AMD

Rev 12-5-22

Computer Architecture

x86-64

Dr Jeff Drobman

website ➡ drjeffsoftware.com/classroom.html

email ➡ jeffrey.drobman@csun.edu

CSUN
CALIFORNIA STATE UNIVERSITY NORTHRIDGE
COMP222

AMD

AMD64

AMD64 Technology

Rev. 3.23—October 2020

AMD64 Architecture Programmer's Manual: Volumes 1-5

Overview of the AMD64 Architecture

The AMD64 architecture is a simple yet powerful 64-bit, backward-compatible extension of the industry-standard (legacy) x86 architecture. It adds 64-bit addressing and expands register resources to support higher performance for recompiled 64-bit programs, while supporting legacy 16-bit and 32-bit applications and operating systems without modification or recompilation. It is the architectural basis on which new processors can provide seamless, high-performance support for both the vast body of existing software and 64-bit software required for higher-performance applications.

The need for a 64-bit x86 architecture is driven by applications that address large amounts of virtual and physical memory, such as high-performance servers, database management systems, and CAD tools. These applications benefit from both 64-bit addresses and an increased number of registers. The small number of registers available in the legacy x86 architecture limits performance in computation-intensive applications. Increasing the number of registers provides a performance boost to many such applications.

AMD64: General

1.1.1 AMD64 Features

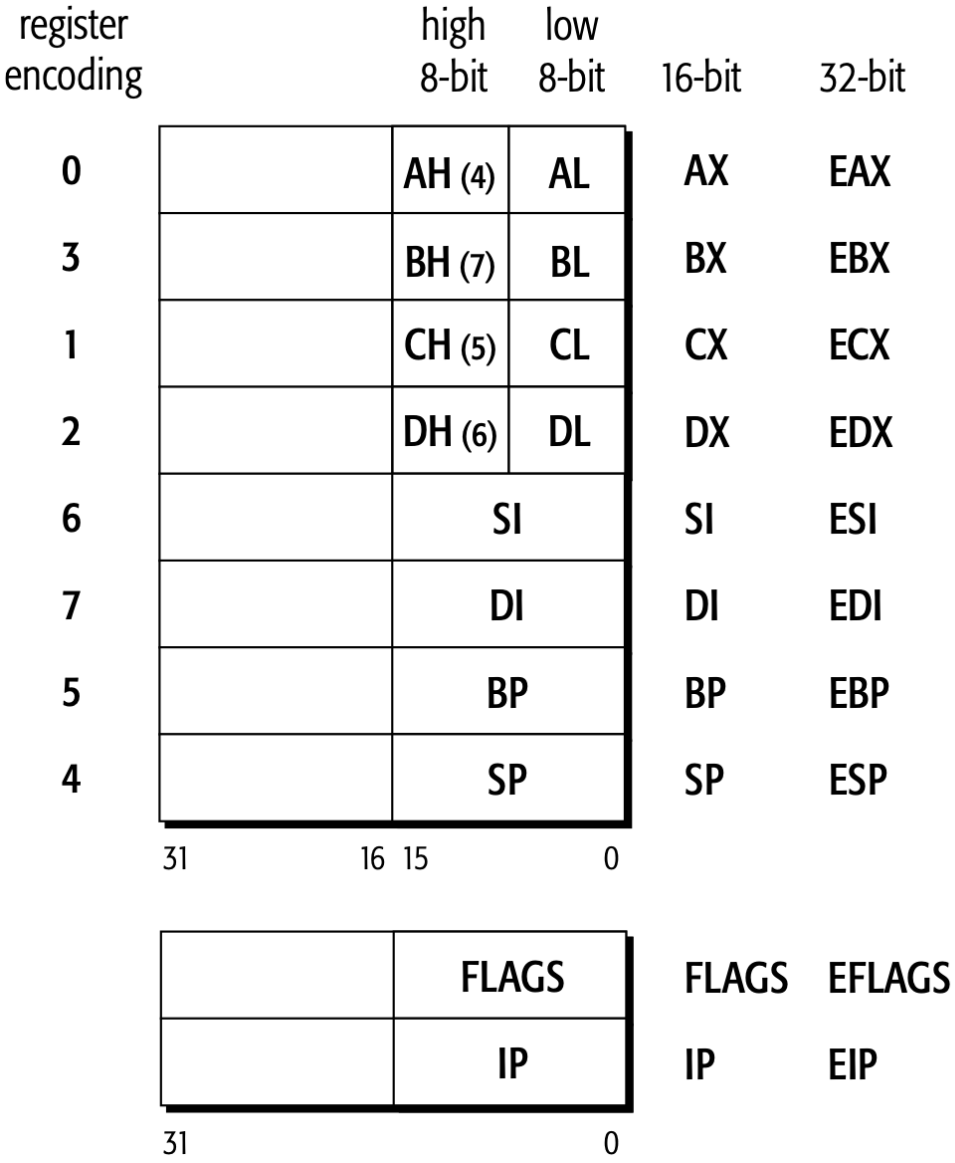
The AMD64 architecture includes these features:

- Register Extensions (see Figure 1-1 on page 2):
  - 8 additional general-purpose registers (GPRs).
  - All 16 GPRs are 64 bits wide.
  - 8 additional YMM/XMM registers.
  - Uniform byte-register addressing for all GPRs.
  - An instruction prefix (REX) accesses the extended registers.
- Long Mode (see Table 1-1 on page 2):
  - Up to 64 bits of virtual address.
  - 64-bit instruction pointer (RIP).
  - Instruction-pointer-relative data-addressing mode.
  - Flat address space.

AMD64: 32-Bit Registers

AMD64 Technology — Rev. 3.23—October 2020

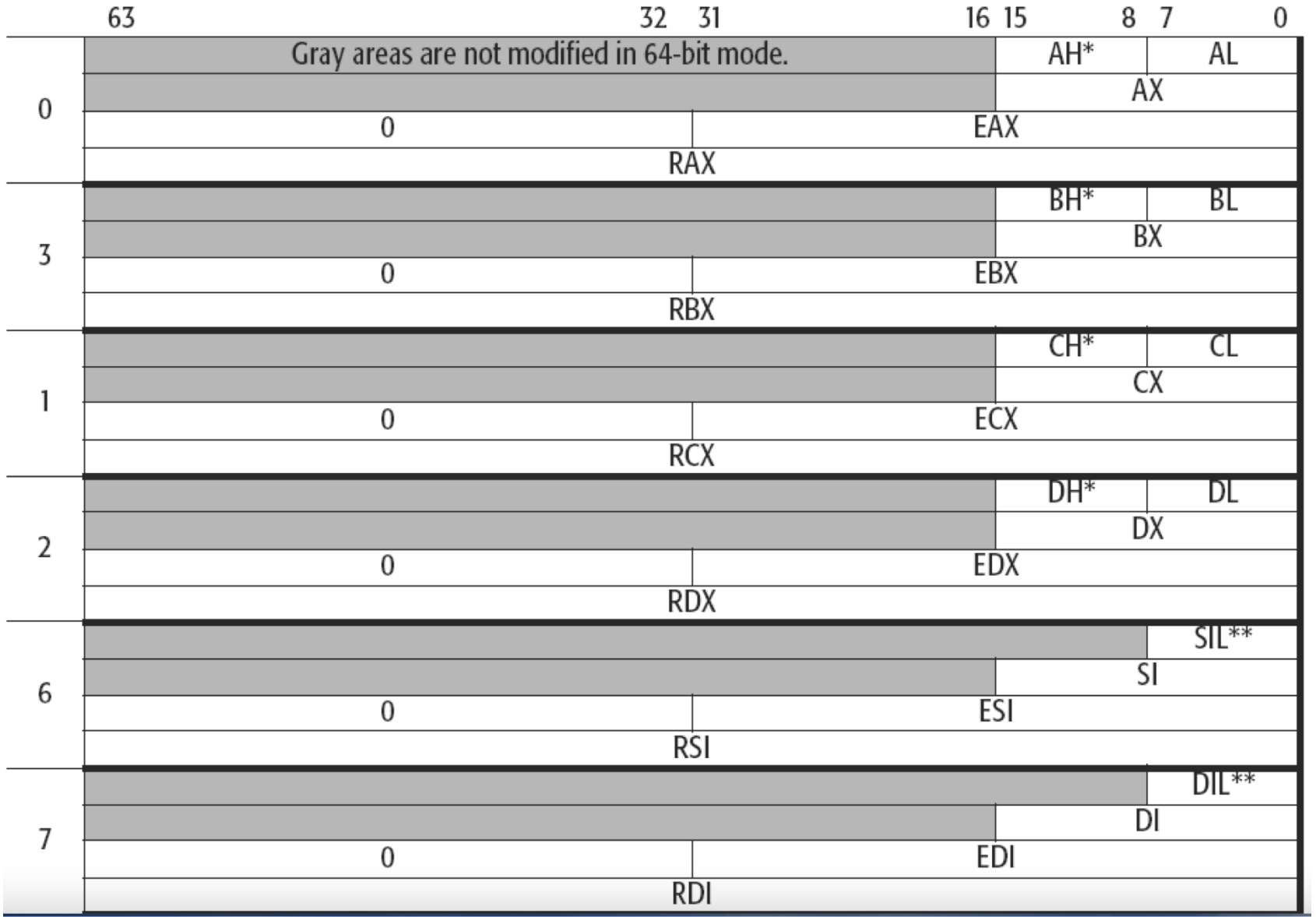
Telescoping EAX



AMD64: 64-Bit Registers

AMD64 Technology — Rev. 3.23—October 2020

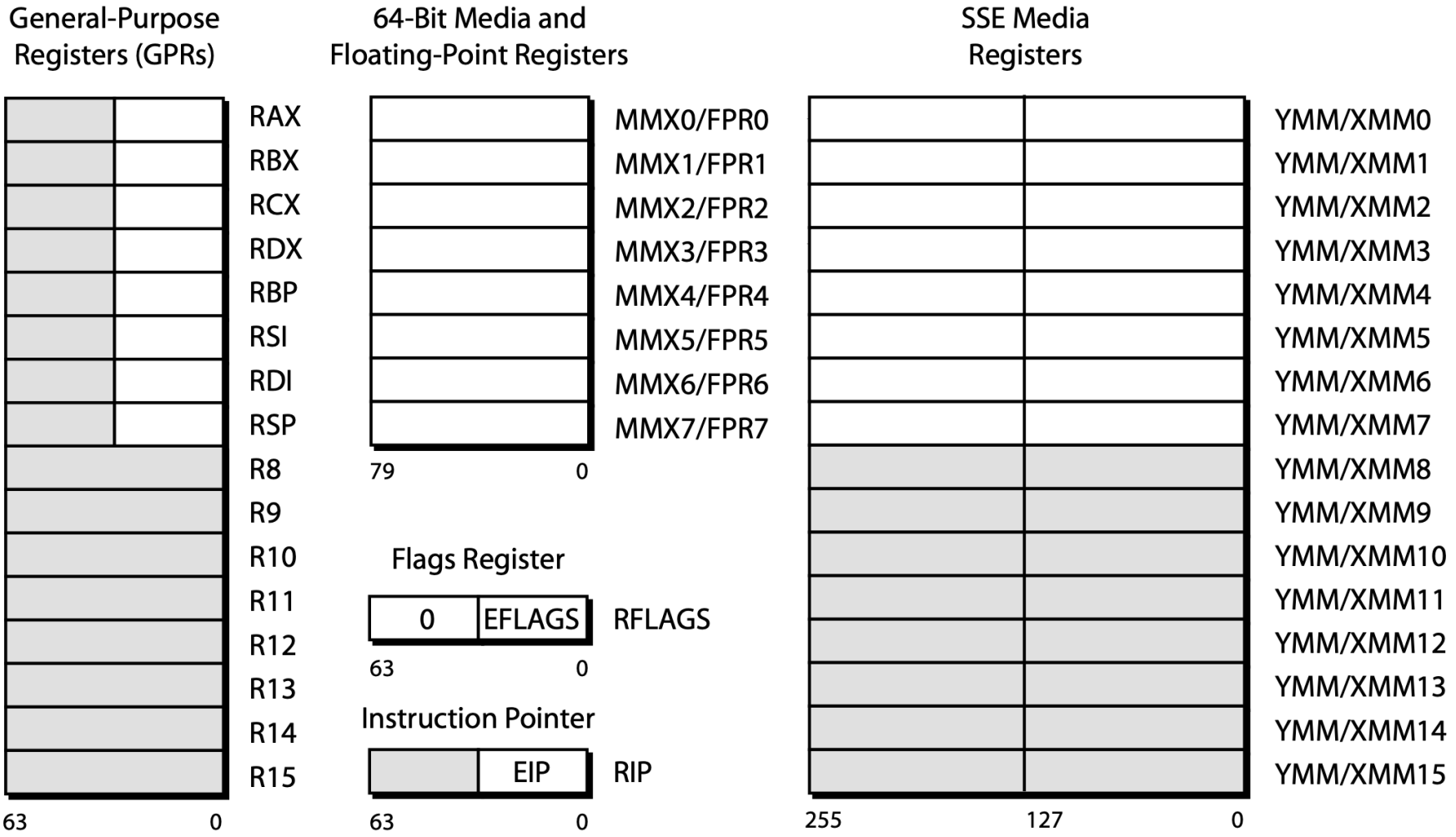
Telescoping RAX



AMD64: 64-Bit Registers

AMD64 Technology — Rev. 3.23—October 2020

Telescoping RAX +8

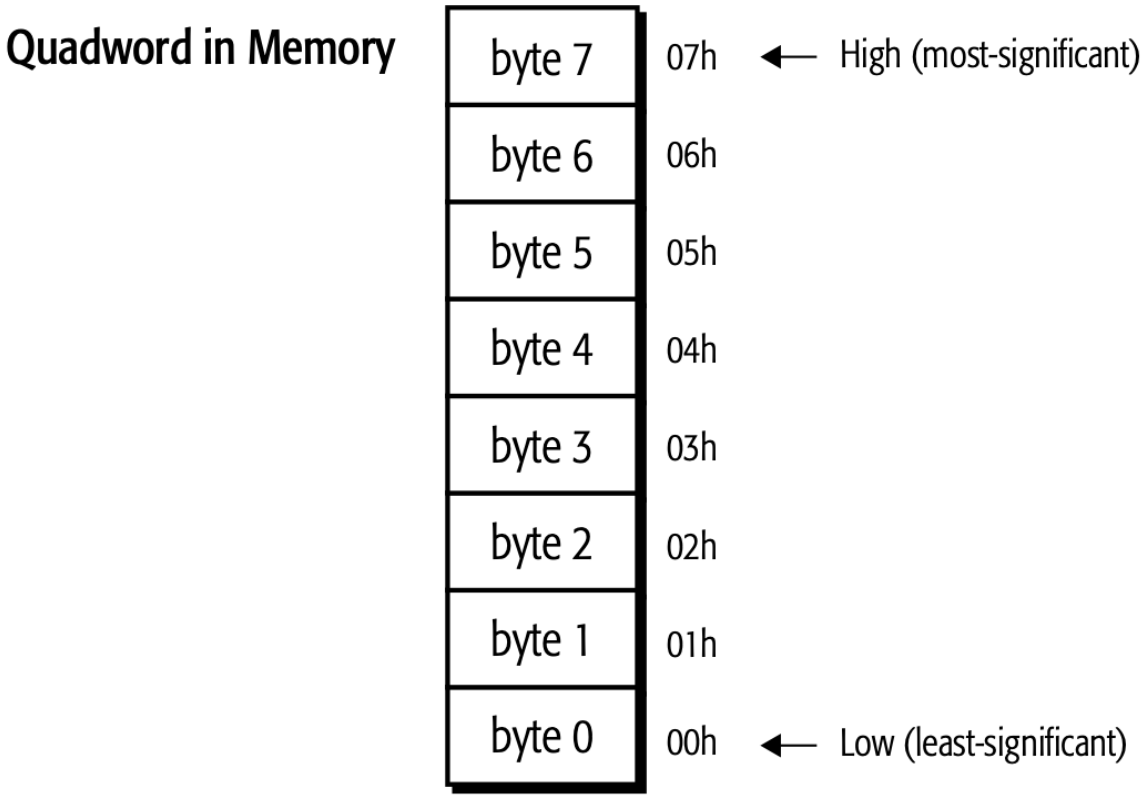


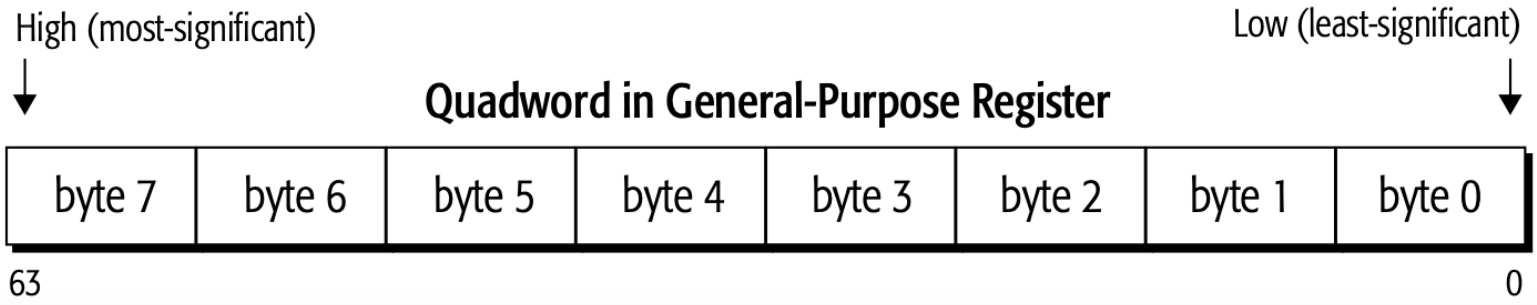
Legacy x86 registers, supported in all modes

Register extensions, supported in 64-bit mode

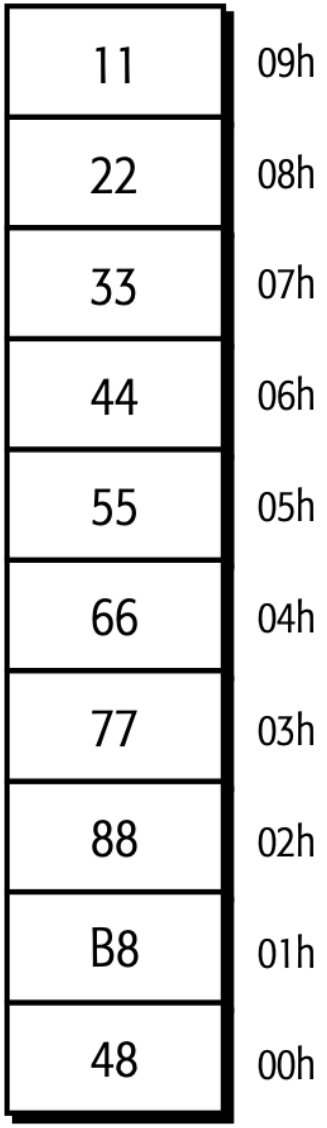
Application-programming registers not shown include Media eXension Control and Status Register (MXCSR) and x87 tag-word, control-word, and status-word registers

Little Endian Data





AMD64: PC=IP Register



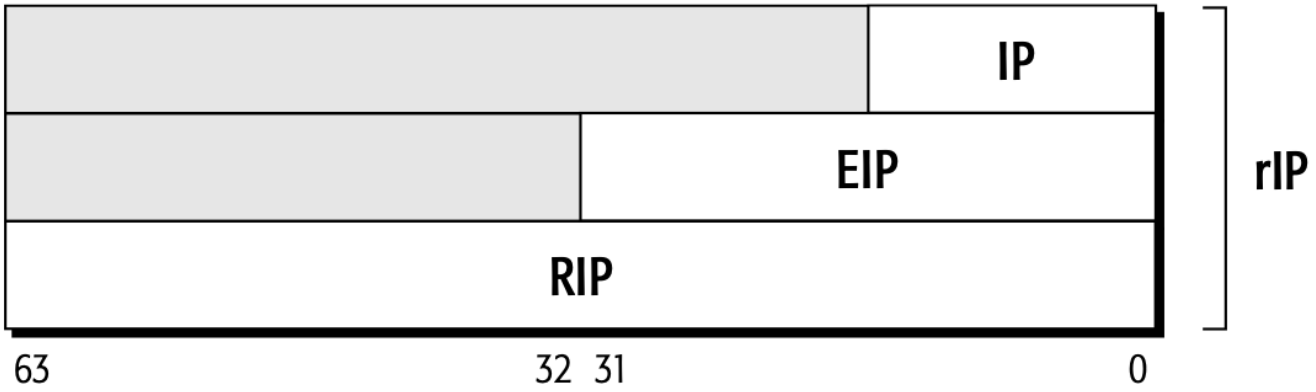


Figure 2-10. Instruction Pointer (rIP) Register

Figure 2-6. Example of 10-Byte Instruction in Memory

Segment Registers

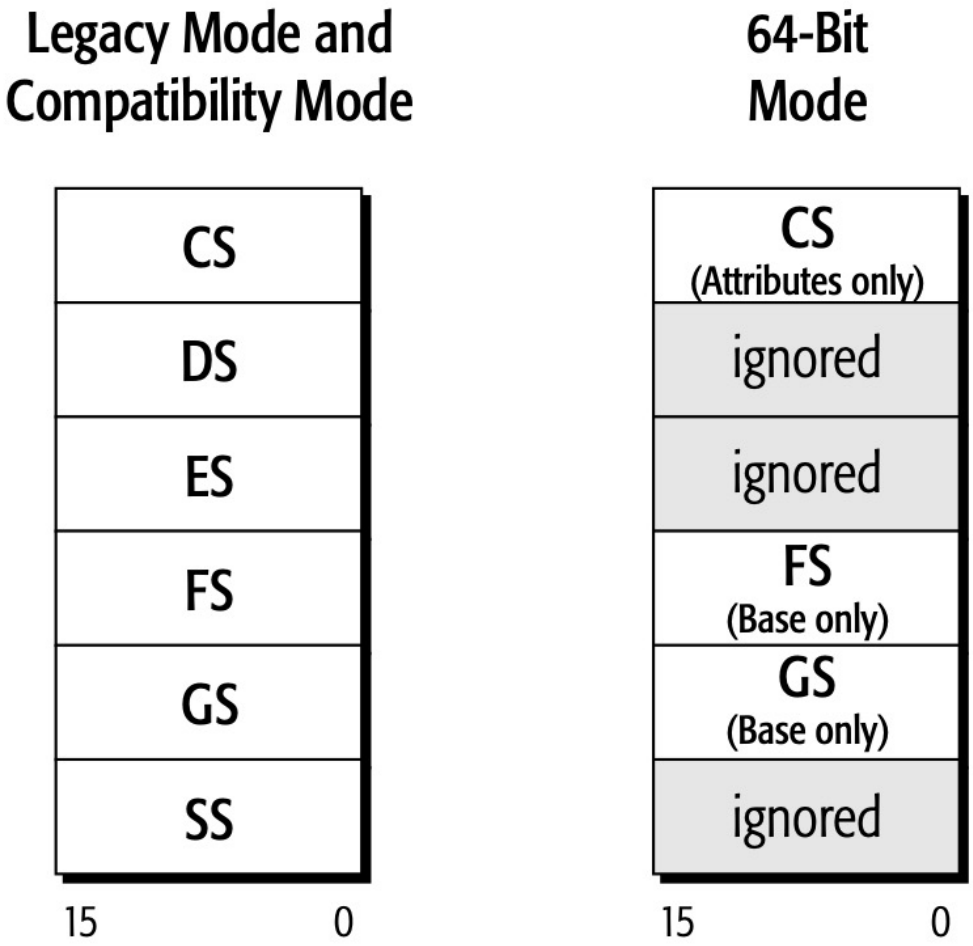


Figure 2-2. Segment Registers

Virtual Memory

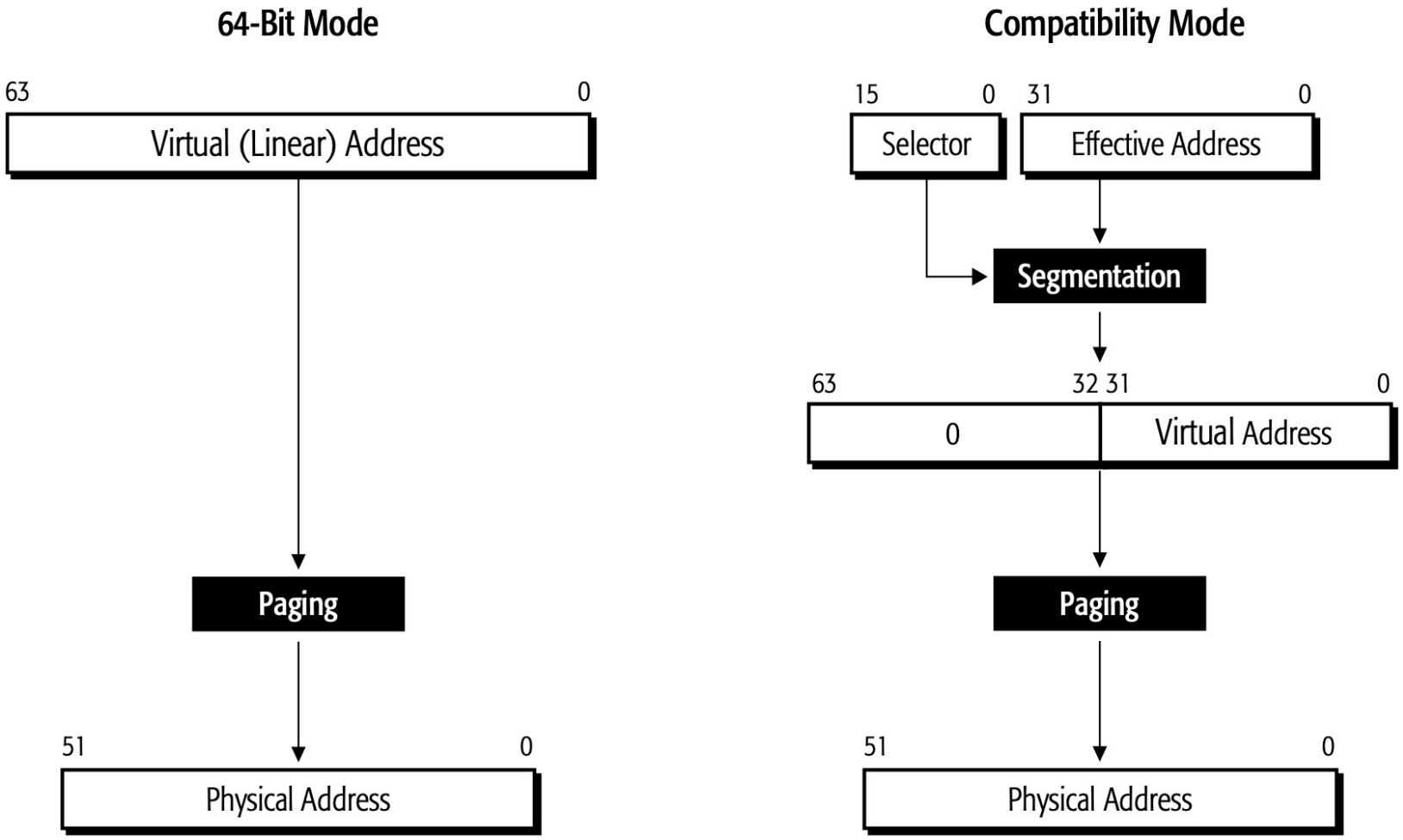


Figure 2-3. Long-Mode Memory Management

Virtual Memory

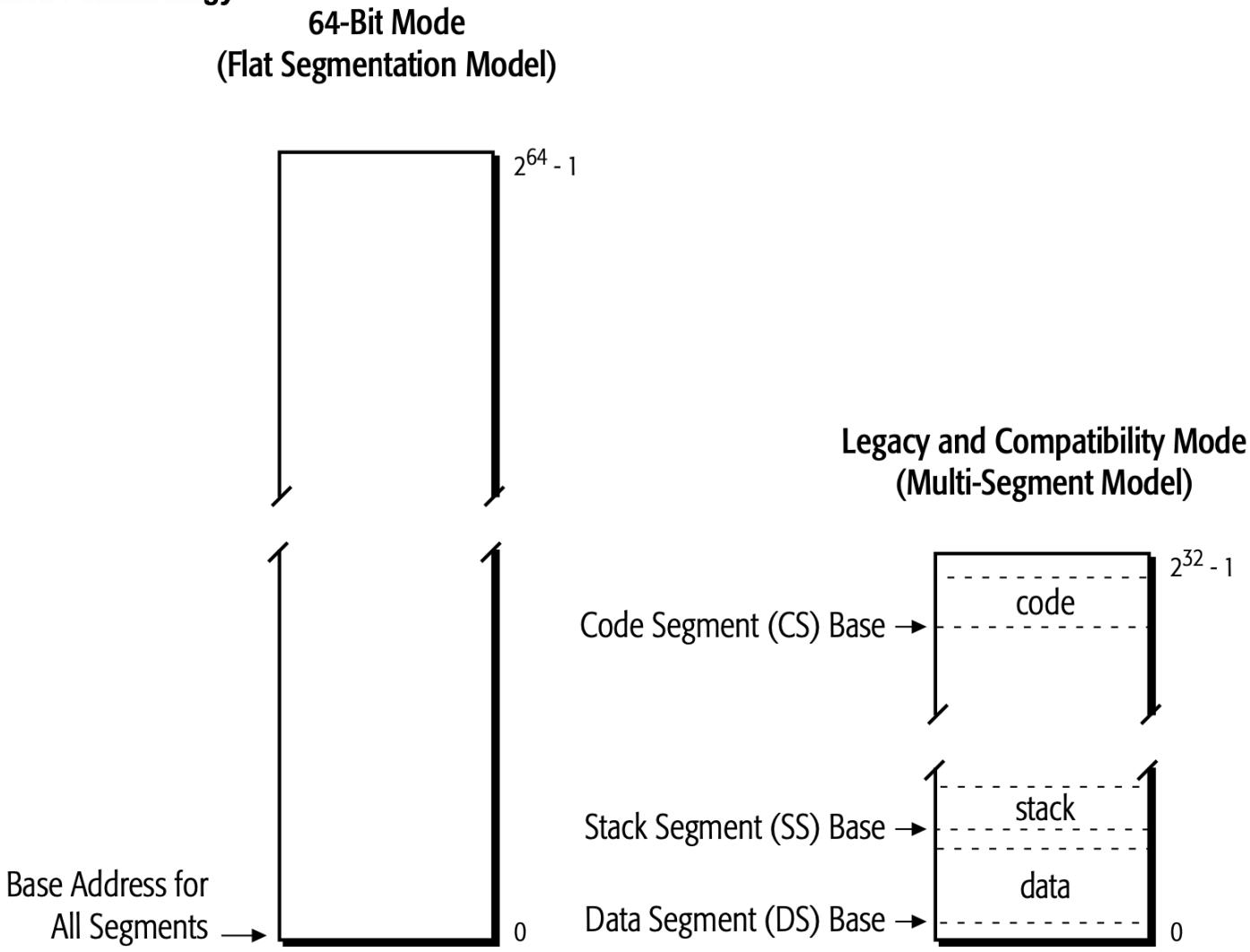


Figure 2-1. Virtual-Memory Segmentation

AMD64: Instr Syntax

3.3.1 Syntax

Each instruction has a *mnemonic syntax* used by assemblers to specify the operation and the operands to be used for source and destination (result) data. Figure 3-7 shows an example of the mnemonic syntax for a compare (CMP) instruction. In this example, the CMP mnemonic is followed by two operands, a 32-bit register or memory operand and an 8-bit immediate operand.

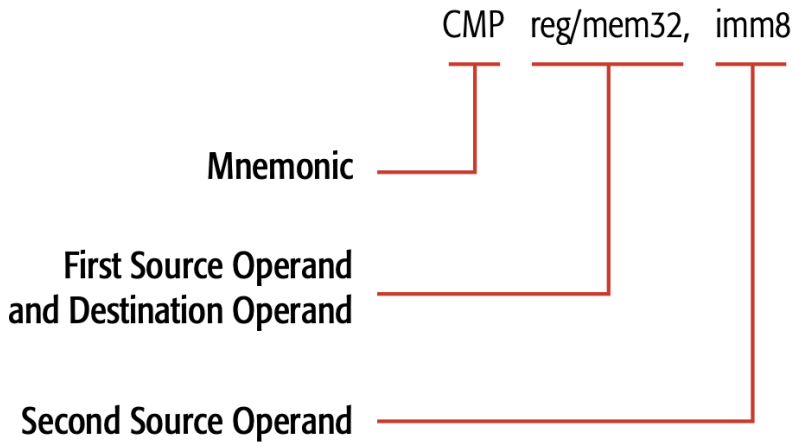


Figure 3-7. Mnemonic Syntax Example

AMD64: Data Sizes

AMD64 Technology
Rev. 3.23—October 2020

3.2.5 Data Alignment

A data access is *aligned* if its address is a multiple of its operand size, in bytes. The following examples illustrate this definition:

- *Byte* accesses are always aligned. Bytes are the smallest addressable parts of memory.
- *Word* (two-byte) accesses are aligned if their address is a multiple of 2.
- *Doubleword* (four-byte) accesses are aligned if their address is a multiple of 4.
- *Quadword* (eight-byte) accesses are aligned if their address is a multiple of 8.

Most others

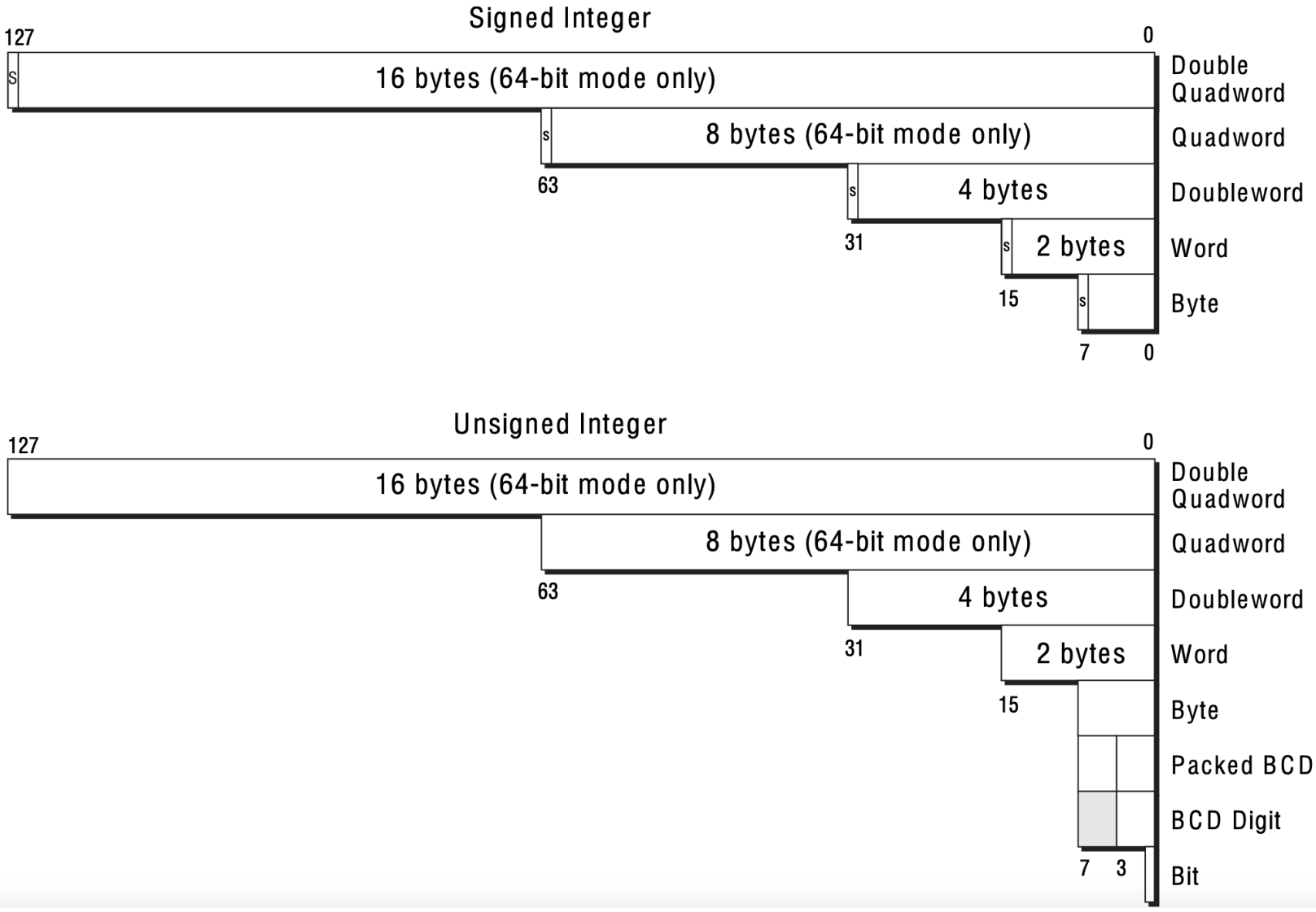
- Byte
- Halfword
- Word
- Doubleword

Java

- Byte
- Short
- Int
- Long

AMD64: Data Sizes

AMD64 Technology — Rev. 3.23—October 2020



AMD64: Data Types

The following data types are supported in the general-purpose programming environment:

- Signed (two's-complement) integers.
- Unsigned integers.
- BCD digits.
- Packed BCD digits.
- Strings, including bit strings.
- Untyped data objects.

AMD64: Data Types

AMD64 Technology — Rev. 3.23—October 2020

The Architecture defines the following fundamental data types:

- Untyped data objects
  - bit
  - nibble (4 bits)
  - byte (8 bits)
  - word (16 bits)
  - doubleword (32 bits)
  - quadword (64 bits)
  - double quadword (octword) (128 bits)
  - double octword (256 bits)
- Unsigned integers
  - 8-bit (byte) unsigned integer
  - 16-bit (word) unsigned integer
  - 32-bit (doubleword) unsigned integer
  - 64-bit (quadword) unsigned integer
  - 128-bit (octword) unsigned integer
- Signed (two's-complement) integers
  - 8-bit (byte) signed integer
  - 16-bit (word) signed integer
  - 32-bit (doubleword) signed integer
  - 64-bit (quadword) signed integer
  - 128-bit (octword) signed integer
- Binary coded decimal (BCD) digits
- Floating-point data types
  - half-precision floating point (16 bits)
  - single-precision floating point (32 bits)
  - double-precision floating point (64 bits)

AMD64: ALU Instructions

Multiply and Divide

- MUL—Multiply Unsigned
- IMUL—Signed Multiply
- DIV—Unsigned Divide
- IDIV—Signed Divide

AMD64: Modes

AMD64 Technology

Rev. 3.23—October 2020

Table 1-1. Operating Modes

Operating Mode		Operating System Required	Application Recompile Required	Defaults		Register Extensions	Typical
				Address Size (bits)	Operand Size (bits)		GPR Width (bits)
Long Mode	64-Bit Mode	64-bit OS	yes	64	32	yes	64
	Compatibility Mode		no	32		no	32
				16	16		16
Legacy Mode	Protected Mode	Legacy 32-bit OS	no	32	32	no	32
				16	16		
	Virtual-8086 Mode			16	16		16
	Real Mode	Legacy 16-bit OS					

AMD64: FPU Instructions

AMD64 Technology Rev. 3.23—October 2020

- SSE
- MMX
- Legacy

1.1.5 Floating-Point Instructions

The AMD64 architecture provides three floating-point instruction subsets, using three distinct register sets:

- SSE instructions support 32-bit single-precision and 64-bit double-precision floating-point operations, in addition to integer operations. Operations on both vector data and scalar data are supported, with a dedicated floating-point exception-reporting mechanism. These floating-point operations comply with the IEEE-754 standard.
- MMX Instructions support single-precision floating-point operations. Operations on both vector data and scalar data are supported, but these instructions do not support floating-point exception reporting.
- x87 Floating-Point Instructions support single-precision, double-precision, and 80-bit extended-precision floating-point operations. Only scalar data are supported, with a dedicated floating-point exception-reporting mechanism. The x87 floating-point instructions contain special instructions for performing trigonometric and logarithmic transcendental operations. The single-precision and double-precision floating-point operations comply with the IEEE-754 standard.

Maximum floating-point performance can be achieved using the 256-bit media instructions. One of

AMD64: Move Data

3.3.2 Data Transfer

The data-transfer instructions copy data between registers and memory.

Move

- MOV—Move
- MOVBE—Move Big-Endian LE default
- MOVSX—Move with Sign-Extend
- MOVZX—Move with Zero-Extend
- MOVD—Move Doubleword or Quadword
- MOVNTI—Move Non-temporal Doubleword or Quadword

AMD64: LEA Instructions

3.3.5 Load Effective Address

- LEA—Load Effective Address

LEA is related to MOV, which copies data from a memory location to a register, but LEA takes the address of the source operand, whereas MOV takes the contents of the memory location specified by the source operand. In the simplest cases, LEA can be replaced with MOV. For example:

```
lea eax, [ebx]
```

has the same effect as:

```
mov eax, ebx
```

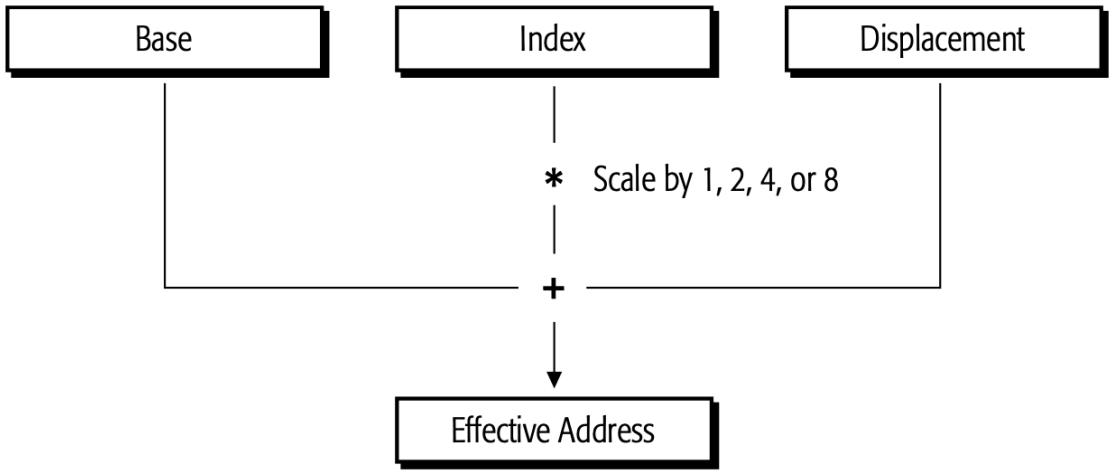


Figure 2-7. Complex Address Calculation (Protected Mode)

AMD64: Cond'l Move

AMD64 Technology — Rev. 3.23—October 2020

Mnemonic	Required Flag State	Description
CMOVAE CMOVNB CMOVNC	CF = 0	Conditional move if above or equal Conditional move if not below Conditional move if not carry
CMOVE CMOVZ	ZF = 1	Conditional move if equal Conditional move if zero
CMOVNE CMOVNZ	ZF = 0	Conditional move if not equal Conditional move if not zero
CMOVBE CMOVNA	CF = 1 or ZF = 1	Conditional move if below or equal Conditional move if not above
CMOVA CMOVNBE	CF = 0 and ZF = 0	Conditional move if not below or equal Conditional move if not below or equal
CMOVS	SF = 1	Conditional move if sign
CMOVNS	SF = 0	Conditional move if not sign
CMOVP CMOVPE	PF = 1	Conditional move if parity Conditional move if parity even
CMOVNP CMOVPO	PF = 0	Conditional move if not parity Conditional move if parity odd

AMD64: Cond'l Move

In assembly languages, the conditional move instructions correspond to small conditional statements like:

```
IF a = b THEN x = y
```

CMOV*cc* instructions can replace two instructions—a conditional jump and a move. For example, to perform a high-level statement like:

```
IF ECX = 5 THEN EAX = EBX
```

without a CMOV*cc* instruction, the code would look like:

```
cmp ecx, 5          ; test if ecx equals 5
jnz Continue        ; test condition and skip if not met
mov eax, ebx        ; move
Continue:           ; continuation
```

but with a CMOV*cc* instruction, the code would look like:

```
cmp ecx, 5          ; test if ecx equals to 5
cmovz eax, ebx      ; test condition and move
```

Replacing conditional jumps with conditional moves also has the advantage that it can avoid branch-prediction penalties that may be caused by conditional jumps.

AMD64: Stack Instructions

Stack Operations

- POP—Pop Stack
- POPA—Pop All to GPR Words
- POPAD—Pop All to GPR Doublewords
- PUSH—Push onto Stack
- PUSHA—Push All GPR Words onto Stack
- PUSHAD—Push All GPR Doublewords onto Stack
- ENTER—Create Procedure Stack Frame
- LEAVE—Delete Procedure Stack Frame

AMD64: Conditionals

AMD64 Technology
Rev. 3.23—October 2020

IF A = B THEN GOTO FarLabel

- Jump vs Branch

where FarLabel is located in another code segment, use the opposite condition
jump before the unconditional far jump. For example:

compare
skip

```
        cmp     A, B                    ; compare operands
        jne     NextInstr               ; continue program if not equal
        jmp     far ptr WhenNE          ; far jump if operands are equal
NextInstr:                              ; continue program
```

Loop

- LOOP*cc*—Loop if *condition*

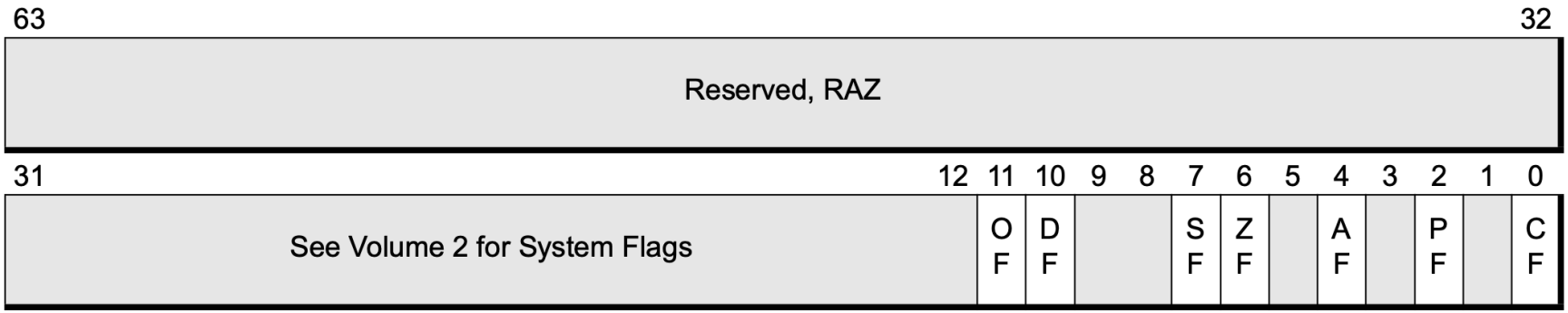
The LOOP*cc* instructions include LOOPE, LOOPNE, LOOPNZ, and LOOPZ.

Call

- CALL—Procedure Call

AMD64: Flags

AMD64 Technology
Rev. 3.23—October 2020



Bits	Mnemonic	Description	R/W
11	OF	Overflow Flag	R/W
10	DF	Direction Flag	R/W
7	SF	Sign Flag	R/W
6	ZF	Zero Flag	R/W
4	AF	Auxiliary Carry Flag	R/W
2	PF	Parity Flag	R/W
0	CF	Carry Flag	R/W

C V N Z

P D AC

Figure 3-5. rFLAGS Register—Flags Visible to Application Software

AMD64: Flag Instructions

Set and Clear Flags

- CLC—Clear Carry Flag
- CMC—Complement Carry Flag
- STC—Set Carry Flag
- CLD—Clear Direction Flag
- STD—Set Direction Flag
- CLI—Clear Interrupt Flag
- STI—Set Interrupt Flag

AMD64: Instructions

MIPS: syscall

ARM: SWI/SVC

3.3.19 System Calls

System Call and Return

- SYSENTER—System Call
- SYSEXIT—System Return
- SYSCALL—Fast System Call
- SYSRET—Fast System Return

AMD64: Int Instructions

Interrupts and Exceptions

- INT—Interrupt to Vector Number
- INTO—Interrupt to Overflow Vector
- IRET—Interrupt Return Word
- IRETD—Interrupt Return Doubleword
- IRETQ—Interrupt Return Quadword

AMD64: Int Instructions

Endian Conversion

Little ← → Big

- BSWAP—Byte Swap

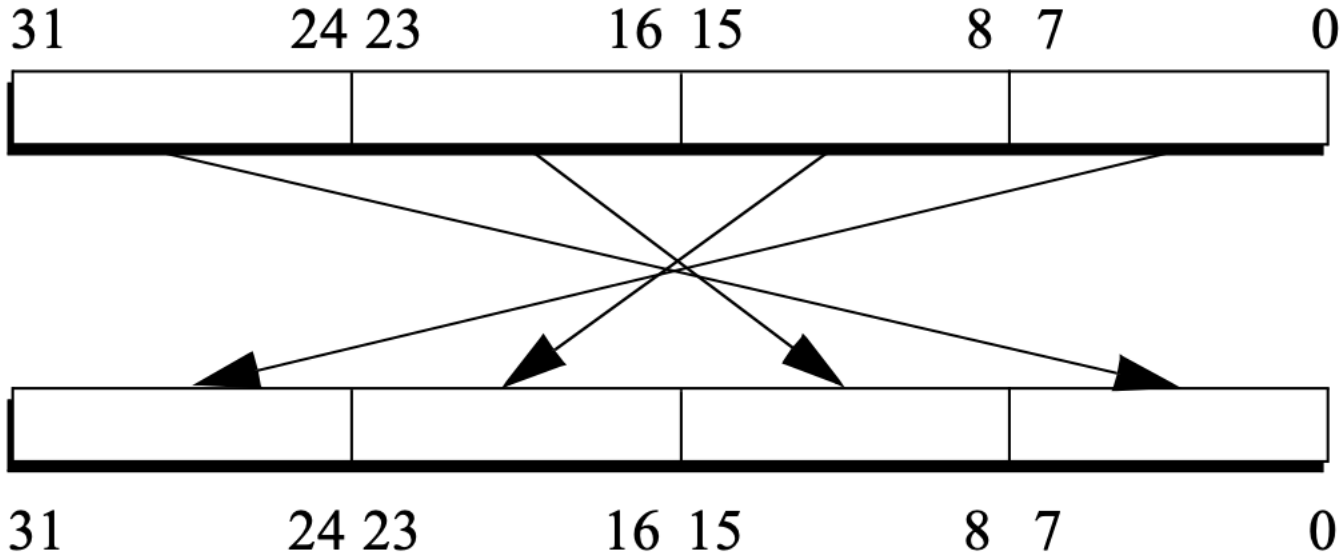


Figure 3-8. BSWAP Doubleword Exchange

AMD64: I/O Instructions

General I/O

- IN—Input from Port
- OUT—Output to Port

String I/O

- INS—Input String
- INSB—Input String Byte
- INSW—Input String Word
- INSD—Input String Doubleword
- OUTS—Output String

- I/O may also be ***MMIO***
  - use **MOV**

AMD64: Privilege Levels

- Onion Skin model

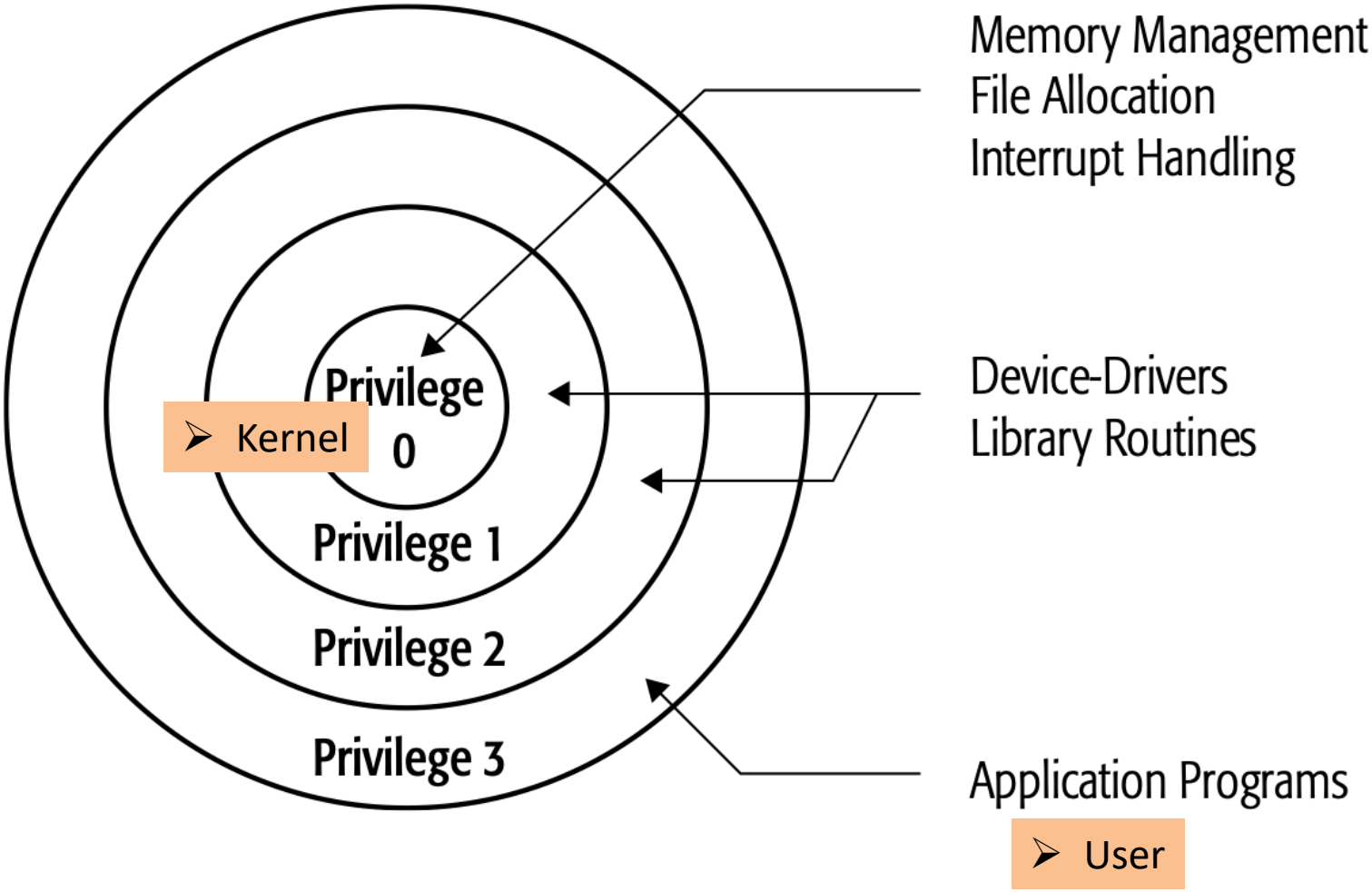


Figure 3-9. Privilege-Level Relationships