1 Overview of the AMD64 Architecture

1.1 Introduction

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 new 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.

1.1.1 New Features

The AMD64 architecture introduces these new features:

- **Register Extensions (see Figure 1-1 on page 2):**
  - 8 new general-purpose registers (GPRs).
  - All 16 GPRs are 64 bits wide.
  - 8 new 128-bit XMM registers.
  - Uniform byte-register addressing for all GPRs.
  - A new instruction prefix (REX) accesses the extended registers.

- **Long Mode (see Table 1-1 on page 3):**
  - Up to 64 bits of virtual address.
  - 64-bit instruction pointer (RIP).
  - New instruction-pointer-relative data-addressing mode.
  - Flat address space.
Figure 1-1. Application-Programming Register Set
Table 1-1. Operating Modes

<table>
<thead>
<tr>
<th>Operating Mode</th>
<th>Operating System Required</th>
<th>Application Recompile Required</th>
<th>Defaults</th>
<th>Register Extensions</th>
<th>Typical</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Address Size (bits)</td>
<td>Operand Size (bits)</td>
<td>GPR Width (bits)</td>
</tr>
<tr>
<td>Long Mode</td>
<td></td>
<td>64-bit Mode</td>
<td>yes</td>
<td>64</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>New 64-bit OS</td>
<td>no</td>
<td>32</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>no</td>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>Legacy Mode</td>
<td></td>
<td>Protected Mode</td>
<td>32</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Legacy 32-bit OS</td>
<td>no</td>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Virtual-8086 Mode</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Real Mode</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

1.1.2 Registers

Table 1-2 on page 4 compares the register and stack resources available to application software, by operating mode. The left set of columns shows the legacy x86 resources, which are available in the AMD64 architecture’s legacy and compatibility modes. The right set of columns shows the comparable resources in 64-bit mode. Gray shading indicates differences between the modes. These register differences (not including stack-width difference) represent the register extensions shown in Figure 1-1.
As Table 1-2 shows, the legacy x86 architecture (called legacy mode in the AMD64 architecture) supports eight GPRs. In reality, however, the general use of at least four registers (EBP, ESI, EDI, and ESP) is compromised because they serve special purposes when executing many instructions. The AMD64 architecture’s addition of eight new GPRs—and the increased width of these registers from 32 bits to 64 bits—allows compilers to substantially improve software performance. Compilers have more flexibility in using registers to hold variables. Compilers can also minimize memory traffic—and thus boost performance—by localizing work within the GPRs.

1.1.3 Instruction Set

The AMD64 architecture supports the full legacy x86 instruction set, and it adds a few new instructions to support long mode (see Table 1-1 for a summary of operating modes). The application-programming instructions are organized and described in the following subsets:

- **General-Purpose Instructions**—These are the basic x86 integer instructions used in virtually all programs. Most of

<table>
<thead>
<tr>
<th>Register or Stack</th>
<th>Legacy and Compatibility Modes</th>
<th>64-Bit Mode¹</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Name</strong></td>
<td><strong>Number</strong></td>
<td><strong>Size (bits)</strong></td>
</tr>
<tr>
<td>General-Purpose Registers (GPRs)²</td>
<td>EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP</td>
<td>8</td>
</tr>
<tr>
<td>128-Bit XMM Registers</td>
<td>XMM0–XMM7</td>
<td>8</td>
</tr>
<tr>
<td>64-Bit MMX Registers</td>
<td>MMX0–MMX7³</td>
<td>8</td>
</tr>
<tr>
<td>x87 Registers</td>
<td>FPR0–FPR7³</td>
<td>8</td>
</tr>
<tr>
<td>Instruction Pointer²</td>
<td>EIP</td>
<td>1</td>
</tr>
<tr>
<td>Flags²</td>
<td>EFLAGS</td>
<td>1</td>
</tr>
<tr>
<td>Stack</td>
<td>–</td>
<td>16 or 32</td>
</tr>
</tbody>
</table>

**Note:**
1. Gray-shaded entries indicate differences between the modes. These differences (except stack-width difference) are the AMD64 architecture’s register extensions.
2. This list of GPRs shows only the 32-bit registers. The 16-bit and 8-bit mappings of the 32-bit registers are also accessible, as described in “Registers” on page 27.
3. The MMX0–MMX7 registers are mapped onto the FPR0–FPR7 physical registers, as shown in Figure 1-1. The x87 stack registers, ST(0)–ST(7), are the logical mappings of the FPR0–FPR7 physical registers.
these instructions load, store, or operate on data located in the general-purpose registers (GPRs) or memory. Some of the instructions alter sequential program flow program by branching to other program locations.

- **128-Bit Media Instructions**—These are the *streaming SIMD extension* (SSE and SSE2) instructions that load, store, or operate on data located primarily in the 128-bit XMM registers. They perform integer and floating-point operations on vector (packed) and scalar data types. Because the vector instructions can independently and simultaneously perform a single operation on multiple sets of data, they are called *single-instruction, multiple-data* (SIMD) instructions. They are useful for high-performance media and scientific applications that operate on blocks of data.

- **64-Bit Media Instructions**—These are the *multimedia extension* (MMX™ technology) and AMD 3DNow!™ technology instructions. They load, store, or operate on data located primarily on the 64-bit MMX registers. Like their 128-bit counterparts, described above, they perform integer and floating-point operations on vector (packed) and scalar data types. Thus, they are also SIMD instructions and are useful in media applications that operate on blocks of data.

- **x87 Floating-Point Instructions**—These are the floating-point instructions used in legacy x87 applications. They load, store, or operate on data located in the x87 registers.

Some of these application-programming instructions bridge two or more of the above subsets. For example, there are instructions that move data between the general-purpose registers and the XMM or MMX registers, and many of the integer vector (packed) instructions can operate on either XMM or MMX registers, although not simultaneously. If instructions bridge two or more subsets, their descriptions are repeated in all subsets to which they apply.

**1.1.4 Media Instructions**

Media applications—such as image processing, music synthesis, speech recognition, full-motion video, and 3D graphics rendering—share certain characteristics:

- They process large amounts of data.
- They often perform the same sequence of operations repeatedly across the data.
The data are often represented as small quantities, such as 8 bits for pixel values, 16 bits for audio samples, and 32 bits for object coordinates in floating-point format.

The 128-bit and 64-bit media instructions are designed to accelerate these applications. The instructions use a form of vector (or packed) parallel processing known as single-instruction, multiple data (SIMD) processing. This vector technology has the following characteristics:

- A single register can hold multiple independent pieces of data. For example, a single 128-bit XMM register can hold 16 8-bit integer data elements, or four 32-bit single-precision floating-point data elements.
- The vector instructions can operate on all data elements in a register, independently and simultaneously. For example, a PADDB instruction operating on byte elements of two vector operands in 128-bit XMM registers performs 16 simultaneous additions and returns 16 independent results in a single operation.

128-bit and 64-bit media instructions take SIMD vector technology a step further by including special instructions that perform operations commonly found in media applications. For example, a graphics application that adds the brightness values of two pixels must prevent the add operation from wrapping around to a small value if the result overflows the destination register, because an overflow result can produce unexpected effects such as a dark pixel where a bright one is expected. The 128-bit and 64-bit media instructions include saturating-arithmetic instructions to simplify this type of operation. A result that otherwise would wrap around due to overflow or underflow is instead forced to saturate at the largest or smallest value that can be represented in the destination register.

1.1.5 **Floating-Point Instructions**

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

- **128-Bit Media 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.
• **64-Bit Media Instructions** (the subset of 3DNow! technology 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 128-bit media instructions. One of these vector instructions can support up to four single-precision (or two double-precision) operations in parallel. In 64-bit mode, the AMD64 architecture doubles the number of legacy XMM registers from 8 to 16.

Applications gain additional benefits using the 64-bit media and x87 instructions. The separate register sets supported by these instructions relieve pressure on the XMM registers available to the 128-bit media instructions. This provides application programs with three distinct sets of floating-point registers. In addition, certain high-end implementations of the AMD64 architecture may support 128-bit media, 64-bit media, and x87 instructions with separate execution units.

### 1.2 Modes of Operation

Table 1-1 on page 3 summarizes the modes of operation supported by the AMD64 architecture. In most cases, the default address and operand sizes can be overridden with instruction prefixes. The register extensions shown in the second-from-right column of Table 1-1 are those illustrated in Figure 1-1 on page 2.

1.2.1 **Long Mode**

Long mode is an extension of legacy protected mode. Long mode consists of two submodes: **64-bit mode** and **compatibility mode**. 64-bit mode supports all of the new features and register extensions of the AMD64 architecture. Compatibility mode
supports binary compatibility with existing 16-bit and 32-bit applications. Long mode does not support legacy real mode or legacy virtual-8086 mode, and it does not support hardware task switching.

Throughout this document, references to long mode refer to both 64-bit mode and compatibility mode. If a function is specific to either of these submodes, then the name of the specific submode is used instead of the name long mode.

1.2.2 64-Bit Mode

64-bit mode—a submode of long mode—supports the full range of 64-bit virtual-addressing and register-extension features. This mode is enabled by the operating system on an individual code-segment basis. Because 64-bit mode supports a 64-bit virtual-address space, it requires a new 64-bit operating system and tool chain. Existing application binaries can run without recompilation in compatibility mode, under an operating system that runs in 64-bit mode, or the applications can also be recompiled to run in 64-bit mode.

Addressing features include a 64-bit instruction pointer (RIP) and a new RIP-relative data-addressing mode. This mode accommodates modern operating systems by supporting only a flat address space, with single code, data, and stack space.

Register Extensions. 64-bit mode implements register extensions through a new group of instruction prefixes, called REX prefixes. These extensions add eight GPRs (R8–R15), widen all GPRs to 64 bits, and add eight 128-bit XMM registers (XMM8–XMM15).

The REX instruction prefixes also provide a new byte-register capability that makes the low byte of any of the sixteen GPRs available for byte operations. This results in a uniform set of byte, word, doubleword, and quadword registers that is better suited to compiler register-allocation.

64-Bit Addresses and Operands. In 64-bit mode, the default virtual-address size is 64 bits (implementations can have fewer). The default operand size for most instructions is 32 bits. For most instructions, these defaults can be overridden on an instruction-by-instruction basis using instruction prefixes. REX prefixes specify the 64-bit operand size and new registers.

RIP-Relative Data Addressing. 64-bit mode supports data addressing relative to the 64-bit instruction pointer (RIP). The legacy x86
architecture supports IP-relative addressing only in control-transfer instructions. RIP-relative addressing improves the efficiency of position-independent code and code that addresses global data.

Opcodes. A few instruction opcodes and prefix bytes are redefined to allow register extensions and 64-bit addressing. These differences are described in “General-Purpose Instructions in 64-Bit Mode” in Volume 3 and “Differences Between Long Mode and Legacy Mode” in Volume 3.

1.2.3 Compatibility Mode

Compatibility mode—the second submode of long mode—allows 64-bit operating systems to run existing 16-bit and 32-bit x86 applications. These legacy applications run in compatibility mode without recompilation.

Applications running in compatibility mode use 32-bit or 16-bit addressing and can access the first 4GB of virtual-address space. Legacy x86 instruction prefixes toggle between 16-bit and 32-bit address and operand sizes.

As with 64-bit mode, compatibility mode is enabled by the operating system on an individual code-segment basis. Unlike 64-bit mode, however, x86 segmentation functions the same as in the legacy x86 architecture, using 16-bit or 32-bit protected-mode semantics. From the application viewpoint, compatibility mode looks like the legacy x86 protected-mode environment. From the operating-system viewpoint, however, address translation, interrupt and exception handling, and system data structures use the 64-bit long-mode mechanisms.

1.2.4 Legacy Mode

Legacy mode preserves binary compatibility not only with existing 16-bit and 32-bit applications but also with existing 16-bit and 32-bit operating systems. Legacy mode consists of the following three submodes:

- **Protected Mode**—Protected mode supports 16-bit and 32-bit programs with memory segmentation, optional paging, and privilege-checking. Programs running in protected mode can access up to 4GB of memory space.
- **Virtual-8086 Mode**—Virtual-8086 mode supports 16-bit real-mode programs running as tasks under protected mode. 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.
- **Real Mode**—Real mode supports 16-bit programs using simple register-based memory segmentation. It does not support paging or protection-checking. Programs running in real mode can access up to 1MB of memory space.

Legacy mode is compatible with existing 32-bit processor implementations of the x86 architecture. Processors that implement the AMD64 architecture boot in legacy real mode, just like processors that implement the legacy x86 architecture.

Throughout this document, references to *legacy mode* refer to all three submodes—*protected mode*, *virtual-8086 mode*, and *real mode*. If a function is specific to either of these submodes, then the name of the specific submode is used instead of the name *legacy mode*. 

2 Memory Model

This chapter describes the memory characteristics that apply to application software in the various operating modes of the AMD64 architecture. These characteristics apply to all instructions in the architecture. Several additional system-level details about memory and cache management are described in Volume 2.

2.1 Memory Organization

2.1.1 Virtual Memory

Virtual memory consists of the entire address space available to programs. It is a large linear-address space that is translated by a combination of hardware and operating-system software to a smaller physical-address space, parts of which are located in memory and parts on disk or other external storage media.

Figure 2-1 on page 12 shows how the virtual-memory space is treated in the two submodes of long mode:

- **64-bit mode**—This mode uses a flat segmentation model of virtual memory. The 64-bit virtual-memory space is treated as a single, flat (unsegmented) address space. Program addresses access locations that can be anywhere in the linear 64-bit address space. The operating system can use separate selectors for code, stack, and data segments for memory-protection purposes, but the base address of all these segments is always 0. (For an exception to this general rule, see “FS and GS as Base of Address Calculation” on page 20.)

- **Compatibility mode**—This mode uses a protected, multi-segment model of virtual memory, just as in legacy protected mode. The 32-bit virtual-memory space is treated as a segmented set of address spaces for code, stack, and data segments, each with its own base address and protection parameters. A segmented space is specified by adding a segment selector to an address.
Segmented memory has been used as a method by which operating systems could isolate programs, and the data used by programs, from each other in an effort to increase the reliability of systems running multiple programs simultaneously. However, most modern operating systems do not use the segmentation features available in the legacy x86 architecture. Instead, these operating systems handle segmentation functions entirely in software. For this reason, the AMD64 architecture dispenses with most of the legacy segmentation functions in 64-bit mode. This allows new 64-bit operating systems to be coded more simply, and it supports more efficient management of multi-programming environments than is possible in the legacy x86 architecture.

2.1.2 Segment Registers

Segment registers hold the selectors used to access memory segments. Figure 2-2 on page 13 shows the application-visible portion of the segment registers. In legacy and compatibility modes, all segment registers are accessible to software. In 64-bit mode, only the CS, FS, and GS segments are recognized by
the processor, and software can use the FS and GS segment-base registers as base registers for address calculation, as described in “FS and GS as Base of Address Calculation” on page 20. For references to the DS, ES, or SS segments in 64-bit mode, the processor assumes that the base for each of these segments is zero, neither their segment limit nor attributes are checked, and the processor simply checks that all such addresses are in canonical form, as described in “64-bit Canonical Addresses” on page 18.

![Segment Registers](image.png)

**Figure 2-2. Segment Registers**

For details on segmentation and the segment registers, see “Segmented Virtual Memory” in Volume 2.

### 2.1.3 Physical Memory

Physical memory is the installed memory (excluding cache memory) in a particular computer system that can be accessed through the processor’s bus interface. The maximum size of the physical memory space is determined by the number of address bits on the bus interface. In a virtual-memory system, the large virtual-address space (also called linear-address space) is translated to a smaller physical-address space by a combination of segmentation and paging hardware and software.

Segmentation is illustrated in Figure 2-1 on page 12. Paging is a mechanism for translating linear (virtual) addresses into fixed-size blocks called pages, which the operating system can move, as needed, between memory and external storage media.
(typically disk). The AMD64 architecture supports an expanded version of the legacy x86 paging mechanism, one that is able to translate the full 64-bit virtual-address space into the physical-address space supported by the particular implementation.

2.1.4 Memory Management

Memory management consists of the methods by which addresses generated by programs are translated via segmentation and/or paging into addresses in physical memory. Memory management is not visible to application programs. It is handled by the operating system and processor hardware. The following description gives a very brief overview of these functions. Details are given in “System-Management Instructions” in Volume 2.

Long-Mode Memory Management. Figure 2-3 shows the flow, from top to bottom, of memory management functions performed in the two submodes of long mode.

![Diagram of Memory Management](image)

**Figure 2-3. Long-Mode Memory Management**

In 64-bit mode, programs generate virtual (linear) addresses that can be up to 64 bits in size. The virtual addresses are
passed to the long-mode paging function, which generates physical addresses that can be up to 52 bits in size. (Specific implementations of the architecture can support fewer virtual-address and physical-address sizes.)

In compatibility mode, legacy 16-bit and 32-bit applications run using legacy x86 protected-mode segmentation semantics. The 16-bit or 32-bit effective addresses generated by programs are combined with their segments to produce 32-bit virtual (linear) addresses that are zero-extended to a maximum of 64 bits. The paging that follows is the same long-mode paging function used in 64-bit mode. It translates the virtual addresses into physical addresses. The combination of segment selector and effective address is also called a logical address or far pointer. The virtual address is also called the linear address.

**Legacy-Mode Memory Management.** Figure 2-4 shows the memory-management functions performed in the three submodes of legacy mode.

---

**Figure 2-4. Legacy-Mode Memory Management**
The memory-management functions differ, depending on the submode, as follows:

- **Protected Mode**—Protected mode supports 16-bit and 32-bit programs with table-based memory segmentation, paging, and privilege-checking. The segmentation function takes 32-bit effective addresses and 16-bit segment selectors and produces 32-bit linear addresses into one of 16K memory segments, each of which can be up to 4GB in size. Paging is optional. The 32-bit physical addresses are either produced by the paging function or the linear addresses are used without modification as physical addresses.

- **Virtual-8086 Mode**—Virtual-8086 mode supports 16-bit programs running as tasks under protected mode. 20-bit linear addresses are formed in the same way as in real mode, but they can optionally be translated through the paging function to form 32-bit physical addresses that access up to 4GB of memory space.

- **Real Mode**—Real mode supports 16-bit programs using register-based shift-and-add segmentation, but it does not support paging. Sixteen-bit effective addresses are zero-extended and added to a 16-bit segment-base address that is left-shifted four bits, producing a 20-bit linear address. The linear address is zero-extended to a 32-bit physical address that can access up to 1MB of memory space.

## 2.2 Memory Addressing

### 2.2.1 Byte Ordering

Instructions and data are stored in memory in *little-endian* byte order. Little-endian ordering places the least-significant byte of the instruction or data item at the lowest memory address and the most-significant byte at the highest memory address.

Figure 2-5 on page 17 shows a generalization of little-endian memory and register images of a quadword data type. The least-significant byte is at the lowest address in memory and at the right-most byte location of the register image.
Figure 2-5.  Byte Ordering

Figure 2-6 on page 18 shows the memory image of a 10-byte instruction. Instructions are byte data types. They are read from memory one byte at a time, starting with the least-significant byte (lowest address). For example, the following instruction specifies the 64-bit instruction MOV RAX, 1122334455667788 instruction that consists of the following ten bytes:

48 B8 8877665544332211

48 is a REX instruction prefix that specifies a 64-bit operand size, B8 is the opcode that—together with the REX prefix—specifies the 64-bit RAX destination register, and 8877665544332211 is the 8-byte immediate value to be moved, where 88 represents the eighth (least-significant) byte and 11 represents the first (most-significant) byte. In memory, the REX prefix byte (48) would be stored at the lowest address, and the first immediate byte (11) would be stored at the highest instruction address.
2.2.2 64-bit Canonical Addresses

Long mode defines 64 bits of virtual address, but implementations of the AMD64 architecture may support fewer bits of virtual address. Although implementations might not use all 64 bits of the virtual address, they 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 said to be in canonical address form. If a virtual-memory reference is not in canonical form, the implementation causes a general-protection exception or stack fault.

2.2.3 Effective Addresses

Programs provide effective addresses to the hardware prior to segmentation and paging translations. Long-mode effective addresses are a maximum of 64 bits wide, as shown in Figure 2-3 on page 14. Programs running in compatibility mode generate (by default) 32-bit effective addresses, which the hardware zero-extends to 64 bits. Legacy-mode effective addresses, with no address-size override, are 32 or 16 bits wide, as shown in Figure 2-4. These sizes can be overridden with an address-size instruction prefix, as described in “Instruction Prefixes” on page 87.

There are five methods for generating effective addresses, depending on the specific instruction encoding:
- **Absolute Addresses**—These addresses are given as displacements (or offsets) from the base address of a data segment. They point directly to a memory location in the data segment.

- **Instruction-Relative Addresses**—These addresses are given as displacements (or offsets) from the current instruction pointer (IP), also called the program counter (PC). They are generated by control-transfer instructions. A displacement in the instruction encoding, or one read from memory, serves as an offset from the address that follows the transfer. See “RIP-Relative Addressing” on page 22 for details about RIP-relative addressing in 64-bit mode.

- **ModR/M Addressing**—These addresses are calculated using a scale, index, base, and displacement. Instruction encodings contain two bytes—MODR/M and optional SIB (scale, index, base) and a variable length displacement—that specify the variables for the calculation. The base and index values are contained in general-purpose registers specified by the SIB byte. The scale and displacement values are specified directly in the instruction encoding. Figure 2-7 shows the components of a complex-address calculation. The resultant effective address is added to the data-segment base address to form a linear address, as described in “Segmented Virtual Memory” in Volume 2. “Instruction Formats” in Volume 3 gives further details on specifying this form of address. The encoding of instructions specifies how the address is calculated.

---

**Figure 2-7. Complex Address Calculation (Protected Mode)**
- **Stack Addresses**—PUSH, POP, CALL, RET, IRET, and INT instructions implicitly use the stack pointer, which contains the address of the procedure stack. See “Stack Operation” on page 23 for details about the size of the stack pointer.

- **String Addresses**—String instructions generate sequential addresses using the rDI and rSI registers, as described in “Implicit Uses of GPRs” on page 34.

In 64-bit mode, with no address-size override, the size of effective-address calculations is 64 bits. An effective-address calculation uses 64-bit base and index registers and sign-extends displacements to 64 bits. Due to the flat address space in 64-bit mode, virtual addresses are equal to effective addresses. (For an exception to this general rule, see “FS and GS as Base of Address Calculation” on page 20.)

**Long-Mode Zero-Extension of 16-Bit and 32-Bit Addresses.** In long mode, all 16-bit and 32-bit address calculations are zero-extended 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.

Because of this, 16-bit and 32-bit applications running in compatibility mode can access only the low 4GB of the long-mode virtual-address space. Likewise, a 32-bit address generated in 64-bit mode can access only the low 4GB of the long-mode virtual-address space.

**Displacements and Immediates.** In general, the maximum size of address displacements and immediate operands is 32 bits. They can be 8, 16, or 32 bits in size, depending on the instruction or, for displacements, the effective address size. In 64-bit mode, displacements are sign-extended to 64 bits during use, but their actual size (for value representation) remains a maximum of 32 bits. The same is true for immediates in 64-bit mode, when the operand size is 64 bits. However, support is provided in 64-bit mode for some 64-bit displacement and immediate forms of the MOV instruction.

**FS and GS as Base of Address Calculation.** In 64-bit mode, the FS and GS segment-base registers (unlike the DS, ES, and SS segment-base registers) can be used as non-zero data-segment base registers for address calculations, as described in “Segmented Virtual Memory” in Volume 2. 64-bit mode assumes all other
data-segment registers (DS, ES, and SS) have a base address of 0.

2.2.4 **Address-Size Prefix**

The default address size of an instruction is determined by the default-size (D) bit and long-mode (L) bit in the current code-segment descriptor (for details, see “Segmented Virtual Memory” in Volume 2). Application software can override the default address size in any operating mode by using the 67th address-size instruction prefix byte. The address-size prefix allows mixing 32-bit and 64-bit addresses on an instruction-by-instruction basis.

Table 2-1 shows the effects of using the address-size prefix in all operating modes. In 64-bit mode, the default address size is 64 bits. The address size can be overridden to 32 bits. 16-bit addresses are not supported in 64-bit mode. In compatibility and legacy modes, the address-size prefix works the same as in the legacy x86 architecture.

<table>
<thead>
<tr>
<th>Operating Mode</th>
<th>Default Address Size (Bits)</th>
<th>Effective Address Size (Bits)</th>
<th>Address-Size Prefix (67h)(^1) Required?</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-Bit Mode</td>
<td>64</td>
<td>64</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>32</td>
<td>yes</td>
</tr>
<tr>
<td>Long Mode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compatibility Mode</td>
<td>32</td>
<td>32</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>16</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>Legacy Mode</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Protected, Virtual-8086, or Real Mode)</td>
<td>32</td>
<td>32</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>16</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>16</td>
<td>no</td>
</tr>
</tbody>
</table>

Note:
1. “No’ indicates that the default address size is used.
RIP-relative addressing—that is, addressing relative to the 64-bit instruction pointer (also called program counter)—is available in 64-bit mode. The effective address is formed by adding the displacement to the 64-bit RIP of the next instruction.

In the legacy x86 architecture, addressing relative to the instruction pointer (IP or EIP) is available only in control-transfer instructions. In the 64-bit mode, any instruction that uses ModRM addressing (see “ModRM and SIB Bytes” in Volume 3) can use RIP-relative addressing. The feature is particularly useful for addressing data in position-independent code and for code that addresses global data.

Programs usually have many references to data, especially global data, that are not register-based. To load such a program, the loader typically selects a location for the program in memory and then adjusts the program’s references to global data based on the load location. RIP-relative addressing of data makes this adjustment unnecessary.

Range of RIP-Relative Addressing. Without RIP-relative addressing, instructions encoded with a ModRM byte address memory relative to zero. With RIP-relative addressing, instructions with a ModRM byte can address memory relative to the 64-bit RIP using a signed 32-bit displacement. This provides an offset range of ±2GB from the RIP.

Effect of Address-Size Prefix on RIP-relative Addressing. RIP-relative addressing is enabled by 64-bit mode, not by a 64-bit address-size. Conversely, use of the address-size prefix does not disable RIP-relative addressing. The effect of the address-size prefix is to truncate and zero-extend the computed effective address to 32 bits, like any other addressing mode.

Encoding. For details on instruction encoding of RIP-relative addressing, see in “RIP-Relative Addressing” in Volume 3.

2.3 Pointers

Pointers are variables that contain addresses rather than data. They are used by instructions to reference memory. Instructions access data using near and far pointers. Stack pointers locate the current stack.
2.3.1 Near and Far Pointers

Near pointers contain only an effective address, which is used as an offset into the current segment. Far pointers contain both an effective address and a segment selector that specifies one of several segments. Figure 2-8 illustrates the two types of pointers.

![Diagram of Near and Far Pointers](513-109.eps)

**Figure 2-8. Near and Far Pointers**

In 64-bit mode, the AMD64 architecture supports only the flat-memory model in which there is only one data segment, so the effective address is used as the virtual (linear) address and far pointers are not needed. In compatibility mode and legacy protected mode, the AMD64 architecture supports multiple memory segments, so effective addresses can be combined with segment selectors to form far pointers, and the terms *logical address* (segment selector and effective address) and *far pointer* are synonyms. Near pointers can also be used in compatibility mode and legacy mode.

2.4 Stack Operation

A stack is a portion of a stack segment in memory that is used to link procedures. Software conventions typically define stacks using a *stack frame*, which consists of two registers—a *stack-frame base pointer* (rBP) and a *stack pointer* (rSP)—as shown in Figure 2-9 on page 24. These stack pointers can be either near pointers or far pointers.

The stack-segment (SS) register, points to the base address of the current stack segment. The stack pointers contain offsets from the base address of the current stack segment. All instructions that address memory using the rBP or rSP registers cause the processor to access the current stack segment.
Figure 2-9. Stack Pointer Mechanism

In typical APIs, the stack-frame base pointer and the stack pointer point to the same location before a procedure call (the top-of-stack of the prior stack frame). After data is pushed onto the stack, the stack-frame base pointer remains where it was and the stack pointer advances downward to the address below the pushed data, where it becomes the new top-of-stack.

In legacy and compatibility modes, the default stack pointer size is 16 bits (SP) or 32 bits (ESP), depending on the default-size (B) bit in the stack-segment descriptor, and multiple stacks can be maintained in separate stack segments. In 64-bit mode, stack pointers are always 64 bits wide (RSP).

Further application-programming details on the stack mechanism are described in “Control Transfers” on page 94. System-programming details on the stack segments are described in “Segmented Virtual Memory” in Volume 2.

### 2.5 Instruction Pointer

The instruction pointer is used in conjunction with the code-segment (CS) register to locate the next instruction in memory. The instruction-pointer register contains the displacement (offset)—from the base address of the current CS segment, or from address 0 in 64-bit mode—to the next instruction to be executed. The pointer is incremented sequentially, except for branch instructions, as described in “Control Transfers” on page 94.
In legacy and compatibility modes, the instruction pointer is a 16-bit (IP) or 32-bit (EIP) register. In 64-bit mode, the instruction pointer is extended to a 64-bit (RIP) register to support 64-bit offsets. The case-sensitive acronym, rIP, is used to refer to any of these three instruction-pointer sizes, depending on the software context.

Figure 2-10 shows the relationship between RIP, EIP, and IP. The 64-bit RIP can be used for RIP-relative addressing, as described in “RIP-Relative Addressing” on page 22.

![Figure 2-10. Instruction Pointer (rIP) Register](513-140.ps)

The contents of the rIP are not directly readable by software. However, the rIP is pushed onto the stack by a call instruction.

The memory model described in this chapter is used by all of the programming environments that make up the AMD64 architecture. The next four chapters of this volume describe the application programming environments, which include:
- General-purpose programming (Chapter 3 on page 27).
- 128-bit media programming (Chapter 4 on page 131).
- 64-bit media programming (Chapter 5 on page 237).
- x87 floating-point programming (Chapter 6 on page 293).