug-extension MSRs 73
debug-status register (DR6) 384
descriptor-table registers 33, 82
eAX–eSP xxxiii
EFER 37, 67
eFLAGS xxxiv
eIP xxxiv
FPR 342, 344
FS and GS 86
FS.base 87
GDTR 89
GPRs 35
GS.base 87
IDTR 95
IORRBase 236
IORRMask 236
last x87 data pointer 342, 344, 353
last x87 instruction pointer .. 342, 344, 353
LDTR 91
LSTAR 181
machine-check MSRs 74
MCG_CAP 306
MCG_CTL 307
MCG_STATUS 306
MCi_ADDR 312
MCi_CTL 308
MCi_MISC 312

MCi_STATUS 309
memory-type range register (MTRR) ... 37, 72, 217
MMX 341
model-specific registers (MSRs)..... 37
MTRR, fixed range 218
MTRR, variable range..... 220
MTRRcap..... 224
MTRRdefType..... 224
MTRRfix16K 220
MTRRfix4K 220
MTRRfix64K 220
MTRRphysBase..... 221
MTRRphysMask 221
MXCSR 340
PAT 228
PerfCtrn 402
PerfEvtSeln 403
performance-monitoring MSRs 73
r8–r15 xxxiv
rAX–rSP xxxv
rFLAGS ..... xxxv, 36, 62
rIP..... xxxv
rSP..... 36
segment registers..... 84
STAR 181
SYSCFG 70
SYSENTER_CS 183
SYSENTER_EIP 183
SYSENTER_ESP..... 183
system-linkage MSRs 72
task-priority register (CR8) 48, 62, 270
time-stamp counter..... 73, 406
TOP_MEM 71, 237
TOP_MEM2 ..... 71, 237
x87 FCW 342, 344, 352
x87 floating-point processor state..... 342
x87 FSW..... 341, 342, 344, 352
x87 FTW..... 341, 342, 344, 352
x87 opcode..... 342, 345, 353
XMM registers..... 340
relative..... xxxi
replacement, cache-line 194
reset 409
processor state 410
RESET# signal 409
resume flag (rFLAGS.RF)..... 65, 249, 398
RET instruction 135
from 64-bit mode..... 137
long mode 42, 136
popping null selector, 64-bit mode..... 137
stack switch..... 136
retire, instruction 194
revision history xxi
REX prefix..... 37
RF bit 65
rFLAGS 36, 62
alignment check (AC) 66, 261
I/O privilege level field (IOPL)..... 64
interrupt flag (IF) 64, 185
nested task (NT)..... 65, 377
processor feature identification (ID) 67
resume flag (RF) 65, 249, 398
trap flag (TF) 63
virtual interrupt (VIF)..... 66, 296
virtual interrupt pending (VIP)..... 66, 296
virtual-8086 mode (VM) 65
rFLAGS register xxxv
rIP 36
rIP register..... xxxv
RIP-relative address..... 39
RIP-relative addressing xxxii
RIPV bit 307
RPL..... 83, 118
adjusting 188
definition 118
in call gate protection..... 130
in data segment protection 119
in far return 136
in IRET instruction 281
in protecting conforming CS 126
in protecting nonconforming CS..... 124
in stack segment protection 121
RPL field..... 360
RSM instruction 319, 335
rSP 36
call gate stack switch 132
implicit reference..... 39

S

S bit 97, 371
SCE bit 68
segment base 97
segment limit..... 96
segment offset 2
segment registers 82, 84
64-bit mode 85
accessing 186
hidden portion..... 84
initializing unused registers..... 84
segmentation 6, 33
64-bit mode 80
combining with paging 10

flat segmentation 7, 10, 80
multi-segmented model 79
segment-not-present exception (#NP) 255
segment-override prefix 38
selector 81, 82, 83, 93, 106, 360
selector index 83
self-modifying code 209
serializing instructions 214
set xxxii
SF exception 260
SFENCE instruction 213
SGDT instruction 187
shared state, MOESI 197
shut down 253
SIDT instruction 187
SIMD floating-point exception (#XF) 61, 263, 339
single-step
all instructions 382, 398
control-transfers 382, 400
SLDT instruction 187
SMBASE register 321
SMI 319
SMM 319
SMM interrupts 330
SMM revision identifier 328
SMM state-save area 322
SMRAM 320
SMRAM state-save area 322
SMSW instruction 184
speculative execution 194
SS register 86
SSE xxxii
SSE-2 xxxii
stack exception (#SS) 256
stack pointers
in 32-bit TSS 365
in 64-bit TSS 369
stack segment 85, 100
64-bit mode 86
default operand size (D) 102
expand down (E) 101
privilege checks 121
stack switch
call gate 132
call gate, long mode 42, 134
far return 136
interrupt 277
interrupt return 281
interrupt, long mode 46
stack-fault exception (SF) 260
STAR register 181, 436, 437
status word 184
stepping ID field 414
STI instruction 185
sticky bits xxxii
store ordering 213
STR instruction 187
supervisor page 173
SWAPGS instruction 183
SYSCALL Flag Mask register 181
SYSCALL, SYSRET instructions 68, 180
SYSCFG
MtrrFixDramEn 71, 234
MtrrFixDramModEn 71, 234
MtrrTom2En 239
MtrrVarDramEn 71, 239
SYSCFG register 70, 436, 440
SYSENTER, SYSEXIT instructions 182
illegal in long mode 182
SYSENTER_CS register 183, 433, 438
SYSENTER_EIP register 183, 433, 438
SYSENTER_ESP register 183, 433, 438
system call and return 179
system data structures 20
system management interrupt (SMI) 319, 329
while in SMM 330
system management mode (SMM) 17, 30
leaving 335
long mode differences 320
operating environment 329
revision identifier 328
saving processor state 332
SMBASE register 321
SMRAM 320
state-save area, legacy 326
state-save area, AMD64 architecture .. 322
system registers 17
system segment 34, 95, 102
ignored fields in 64-bit mode 110
illegal types in long mode 109
long mode 109
type field 103
system-call extension (EFER.SCE) 68
system-linkage MSRs 72, 181

T

T bit 366
table indicator, selector 83
task gate 104
in task switching 375
long mode 114

Task Register (TR) 82
task register (TR) 361
  loading 187
  selector 360
  storing 187
task switch 357, 373
  disabled in long mode 48
  lazy context switch 54, 356
  nesting tasks 377
  preventing recursion 379
task switched (CR0.TS) 54, 185
task, execution space 357
task-priority register (CR8) 48, 62, 270
task-state segment (TSS)
  descriptor 360
  dynamic fields 365
  I/O-permission bitmap 366, 369
  interrupt-redirection bitmap 366
  interrupt-stack table 369
  legacy 32-bit 363
  link field 377
  software-defined fields 365
  stack pointers 365, 369
  static fields 365
TF bit 63
TI bit 83, 360
time-stamp counter 186, 406
time-stamp disable (CR4.TSD) .. 59, 186, 407
TLB 169, 170
  explicit invalidation 172, 189
  implicit invalidation 172
top of memory 237
TOP_MEM register 71, 237, 437, 440
TOP_MEM2 register 71, 237, 437, 440
TPR register 48, 62, 271
TR register 358, 361
translation lookaside buffer (TLB) 170
trap 243
trap flag (rFLAGS.TF) 63
trap gate 104, 285
TS bit 54
TSC register 406, 433, 443
TSD bit 59
TSS ..... xxxii, 358, 363
TSS descriptor 358
TSS selector 106, 358
type check 139
Type field 97, 361, 371

U

U/S bit 167, 173
UC bit 312
UC memory type 200
UD2 instruction 251
UE exception 260, 264
uncacheable (UC), memory type 200
uncacheable (UC-), memory type 228
underflow xxxii
underflow exception (UE) 260, 264
unit mask field 404
user page 173
user segment 95
user/supervisor (U/S)
  page protection 173
  page-translation tables, bit in 167
USR bit 404

V

VAL bit 312
Variable-range IORRs 235
vector xxxii
vector, interrupt 245
VERR instruction 188
VERW instruction 188
VIF bit 66
VIP bit 66
virtual address 3, 31
virtual interrupt (rFLAGS.VIF) 66, 296
virtual interrupt pending (rFLAGS.VIP) 66, 296
virtual interrupts 64, 66, 293, 296
virtual interrupts, protected mode 298
virtual memory 4
virtual-8086 mode ..... xxxiii, 17
  interrupt to protected mode 283
  interrupts 282
virtual-8086 mode (rFLAGS.VM) 65
virtual-8086 mode extensions (CR4.VME) 59, 294
VM bit 65
VME 294
VME bit 59, 282

W

W bit 101
WAIT/FWAIT instruction 53
WB memory type 202
WBINVD instruction 189, 214
WC memory type 201
WP bit 55, 174
WP memory type 201
writable (W), data segment 101
write buffer 194, 204
  emptying 204
write hit 193

write miss 193
write ordering 195, 213
write protect (CR0.WP) 55
  page protection 174
write-back (WB), memory type..... 202
writeback, cache line..... 194
write-combining (WC), memory type 201
write-combining buffer 194, 205
  emptying..... 206
write-protect (WP), memory type 201
write-though (WT), memory type..... 202
WrMem, MTRR type field 71, 233
WRMSR instruction 69, 186
WT memory type 202

X

x87 control word 342, 344, 352
x87 data pointer register 342, 344, 353
x87 environment 344
x87 floating-point instructions
  feature identification 338
  initializing 415
  processor state 342
  saving state..... 344
x87 floating-point state, initialization 411
x87 instruction pointer register 342, 344, 353
x87 opcode register 342, 345, 353
x87 status word 341, 342, 344, 352
x87 tag word 341, 342, 344, 352
  FXSAVE format 354
XMM registers 340

Z

ZE exception 260, 263
zero extension 38, 39
zero-divide exception (ZE) 260, 263