

This chapter describes the basic execution environment of an Intel 64 or IA-32 processor as seen by assembly-language programmers. It describes how the processor executes instructions and how it stores and manipulates data. The execution environment described here includes memory (the address space), general-purpose data registers, segment registers, the flag register, and the instruction pointer register.

3.1 MODES OF OPERATION

The IA-32 architecture supports three basic operating modes: protected mode, real-address mode, and system management mode. The operating mode determines which instructions and architectural features are accessible:

- **Protected mode** — This mode is the native state of the processor. Among the capabilities of protected mode is the ability to directly execute “real-address mode” 8086 software in a protected, multi-tasking environment. This feature is called **virtual-8086 mode**, although it is not actually a processor mode. Virtual-8086 mode is actually a protected mode attribute that can be enabled for any task.
- **Real-address mode** — This mode implements the programming environment of the Intel 8086 processor with extensions (such as the ability to switch to protected or system management mode). The processor is placed in real-address mode following power-up or a reset.
- **System management mode (SMM)** — This mode provides an operating system or executive with a transparent mechanism for implementing platform-specific functions such as power management and system security. The processor enters SMM when the external SMM interrupt pin (SMI#) is activated or an SMI is received from the advanced programmable interrupt controller (APIC).

In SMM, the processor switches to a separate address space while saving the basic context of the currently running program or task. SMM-specific code may then be executed transparently. Upon returning from SMM, the processor is placed back into its state prior to the system management interrupt. SMM was introduced with the Intel386™ SL and Intel486™ SL processors and became a standard IA-32 feature with the Pentium processor family.

3.1.1 Intel® 64 Architecture

Intel 64 architecture adds IA-32e mode. IA-32e mode has two sub-modes. These are:

- **Compatibility mode (sub-mode of IA-32e mode)** — Compatibility mode permits most legacy 16-bit and 32-bit applications to run without re-compilation under a 64-bit operating system. For brevity, the compatibility sub-mode is referred to as compatibility mode in IA-32 architecture. The execution environment of compatibility mode is the same as described in Section 3.2. Compatibility mode also supports all of the privilege levels that are supported in 64-bit and protected modes. Legacy applications that run in Virtual 8086 mode or use hardware task management will not work in this mode.

Compatibility mode is enabled by the operating system (OS) on a code segment basis. This means that a single 64-bit OS can support 64-bit applications running in 64-bit mode and support legacy 32-bit applications (not recompiled for 64-bits) running in compatibility mode.

Compatibility mode is similar to 32-bit protected mode. Applications access only the first 4 GByte of linear-address space. Compatibility mode uses 16-bit and 32-bit address and operand sizes. Like protected mode, this mode allows applications to access physical memory greater than 4 GByte using PAE (Physical Address Extensions).

- **64-bit mode (sub-mode of IA-32e mode)** — This mode enables a 64-bit operating system to run applications written to access 64-bit linear address space. For brevity, the 64-bit sub-mode is referred to as 64-bit mode in IA-32 architecture.

64-bit mode extends the number of general purpose registers and SIMD extension registers from 8 to 16. General purpose registers are widened to 64 bits. The mode also introduces a new opcode prefix (REX) to access the register extensions. See Section 3.2.1 for a detailed description.

64-bit mode is enabled by the operating system on a code-segment basis. Its default address size is 64 bits and its default operand size is 32 bits. The default operand size can be overridden on an instruction-by-instruction basis using a REX opcode prefix in conjunction with an operand size override prefix.

REX prefixes allow a 64-bit operand to be specified when operating in 64-bit mode. By using this mechanism, many existing instructions have been promoted to allow the use of 64-bit registers and 64-bit addresses.

3.2 OVERVIEW OF THE BASIC EXECUTION ENVIRONMENT

Any program or task running on an IA-32 processor is given a set of resources for executing instructions and for storing code, data, and state information. These resources (described briefly in the following paragraphs and shown in Figure 3-1) make up the basic execution environment for an IA-32 processor.

An Intel 64 processor supports the basic execution environment of an IA-32 processor, and a similar environment under IA-32e mode that can execute 64-bit programs (64-bit sub-mode) and 32-bit programs (compatibility sub-mode).

The basic execution environment is used jointly by the application programs and the operating system or executive running on the processor.

- **Address space** — Any task or program running on an IA-32 processor can address a linear address space of up to 4 GBytes (2^{32} bytes) and a physical address space of up to 64 GBytes (2^{36} bytes). See Section 3.3.6, “Extended Physical Addressing in Protected Mode,” for more information about addressing an address space greater than 4 GBytes.
- **Basic program execution registers** — The eight general-purpose registers, the six segment registers, the EFLAGS register, and the EIP (instruction pointer) register comprise a basic execution environment in which to execute a set of general-purpose instructions. These instructions perform basic integer arithmetic on byte, word, and doubleword integers, handle program flow control, operate on bit and byte strings, and address memory. See Section 3.4, “Basic Program Execution Registers,” for more information about these registers.
- **x87 FPU registers** — The eight x87 FPU data registers, the x87 FPU control register, the status register, the x87 FPU instruction pointer register, the x87 FPU operand (data) pointer register, the x87 FPU tag register, and the x87 FPU opcode register provide an execution environment for operating on single-precision, double-precision, and double extended-precision floating-point values, word integers, doubleword integers, quadword integers, and binary coded decimal (BCD) values. See Section 8.1, “x87 FPU Execution Environment,” for more information about these registers.
- **MMX registers** — The eight MMX registers support execution of single-instruction, multiple-data (SIMD) operations on 64-bit packed byte, word, and doubleword integers. See Section 9.2, “The MMX Technology Programming Environment,” for more information about these registers.
- **XMM registers** — The eight XMM data registers and the MXCSR register support execution of SIMD operations on 128-bit packed single-precision and double-precision floating-point values and on 128-bit packed byte, word, doubleword, and quadword integers. See Section 10.2, “SSE Programming Environment,” for more information about these registers.

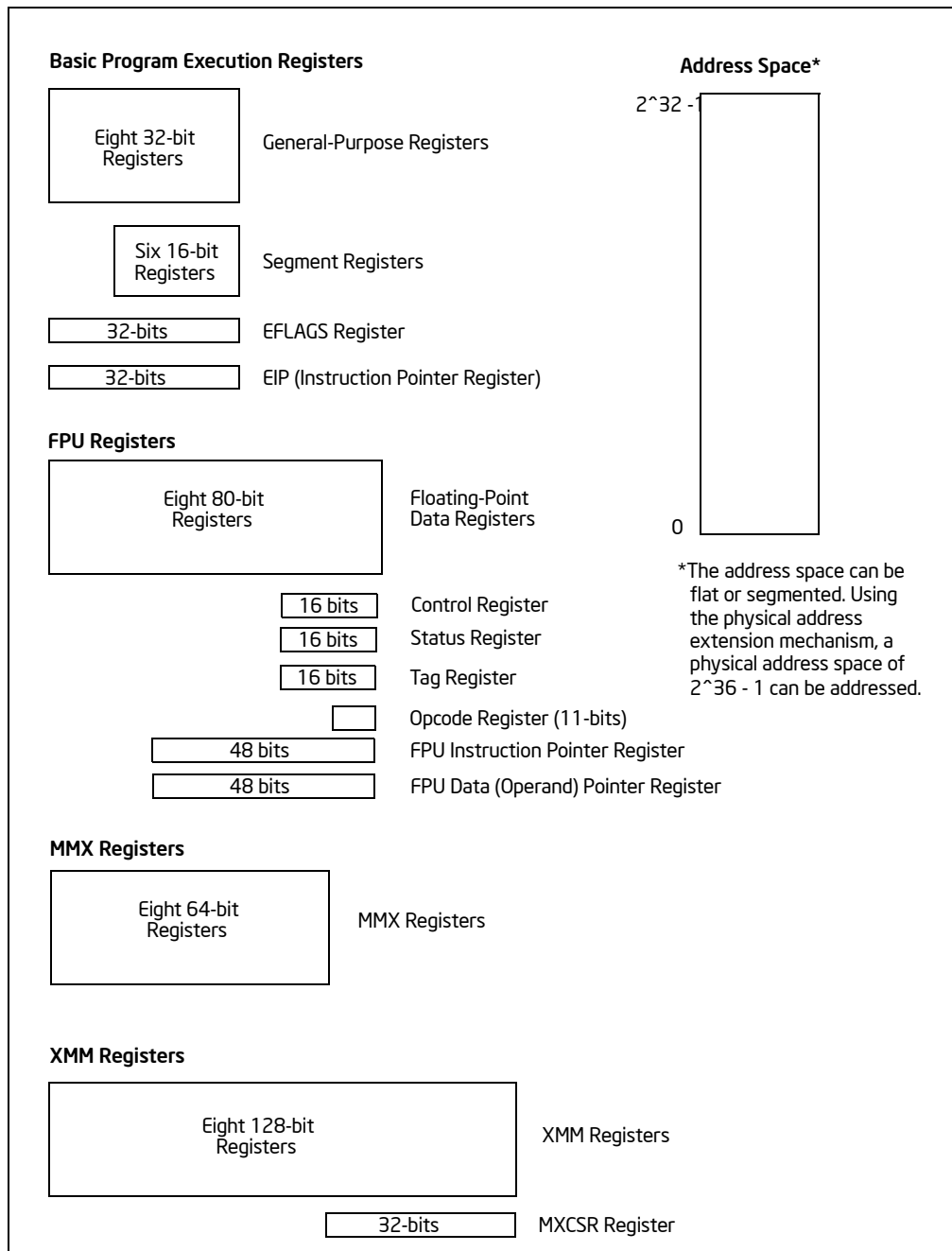


Figure 3-1. IA-32 Basic Execution Environment for Non-64-bit Modes

- **Stack** — To support procedure or subroutine calls and the passing of parameters between procedures or subroutines, a stack and stack management resources are included in the execution environment. The stack (not shown in Figure 3-1) is located in memory. See Section 6.2, “Stacks,” for more information about stack structure.

In addition to the resources provided in the basic execution environment, the IA-32 architecture provides the following resources as part of its system-level architecture. They provide extensive support for operating-system and system-development software. Except for the I/O ports, the system resources are described in detail in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 3A & 3B.

- **I/O ports** — The IA-32 architecture supports a transfers of data to and from input/output (I/O) ports. See Chapter 16, “Input/Output,” in this volume.

- **Control registers** — The five control registers (CR0 through CR4) determine the operating mode of the processor and the characteristics of the currently executing task. See Chapter 2, “System Architecture Overview,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.
- **Memory management registers** — The GDTR, IDTR, task register, and LDTR specify the locations of data structures used in protected mode memory management. See Chapter 2, “System Architecture Overview,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.
- **Debug registers** — The debug registers (DR0 through DR7) control and allow monitoring of the processor’s debugging operations. See in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3B.
- **Memory type range registers (MTRRs)** — The MTRRs are used to assign memory types to regions of memory. See the sections on MTRRs in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 3A & 3B.
- **Machine specific registers (MSRs)** — The processor provides a variety of machine specific registers that are used to control and report on processor performance. Virtually all MSRs handle system related functions and are not accessible to an application program. One exception to this rule is the time-stamp counter. The MSRs are described in Chapter 35, “Model-Specific Registers (MSRs),” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C.
- **Machine check registers** — The machine check registers consist of a set of control, status, and error-reporting MSRs that are used to detect and report on hardware (machine) errors. See Chapter 15, “Machine-Check Architecture,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.
- **Performance monitoring counters** — The performance monitoring counters allow processor performance events to be monitored. See Chapter 23, “Introduction to Virtual-Machine Extensions,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C.

The remainder of this chapter describes the organization of memory and the address space, the basic program execution registers, and addressing modes. Refer to the following chapters in this volume for descriptions of the other program execution resources shown in Figure 3-1:

- **x87 FPU registers** — See Chapter 8, “Programming with the x87 FPU.”
- **MMX Registers** — See Chapter 9, “Programming with Intel® MMX™ Technology.”
- **XMM registers** — See Chapter 10, “Programming with Streaming SIMD Extensions (SSE),” Chapter 11, “Programming with Streaming SIMD Extensions 2 (SSE2),” and Chapter 12, “Programming with SSE3, SSSE3, SSE4 and AESNI.”
- **Stack implementation and procedure calls** — See Chapter 6, “Procedure Calls, Interrupts, and Exceptions.”

3.2.1 64-Bit Mode Execution Environment

The execution environment for 64-bit mode is similar to that described in Section 3.2. The following paragraphs describe the differences that apply.

- **Address space** — A task or program running in 64-bit mode on an IA-32 processor can address linear address space of up to 2^{64} bytes (subject to the canonical addressing requirement described in Section 3.3.7.1) and physical address space of up to 2^{46} bytes. Software can query CPUID for the physical address size supported by a processor.
- **Basic program execution registers** — The number of general-purpose registers (GPRs) available is 16. GPRs are 64-bits wide and they support operations on byte, word, doubleword and quadword integers. Accessing byte registers is done uniformly to the lowest 8 bits. The instruction pointer register becomes 64 bits. The EFLAGS register is extended to 64 bits wide, and is referred to as the RFLAGS register. The upper 32 bits of RFLAGS is reserved. The lower 32 bits of RFLAGS is the same as EFLAGS. See Figure 3-2.
- **XMM registers** — There are 16 XMM data registers for SIMD operations. See Section 10.2, “SSE Programming Environment,” for more information about these registers.
- **Stack** — The stack pointer size is 64 bits. Stack size is not controlled by a bit in the SS descriptor (as it is in non-64-bit modes) nor can the pointer size be overridden by an instruction prefix.
- **Control registers** — Control registers expand to 64 bits. A new control register (the task priority register: CR8 or TPR) has been added. See Chapter 2, “Intel® 64 and IA-32 Architectures,” in this volume.

- **Debug registers** — Debug registers expand to 64 bits. See Chapter 17, “Debugging, Branch Profiles and Time-Stamp Counter,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.
- **Descriptor table registers** — The global descriptor table register (GDTR) and interrupt descriptor table register (IDTR) expand to 10 bytes so that they can hold a full 64-bit base address. The local descriptor table register (LDTR) and the task register (TR) also expand to hold a full 64-bit base address.

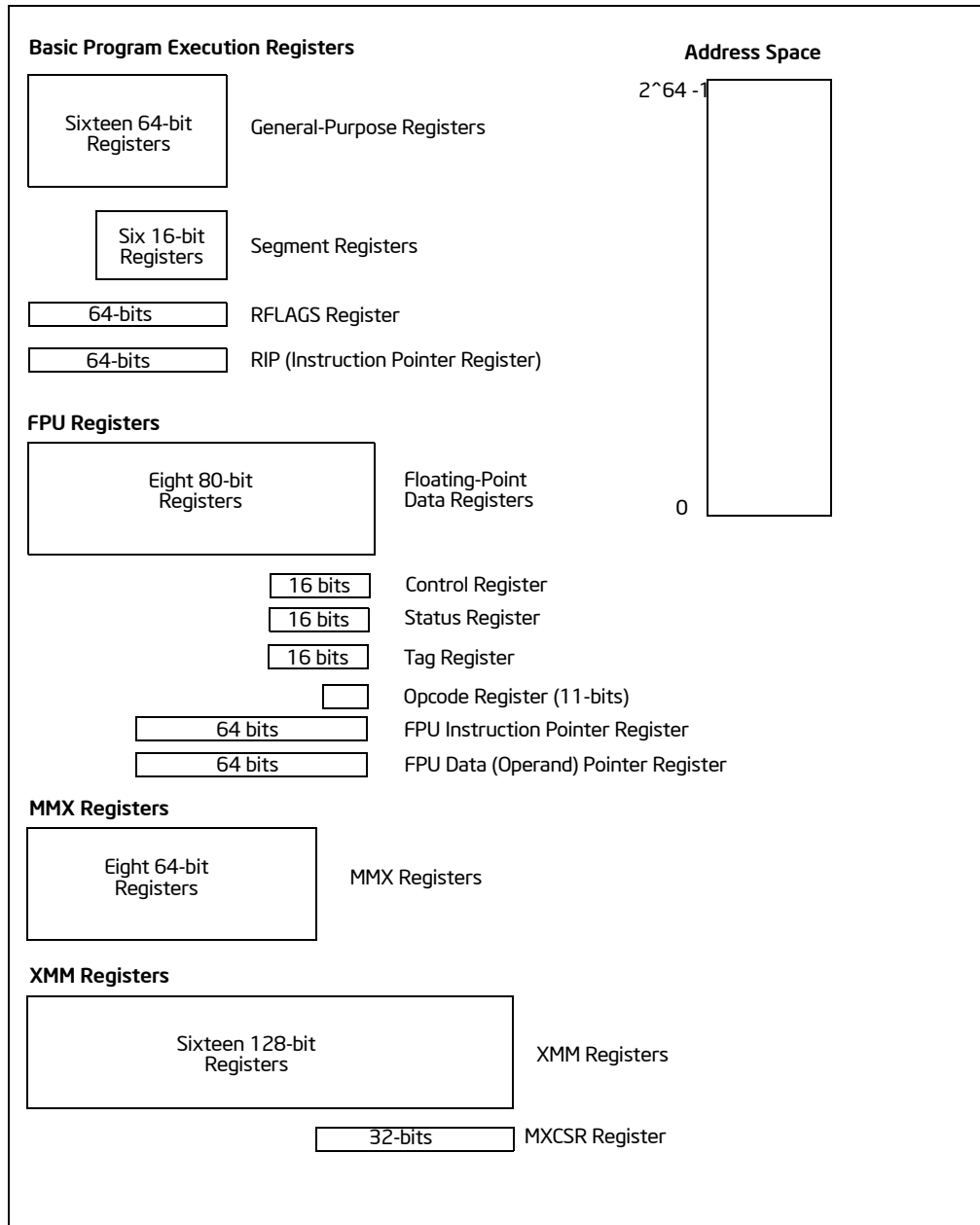


Figure 3-2. 64-Bit Mode Execution Environment

3.3 MEMORY ORGANIZATION

The memory that the processor addresses on its bus is called **physical memory**. Physical memory is organized as a sequence of 8-bit bytes. Each byte is assigned a unique address, called a **physical address**. The **physical address space** ranges from zero to a maximum of $2^{36} - 1$ (64 GBytes) if the processor does not support Intel

64 architecture. Intel 64 architecture introduces a changes in physical and linear address space; these are described in Section 3.3.3, Section 3.3.4, and Section 3.3.7.

Virtually any operating system or executive designed to work with an IA-32 or Intel 64 processor will use the processor's memory management facilities to access memory. These facilities provide features such as segmentation and paging, which allow memory to be managed efficiently and reliably. Memory management is described in detail in Chapter 3, "Protected-Mode Memory Management," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A. The following paragraphs describe the basic methods of addressing memory when memory management is used.

3.3.1 IA-32 Memory Models

When employing the processor's memory management facilities, programs do not directly address physical memory. Instead, they access memory using one of three memory models: flat, segmented, or real address mode:

- **Flat memory model** — Memory appears to a program as a single, continuous address space (Figure 3-3). This space is called a **linear address space**. Code, data, and stacks are all contained in this address space. Linear address space is byte addressable, with addresses running contiguously from 0 to $2^{32} - 1$ (if not in 64-bit mode). An address for any byte in linear address space is called a **linear address**.
- **Segmented memory model** — Memory appears to a program as a group of independent address spaces called segments. Code, data, and stacks are typically contained in separate segments. To address a byte in a segment, a program issues a logical address. This consists of a segment selector and an offset (logical addresses are often referred to as far pointers). The segment selector identifies the segment to be accessed and the offset identifies a byte in the address space of the segment. Programs running on an IA-32 processor can address up to 16,383 segments of different sizes and types, and each segment can be as large as 2^{32} bytes.

Internally, all the segments that are defined for a system are mapped into the processor's linear address space. To access a memory location, the processor thus translates each logical address into a linear address. This translation is transparent to the application program.

The primary reason for using segmented memory is to increase the reliability of programs and systems. For example, placing a program's stack in a separate segment prevents the stack from growing into the code or data space and overwriting instructions or data, respectively.

- **Real-address mode memory model** — This is the memory model for the Intel 8086 processor. It is supported to provide compatibility with existing programs written to run on the Intel 8086 processor. The real-address mode uses a specific implementation of segmented memory in which the linear address space for the program and the operating system/executive consists of an array of segments of up to 64 KBytes in size each. The maximum size of the linear address space in real-address mode is 2^{20} bytes.

See also: Chapter 20, "8086 Emulation," Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B.

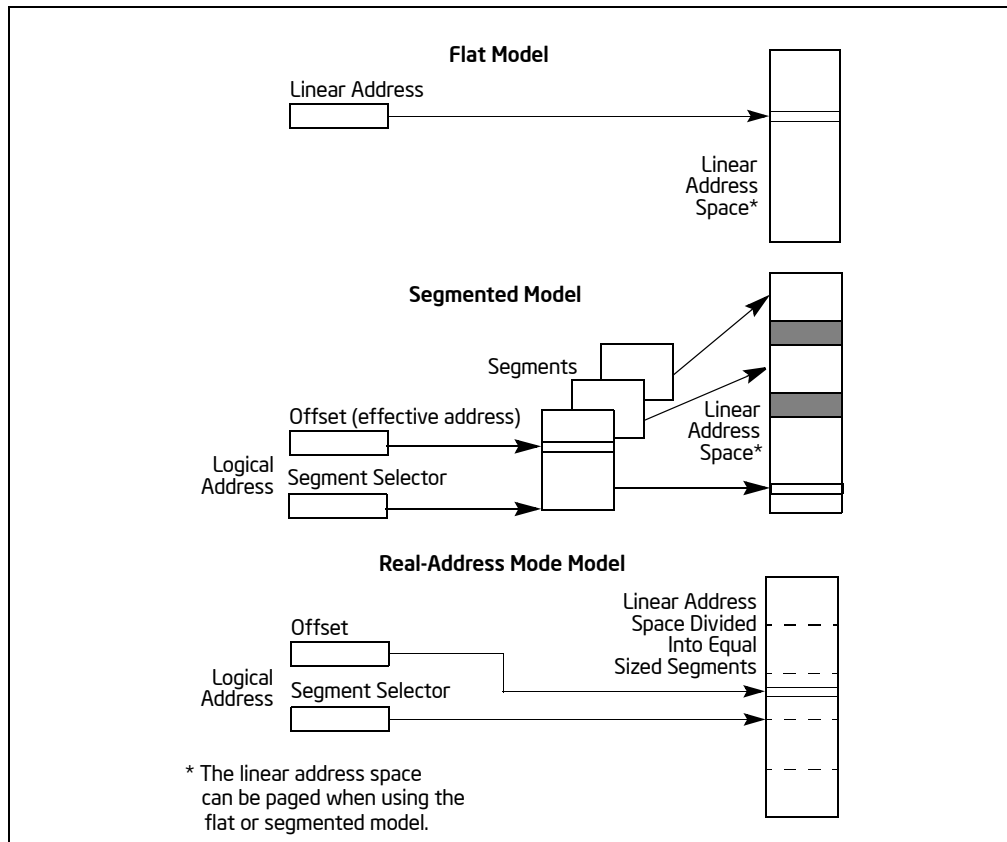


Figure 3-3. Three Memory Management Models

3.3.2 Paging and Virtual Memory

With the flat or the segmented memory model, linear address space is mapped into the processor's physical address space either directly or through paging. When using direct mapping (paging disabled), each linear address has a one-to-one correspondence with a physical address. Linear addresses are sent out on the processor's address lines without translation.

When using the IA-32 architecture's paging mechanism (paging enabled), linear address space is divided into pages which are mapped to virtual memory. The pages of virtual memory are then mapped as needed into physical memory. When an operating system or executive uses paging, the paging mechanism is transparent to an application program. All that the application sees is linear address space.

In addition, IA-32 architecture's paging mechanism includes extensions that support:

- Physical Address Extensions (PAE) to address physical address space greater than 4 GBytes.
- Page Size Extensions (PSE) to map linear address to physical address in 4-MBytes pages.

See also: Chapter 3, "Protected-Mode Memory Management," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

3.3.3 Memory Organization in 64-Bit Mode

Intel 64 architecture supports physical address space greater than 64 GBytes; the actual physical address size of IA-32 processors is implementation specific. In 64-bit mode, there is architectural support for 64-bit linear address space. However, processors supporting Intel 64 architecture may implement less than 64-bits (see Section 3.3.7.1). The linear address space is mapped into the processor physical address space through the PAE paging mechanism.

3.3.4 Modes of Operation vs. Memory Model

When writing code for an IA-32 or Intel 64 processor, a programmer needs to know the operating mode the processor is going to be in when executing the code and the memory model being used. The relationship between operating modes and memory models is as follows:

- **Protected mode** — When in protected mode, the processor can use any of the memory models described in this section. (The real-addressing mode memory model is ordinarily used only when the processor is in the virtual-8086 mode.) The memory model used depends on the design of the operating system or executive. When multitasking is implemented, individual tasks can use different memory models.
- **Real-address mode** — When in real-address mode, the processor only supports the real-address mode memory model.
- **System management mode** — When in SMM, the processor switches to a separate address space, called the system management RAM (SMRAM). The memory model used to address bytes in this address space is similar to the real-address mode model. See Chapter 34, “System Management Mode,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3C, for more information on the memory model used in SMM.
- **Compatibility mode** — Software that needs to run in compatibility mode should observe the same memory model as those targeted to run in 32-bit protected mode. The effect of segmentation is the same as it is in 32-bit protected mode semantics.
- **64-bit mode** — Segmentation is generally (but not completely) disabled, creating a flat 64-bit linear-address space. Specifically, the processor treats the segment base of CS, DS, ES, and SS as zero in 64-bit mode (this makes a linear address equal an effective address). Segmented and real address modes are not available in 64-bit mode.

3.3.5 32-Bit and 16-Bit Address and Operand Sizes

IA-32 processors in protected mode can be configured for 32-bit or 16-bit address and operand sizes. With 32-bit address and operand sizes, the maximum linear address or segment offset is FFFFFFFFH ($2^{32}-1$); operand sizes are typically 8 bits or 32 bits. With 16-bit address and operand sizes, the maximum linear address or segment offset is FFFFH ($2^{16}-1$); operand sizes are typically 8 bits or 16 bits.

When using 32-bit addressing, a logical address (or far pointer) consists of a 16-bit segment selector and a 32-bit offset; when using 16-bit addressing, an address consists of a 16-bit segment selector and a 16-bit offset.

Instruction prefixes allow temporary overrides of the default address and/or operand sizes from within a program.

When operating in protected mode, the segment descriptor for the currently executing code segment defines the default address and operand size. A segment descriptor is a system data structure not normally visible to application code. Assembler directives allow the default addressing and operand size to be chosen for a program. The assembler and other tools then set up the segment descriptor for the code segment appropriately.

When operating in real-address mode, the default addressing and operand size is 16 bits. An address-size override can be used in real-address mode to enable 32-bit addressing. However, the maximum allowable 32-bit linear address is still 000FFFFFFH ($2^{20}-1$).

3.3.6 Extended Physical Addressing in Protected Mode

Beginning with P6 family processors, the IA-32 architecture supports addressing of up to 64 GBytes (2^{36} bytes) of physical memory. A program or task could not address locations in this address space directly. Instead, it addresses individual linear address spaces of up to 4 GBytes that mapped to 64-GByte physical address space through a virtual memory management mechanism. Using this mechanism, an operating system can enable a program to switch 4-GByte linear address spaces within 64-GByte physical address space.

The use of extended physical addressing requires the processor to operate in protected mode and the operating system to provide a virtual memory management system. See “36-Bit Physical Addressing Using the PAE Paging Mechanism” in Chapter 3, “Protected-Mode Memory Management,” of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

3.3.7 Address Calculations in 64-Bit Mode

In most cases, 64-bit mode uses flat address space for code, data, and stacks. In 64-bit mode (if there is no address-size override), the size of effective address calculations is 64 bits. An effective-address calculation uses a 64-bit base and index registers and sign-extend displacements to 64 bits.

In the flat address space of 64-bit mode, linear addresses are equal to effective addresses because the base address is zero. In the event that FS or GS segments are used with a non-zero base, this rule does not hold. In 64-bit mode, the effective address components are added and the effective address is truncated (See for example the instruction LEA) before adding the full 64-bit segment base. The base is never truncated, regardless of addressing mode in 64-bit mode.

The instruction pointer is extended to 64 bits to support 64-bit code offsets. The 64-bit instruction pointer is called the RIP. Table 3-1 shows the relationship between RIP, EIP, and IP.

Table 3-1. Instruction Pointer Sizes

	Bits 63:32	Bits 31:16	Bits 15:0
16-bit instruction pointer	Not Modified		IP
32-bit instruction pointer	Zero Extension	EIP	
64-bit instruction pointer	RIP		

Generally, displacements and immediates 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 64-bit displacement and immediate forms of the MOV instruction.

All 16-bit and 32-bit address calculations are zero-extended in IA-32e 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. Because of this, 16-bit and 32-bit applications running in compatibility mode can access only the low 4 GBytes of the 64-bit mode effective addresses. Likewise, a 32-bit address generated in 64-bit mode can access only the low 4 GBytes of the 64-bit mode effective addresses.

3.3.7.1 Canonical Addressing

In 64-bit mode, an address is considered to be in canonical form if address bits 63 through to the most-significant implemented bit by the microarchitecture are set to either all ones or all zeros.

Intel 64 architecture defines a 64-bit linear address. Implementations can support less. The first implementation of IA-32 processors with Intel 64 architecture supports a 48-bit linear address. This means a canonical address must have bits 63 through 48 set to zeros or ones (depending on whether bit 47 is a zero or one).

Although implementations may not use all 64 bits of the linear address, they should check bits 63 through the most-significant implemented bit to see if the address is in canonical form. If a linear-memory reference is not in canonical form, the implementation should generate an exception. In most cases, a general-protection exception (#GP) is generated. However, in the case of explicit or implied stack references, a stack fault (#SS) is generated.

Instructions that have implied stack references, by default, use the SS segment register. These include PUSH/POP-related instructions and instructions using RSP/RBP as base registers. In these cases, the canonical fault is #SF.

If an instruction uses base registers RSP/RBP and uses a segment override prefix to specify a non-SS segment, a canonical fault generates a #GP (instead of an #SF). In 64-bit mode, only FS and GS segment-overrides are applicable in this situation. Other segment override prefixes (CS, DS, ES and SS) are ignored. Note that this also means that an SS segment-override applied to a “non-stack” register reference is ignored. Such a sequence still produces a #GP for a canonical fault (and not an #SF).

3.4 BASIC PROGRAM EXECUTION REGISTERS

IA-32 architecture provides 16 basic program execution registers for use in general system and application programming (see Figure 3-4). These registers can be grouped as follows:

- **General-purpose registers.** These eight registers are available for storing operands and pointers.
- **Segment registers.** These registers hold up to six segment selectors.
- **EFLAGS (program status and control) register.** The EFLAGS register report on the status of the program being executed and allows limited (application-program level) control of the processor.
- **EIP (instruction pointer) register.** The EIP register contains a 32-bit pointer to the next instruction to be executed.

3.4.1 General-Purpose Registers

The 32-bit general-purpose registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP are provided for holding the following items:

- Operands for logical and arithmetic operations
- Operands for address calculations
- Memory pointers

Although all of these registers are available for general storage of operands, results, and pointers, caution should be used when referencing the ESP register. The ESP register holds the stack pointer and as a general rule should not be used for another purpose.

Many instructions assign specific registers to hold operands. For example, string instructions use the contents of the ECX, ESI, and EDI registers as operands. When using a segmented memory model, some instructions assume that pointers in certain registers are relative to specific segments. For instance, some instructions assume that a pointer in the EBX register points to a memory location in the DS segment.

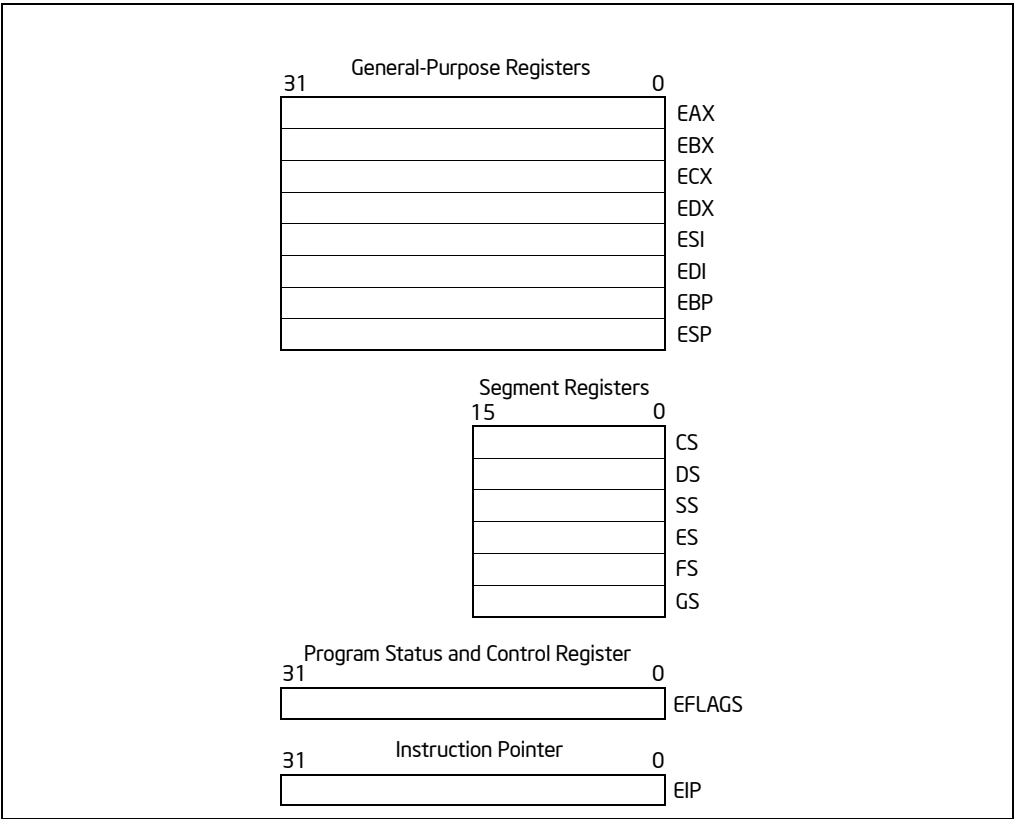


Figure 3-4. General System and Application Programming Registers

The special uses of general-purpose registers by instructions are described in Chapter 5, “Instruction Set Summary,” in this volume. See also: Chapter 3 and Chapter 4 of Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A & 2B. The following is a summary of special uses:

- **EAX** — Accumulator for operands and results data
- **EBX** — Pointer to data in the DS segment
- **ECX** — Counter for string and loop operations
- **EDX** — I/O pointer
- **ESI** — Pointer to data in the segment pointed to by the DS register; source pointer for string operations
- **EDI** — Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations
- **ESP** — Stack pointer (in the SS segment)
- **EBP** — Pointer to data on the stack (in the SS segment)

As shown in Figure 3-5, the lower 16 bits of the general-purpose registers map directly to the register set found in the 8086 and Intel 286 processors and can be referenced with the names AX, BX, CX, DX, BP, SI, DI, and SP. Each of the lower two bytes of the EAX, EBX, ECX, and EDX registers can be referenced by the names AH, BH, CH, and DH (high bytes) and AL, BL, CL, and DL (low bytes).

General-Purpose Registers			16-bit	32-bit
31	16 15	8 7	0	
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

Figure 3-5. Alternate General-Purpose Register Names

3.4.1.1 General-Purpose Registers in 64-Bit Mode

In 64-bit mode, there are 16 general purpose registers and the default operand size is 32 bits. However, general-purpose registers are able to work with either 32-bit or 64-bit operands. If a 32-bit operand size is specified: EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D are available. If a 64-bit operand size is specified: RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8-R15 are available. R8D-R15D/R8-R15 represent eight new general-purpose registers. All of these registers can be accessed at the byte, word, dword, and qword level. REX prefixes are used to generate 64-bit operand sizes or to reference registers R8-R15.

Registers only available in 64-bit mode (R8-R15 and XMM8-XMM15) are preserved across transitions from 64-bit mode into compatibility mode then back into 64-bit mode. However, values of R8-R15 and XMM8-XMM15 are undefined after transitions from 64-bit mode through compatibility mode to legacy or real mode and then back through compatibility mode to 64-bit mode.

Table 3-2. Addressable General Purpose Registers

Register Type	Without REX	With REX
Byte Registers	AL, BL, CL, DL, AH, BH, CH, DH	AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8L - R15L
Word Registers	AX, BX, CX, DX, DI, SI, BP, SP	AX, BX, CX, DX, DI, SI, BP, SP, R8W - R15W
Doubleword Registers	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D - R15D
Quadword Registers	N.A.	RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8 - R15

In 64-bit mode, there are limitations on accessing byte registers. An instruction cannot reference legacy high-bytes (for example: AH, BH, CH, DH) and one of the new byte registers at the same time (for example: the low byte of the RAX register). However, instructions may reference legacy low-bytes (for example: AL, BL, CL or DL) and new byte registers at the same time (for example: the low byte of the R8 register, or RBP). The architecture enforces this limitation by changing high-byte references (AH, BH, CH, DH) to low byte references (BPL, SPL, DIL, SIL: the low 8 bits for RBP, RSP, RDI and RSI) for instructions using a REX prefix.

When in 64-bit mode, operand size determines the number of valid bits in the destination general-purpose register:

- 64-bit operands generate a 64-bit result in the destination general-purpose register.
- 32-bit operands generate a 32-bit result, zero-extended to a 64-bit result in the destination general-purpose register.
- 8-bit and 16-bit operands generate an 8-bit or 16-bit result. The upper 56 bits or 48 bits (respectively) of the destination general-purpose register are not modified by the operation. If the result of an 8-bit or 16-bit operation is intended for 64-bit address calculation, explicitly sign-extend the register to the full 64-bits.

Because the upper 32 bits of 64-bit general-purpose registers are undefined in 32-bit modes, the upper 32 bits of any general-purpose register are not preserved when switching from 64-bit mode to a 32-bit mode (to protected mode or compatibility mode). Software must not depend on these bits to maintain a value after a 64-bit to 32-bit mode switch.

3.4.2 Segment Registers

The segment registers (CS, DS, SS, ES, FS, and GS) hold 16-bit segment selectors. A segment selector is a special pointer that identifies a segment in memory. To access a particular segment in memory, the segment selector for that segment must be present in the appropriate segment register.

When writing application code, programmers generally create segment selectors with assembler directives and symbols. The assembler and other tools then create the actual segment selector values associated with these directives and symbols. If writing system code, programmers may need to create segment selectors directly. See Chapter 3, “Protected-Mode Memory Management,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A.

How segment registers are used depends on the type of memory management model that the operating system or executive is using. When using the flat (unsegmented) memory model, segment registers are loaded with segment selectors that point to overlapping segments, each of which begins at address 0 of the linear address space (see Figure 3-6). These overlapping segments then comprise the linear address space for the program. Typically, two overlapping segments are defined: one for code and another for data and stacks. The CS segment register points to the code segment and all the other segment registers point to the data and stack segment.

When using the segmented memory model, each segment register is ordinarily loaded with a different segment selector so that each segment register points to a different segment within the linear address space (see Figure 3-7). At any time, a program can thus access up to six segments in the linear address space. To access a segment not pointed to by one of the segment registers, a program must first load the segment selector for the segment to be accessed into a segment register.

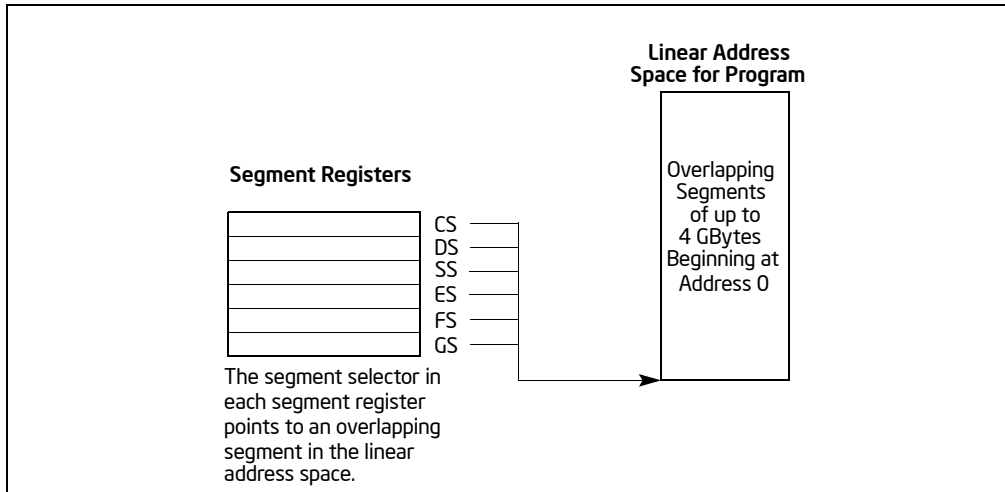


Figure 3-6. Use of Segment Registers for Flat Memory Model

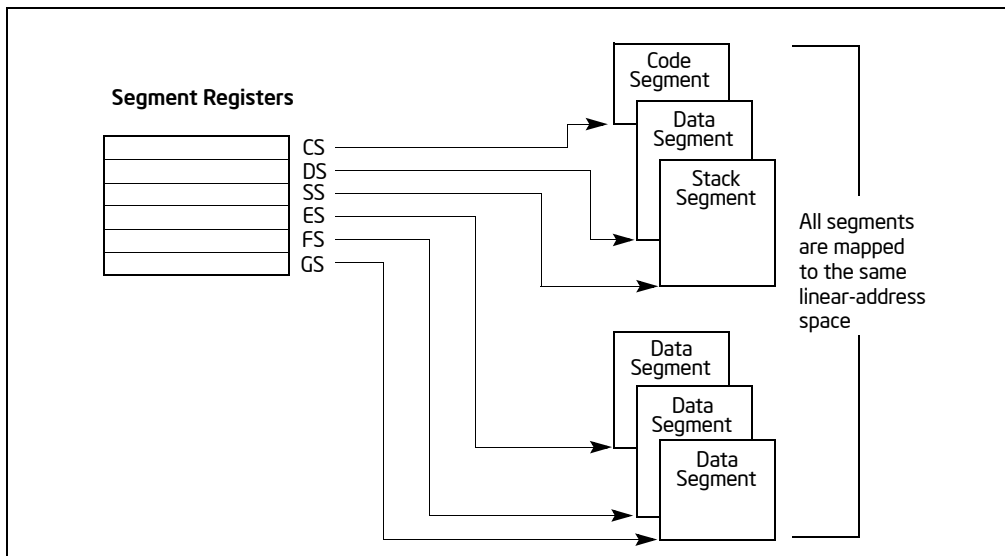


Figure 3-7. Use of Segment Registers in Segmented Memory Model

Each of the segment registers is associated with one of three types of storage: code, data, or stack. For example, the CS register contains the segment selector for the **code segment**, where the instructions being executed are stored. The processor fetches instructions from the code segment, using a logical address that consists of the segment selector in the CS register and the contents of the EIP register. The EIP register contains the offset within the code segment of the next instruction to be executed. The CS register cannot be loaded explicitly by an application program. Instead, it is loaded implicitly by instructions or internal processor operations that change program control (such as, procedure calls, interrupt handling, or task switching).

The DS, ES, FS, and GS registers point to four **data segments**. The availability of four data segments permits efficient and secure access to different types of data structures. For example, four separate data segments might be created: one for the data structures of the current module, another for the data exported from a higher-level module, a third for a dynamically created data structure, and a fourth for data shared with another program. To access additional data segments, the application program must load segment selectors for these segments into the DS, ES, FS, and GS registers, as needed.

The SS register contains the segment selector for the **stack segment**, where the procedure stack is stored for the program, task, or handler currently being executed. All stack operations use the SS register to find the stack

segment. Unlike the CS register, the SS register can be loaded explicitly, which permits application programs to set up multiple stacks and switch among them.

See Section 3.3, “Memory Organization,” for an overview of how the segment registers are used in real-address mode.

The four segment registers CS, DS, SS, and ES are the same as the segment registers found in the Intel 8086 and Intel 286 processors and the FS and GS registers were introduced into the IA-32 Architecture with the Intel386™ family of processors.

3.4.2.1 Segment Registers in 64-Bit Mode

In 64-bit mode: CS, DS, ES, SS are treated as if each segment base is 0, regardless of the value of the associated segment descriptor base. This creates a flat address space for code, data, and stack. FS and GS are exceptions. Both segment registers may be used as additional base registers in linear address calculations (in the addressing of local data and certain operating system data structures).

Even though segmentation is generally disabled, segment register loads may cause the processor to perform segment access assists. During these activities, enabled processors will still perform most of the legacy checks on loaded values (even if the checks are not applicable in 64-bit mode). Such checks are needed because a segment register loaded in 64-bit mode may be used by an application running in compatibility mode.

Limit checks for CS, DS, ES, SS, FS, and GS are disabled in 64-bit mode.

3.4.3 EFLAGS Register

The 32-bit EFLAGS register contains a group of status flags, a control flag, and a group of system flags. Figure 3-8 defines the flags within this register. Following initialization of the processor (either by asserting the RESET pin or the INIT pin), the state of the EFLAGS register is 00000002H. Bits 1, 3, 5, 15, and 22 through 31 of this register are reserved. Software should not use or depend on the states of any of these bits.

Some of the flags in the EFLAGS register can be modified directly, using special-purpose instructions (described in the following sections). There are no instructions that allow the whole register to be examined or modified directly.

The following instructions can be used to move groups of flags to and from the procedure stack or the EAX register: LAHF, SAHF, PUSHF, PUSHFD, POPF, and POPFD. After the contents of the EFLAGS register have been transferred to the procedure stack or EAX register, the flags can be examined and modified using the processor's bit manipulation instructions (BT, BTS, BTR, and BTC).

When suspending a task (using the processor's multitasking facilities), the processor automatically saves the state of the EFLAGS register in the task state segment (TSS) for the task being suspended. When binding itself to a new task, the processor loads the EFLAGS register with data from the new task's TSS.

When a call is made to an interrupt or exception handler procedure, the processor automatically saves the state of the EFLAGS registers on the procedure stack. When an interrupt or exception is handled with a task switch, the state of the EFLAGS register is saved in the TSS for the task being suspended.

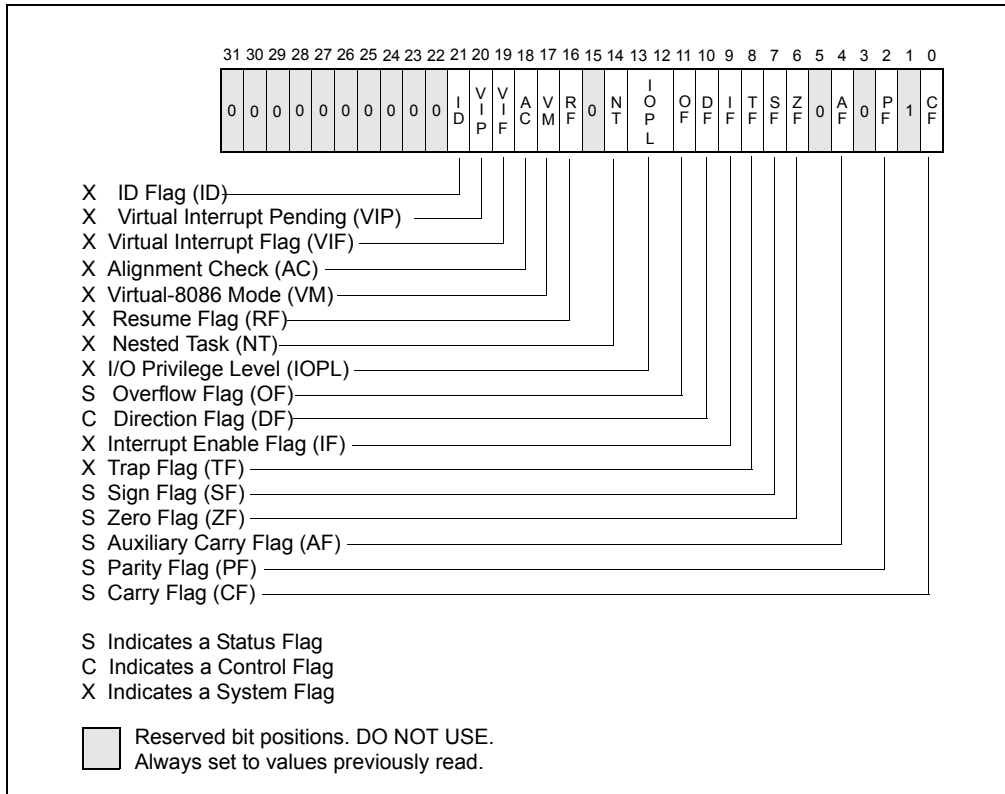


Figure 3-8. EFLAGS Register

As the IA-32 Architecture has evolved, flags have been added to the EFLAGS register, but the function and placement of existing flags have remained the same from one family of the IA-32 processors to the next. As a result, code that accesses or modifies these flags for one family of IA-32 processors works as expected when run on later families of processors.

3.4.3.1 Status Flags

The status flags (bits 0, 2, 4, 6, 7, and 11) of the EFLAGS register indicate the results of arithmetic instructions, such as the ADD, SUB, MUL, and DIV instructions. The status flag functions are:

- | | |
|--------------------|---|
| CF (bit 0) | Carry flag — Set if an arithmetic operation generates a carry or a borrow out of the most-significant bit of the result; cleared otherwise. This flag indicates an overflow condition for unsigned-integer arithmetic. It is also used in multiple-precision arithmetic. |
| PF (bit 2) | Parity flag — Set if the least-significant byte of the result contains an even number of 1 bits; cleared otherwise. |
| AF (bit 4) | Adjust flag — Set if an arithmetic operation generates a carry or a borrow out of bit 3 of the result; cleared otherwise. This flag is used in binary-coded decimal (BCD) arithmetic. |
| ZF (bit 6) | Zero flag — Set if the result is zero; cleared otherwise. |
| SF (bit 7) | Sign flag — Set equal to the most-significant bit of the result, which is the sign bit of a signed integer. (0 indicates a positive value and 1 indicates a negative value.) |
| OF (bit 11) | Overflow flag — Set if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; cleared otherwise. This flag indicates an overflow condition for signed-integer (two's complement) arithmetic. |

Of these status flags, only the CF flag can be modified directly, using the STC, CLC, and CMC instructions. Also the bit instructions (BT, BTS, BTR, and BTC) copy a specified bit into the CF flag.

The status flags allow a single arithmetic operation to produce results for three different data types: unsigned integers, signed integers, and BCD integers. If the result of an arithmetic operation is treated as an unsigned integer, the CF flag indicates an out-of-range condition (carry or a borrow); if treated as a signed integer (two's complement number), the OF flag indicates a carry or borrow; and if treated as a BCD digit, the AF flag indicates a carry or borrow. The SF flag indicates the sign of a signed integer. The ZF flag indicates either a signed- or an unsigned-integer zero.

When performing multiple-precision arithmetic on integers, the CF flag is used in conjunction with the add with carry (ADC) and subtract with borrow (SBB) instructions to propagate a carry or borrow from one computation to the next.

The condition instructions Jcc (jump on condition code cc), SETcc (byte set on condition code cc), LOOPcc, and CMOVcc (conditional move) use one or more of the status flags as condition codes and test them for branch, set-byte, or end-loop conditions.

3.4.3.2 DF Flag

The direction flag (DF, located in bit 10 of the EFLAGS register) controls string instructions (MOVS, CMPS, SCAS, LODS, and STOS). Setting the DF flag causes the string instructions to auto-decrement (to process strings from high addresses to low addresses). Clearing the DF flag causes the string instructions to auto-increment (process strings from low addresses to high addresses).

The STD and CLD instructions set and clear the DF flag, respectively.

3.4.3.3 System Flags and IOPL Field

The system flags and IOPL field in the EFLAGS register control operating-system or executive operations. **They should not be modified by application programs.** The functions of the system flags are as follows:

TF (bit 8)	Trap flag — Set to enable single-step mode for debugging; clear to disable single-step mode.
IF (bit 9)	Interrupt enable flag — Controls the response of the processor to maskable interrupt requests. Set to respond to maskable interrupts; cleared to inhibit maskable interrupts.
IOPL (bits 12 and 13)	I/O privilege level field — Indicates the I/O privilege level of the currently running program or task. The current privilege level (CPL) of the currently running program or task must be less than or equal to the I/O privilege level to access the I/O address space. The POPF and IRET instructions can modify this field only when operating at a CPL of 0.
NT (bit 14)	Nested task flag — Controls the chaining of interrupted and called tasks. Set when the current task is linked to the previously executed task; cleared when the current task is not linked to another task.
RF (bit 16)	Resume flag — Controls the processor's response to debug exceptions.
VM (bit 17)	Virtual-8086 mode flag — Set to enable virtual-8086 mode; clear to return to protected mode without virtual-8086 mode semantics.
AC (bit 18)	Alignment check flag — Set this flag and the AM bit in the CR0 register to enable alignment checking of memory references; clear the AC flag and/or the AM bit to disable alignment checking.
VIF (bit 19)	Virtual interrupt flag — Virtual image of the IF flag. Used in conjunction with the VIP flag. (To use this flag and the VIP flag the virtual mode extensions are enabled by setting the VME flag in control register CR4.)
VIP (bit 20)	Virtual interrupt pending flag — Set to indicate that an interrupt is pending; clear when no interrupt is pending. (Software sets and clears this flag; the processor only reads it.) Used in conjunction with the VIF flag.
ID (bit 21)	Identification flag — The ability of a program to set or clear this flag indicates support for the CPUID instruction.

For a detailed description of these flags: see Chapter 3, "Protected-Mode Memory Management," in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A.

3.4.3.4 RFLAGS Register in 64-Bit Mode

In 64-bit mode, EFLAGS is extended to 64 bits and called RFLAGS. The upper 32 bits of RFLAGS register is reserved. The lower 32 bits of RFLAGS is the same as EFLAGS.

3.5 INSTRUCTION POINTER

The instruction pointer (EIP) register contains the offset in the current code segment for the next instruction to be executed. It is advanced from one instruction boundary to the next in straight-line code or it is moved ahead or backwards by a number of instructions when executing JMP, Jcc, CALL, RET, and IRET instructions.

The EIP register cannot be accessed directly by software; it is controlled implicitly by control-transfer instructions (such as JMP, Jcc, CALL, and RET), interrupts, and exceptions. The only way to read the EIP register is to execute a CALL instruction and then read the value of the return instruction pointer from the procedure stack. The EIP register can be loaded indirectly by modifying the value of a return instruction pointer on the procedure stack and executing a return instruction (RET or IRET). See Section 6.2.4.2, “Return Instruction Pointer.”

All IA-32 processors prefetch instructions. Because of instruction prefetching, an instruction address read from the bus during an instruction load does not match the value in the EIP register. Even though different processor generations use different prefetching mechanisms, the function of the EIP register to direct program flow remains fully compatible with all software written to run on IA-32 processors.

3.5.1 Instruction Pointer in 64-Bit Mode

In 64-bit mode, the RIP register becomes the instruction pointer. This register holds the 64-bit offset of the next instruction to be executed. 64-bit mode also supports a technique called RIP-relative addressing. Using this technique, the effective address is determined by adding a displacement to the RIP of the next instruction.

3.6 OPERAND-SIZE AND ADDRESS-SIZE ATTRIBUTES

When the processor is executing in protected mode, every code segment has a default operand-size attribute and address-size attribute. These attributes are selected with the D (default size) flag in the segment descriptor for the code segment (see Chapter 3, “Protected-Mode Memory Management,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A). When the D flag is set, the 32-bit operand-size and address-size attributes are selected; when the flag is clear, the 16-bit size attributes are selected. When the processor is executing in real-address mode, virtual-8086 mode, or SMM, the default operand-size and address-size attributes are always 16 bits.

The operand-size attribute selects the size of operands. When the 16-bit operand-size attribute is in force, operands can generally be either 8 bits or 16 bits, and when the 32-bit operand-size attribute is in force, operands can generally be 8 bits or 32 bits.

The address-size attribute selects the sizes of addresses used to address memory: 16 bits or 32 bits. When the 16-bit address-size attribute is in force, segment offsets and displacements are 16 bits. This restriction limits the size of a segment to 64 KBytes. When the 32-bit address-size attribute is in force, segment offsets and displacements are 32 bits, allowing up to 4 GBytes to be addressed.

The default operand-size attribute and/or address-size attribute can be overridden for a particular instruction by adding an operand-size and/or address-size prefix to an instruction. See Chapter 2, “Instruction Format,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A. The effect of this prefix applies only to the targeted instruction.

Table 3-4 shows effective operand size and address size (when executing in protected mode or compatibility mode) depending on the settings of the D flag and the operand-size and address-size prefixes.

Table 3-3. Effective Operand- and Address-Size Attributes

D Flag in Code Segment Descriptor	0	0	0	0	1	1	1	1
Operand-Size Prefix 66H	N	N	Y	Y	N	N	Y	Y
Address-Size Prefix 67H	N	Y	N	Y	N	Y	N	Y
Effective Operand Size	16	16	32	32	32	32	16	16
Effective Address Size	16	32	16	32	32	16	32	16

NOTES:

Y: Yes - this instruction prefix is present.

N: No - this instruction prefix is not present.

3.6.1 Operand Size and Address Size in 64-Bit Mode

In 64-bit mode, the default address size is 64 bits and the default operand size is 32 bits. Defaults can be overridden using prefixes. Address-size and operand-size prefixes allow mixing of 32/64-bit data and 32/64-bit addresses on an instruction-by-instruction basis. Table 3-4 shows valid combinations of the 66H instruction prefix and the REX.W prefix that may be used to specify operand-size overrides in 64-bit mode. Note that 16-bit addresses are not supported in 64-bit mode.

REX prefixes consist of 4-bit fields that form 16 different values. The W-bit field in the REX prefixes is referred to as REX.W. If the REX.W field is properly set, the prefix specifies an operand size override to 64 bits. Note that software can still use the operand-size 66H prefix to toggle to a 16-bit operand size. However, setting REX.W takes precedence over the operand-size prefix (66H) when both are used.

In the case of SSE/SSE2/SSE3/SSSE3 SIMD instructions: the 66H, F2H, and F3H prefixes are mandatory for opcode extensions. In such a case, there is no interaction between a valid REX.W prefix and a 66H opcode extension prefix.

See Chapter 2, “Instruction Format,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

Table 3-4. Effective Operand- and Address-Size Attributes in 64-Bit Mode

L Flag in Code Segment Descriptor	1	1	1	1	1	1	1	1
REX.W Prefix	0	0	0	0	1	1	1	1
Operand-Size Prefix 66H	N	N	Y	Y	N	N	Y	Y
Address-Size Prefix 67H	N	Y	N	Y	N	Y	N	Y
Effective Operand Size	32	32	16	16	64	64	64	64
Effective Address Size	64	32	64	32	64	32	64	32

NOTES:

Y: Yes - this instruction prefix is present.

N: No - this instruction prefix is not present.

3.7 OPERAND ADDRESSING

IA-32 machine-instructions act on zero or more operands. Some operands are specified explicitly and others are implicit. The data for a source operand can be located in:

- the instruction itself (an immediate operand)
- a register
- a memory location
- an I/O port

When an instruction returns data to a destination operand, it can be returned to:

- a register
- a memory location
- an I/O port

3.7.1 Immediate Operands

Some instructions use data encoded in the instruction itself as a source operand. These operands are called **immediate** operands (or simply immediates). For example, the following ADD instruction adds an immediate value of 14 to the contents of the EAX register:

```
ADD EAX, 14
```

All arithmetic instructions (except the DIV and IDIV instructions) allow the source operand to be an immediate value. The maximum value allowed for an immediate operand varies among instructions, but can never be greater than the maximum value of an unsigned doubleword integer (2^{32}).

3.7.2 Register Operands

Source and destination operands can be any of the following registers, depending on the instruction being executed:

- 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP)
- 16-bit general-purpose registers (AX, BX, CX, DX, SI, DI, SP, or BP)
- 8-bit general-purpose registers (AH, BH, CH, DH, AL, BL, CL, or DL)
- segment registers (CS, DS, SS, ES, FS, and GS)
- EFLAGS register
- x87 FPU registers (ST0 through ST7, status word, control word, tag word, data operand pointer, and instruction pointer)
- MMX registers (MM0 through MM7)
- XMM registers (XMM0 through XMM7) and the MXCSR register
- control registers (CR0, CR2, CR3, and CR4) and system table pointer registers (GDTR, LDTR, IDTR, and task register)
- debug registers (DR0, DR1, DR2, DR3, DR6, and DR7)
- MSR registers

Some instructions (such as the DIV and MUL instructions) use quadword operands contained in a pair of 32-bit registers. Register pairs are represented with a colon separating them. For example, in the register pair EDX:EAX, EDX contains the high order bits and EAX contains the low order bits of a quadword operand.

Several instructions (such as the PUSHFD and POPFD instructions) are provided to load and store the contents of the EFLAGS register or to set or clear individual flags in this register. Other instructions (such as the Jcc instructions) use the state of the status flags in the EFLAGS register as condition codes for branching or other decision making operations.

The processor contains a selection of system registers that are used to control memory management, interrupt and exception handling, task management, processor management, and debugging activities. Some of these system registers are accessible by an application program, the operating system, or the executive through a set of system instructions. When accessing a system register with a system instruction, the register is generally an implied operand of the instruction.

3.7.2.1 Register Operands in 64-Bit Mode

Register operands in 64-bit mode can be any of the following:

- 64-bit general-purpose registers (RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, or R8-R15)
- 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, or R8D-R15D)
- 16-bit general-purpose registers (AX, BX, CX, DX, SI, DI, SP, BP, or R8W-R15W)
- 8-bit general-purpose registers: AL, BL, CL, DL, SIL, DIL, SPL, BPL, and R8L-R15L are available using REX prefixes; AL, BL, CL, DL, AH, BH, CH, DH are available without using REX prefixes.
- Segment registers (CS, DS, SS, ES, FS, and GS)
- RFLAGS register
- x87 FPU registers (ST0 through ST7, status word, control word, tag word, data operand pointer, and instruction pointer)
- MMX registers (MM0 through MM7)
- XMM registers (XMM0 through XMM15) and the MXCSR register
- Control registers (CR0, CR2, CR3, CR4, and CR8) and system table pointer registers (GDTR, LDTR, IDTR, and task register)
- Debug registers (DR0, DR1, DR2, DR3, DR6, and DR7)
- MSR registers
- RDX:RAX register pair representing a 128-bit operand

3.7.3 Memory Operands

Source and destination operands in memory are referenced by means of a segment selector and an offset (see Figure 3-9). Segment selectors specify the segment containing the operand. Offsets specify the linear or effective address of the operand. Offsets can be 32 bits (represented by the notation m16:32) or 16 bits (represented by the notation m16:16).

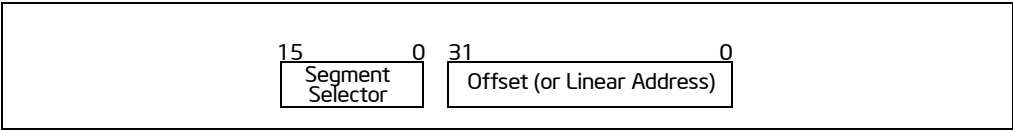


Figure 3-9. Memory Operand Address

3.7.3.1 Memory Operands in 64-Bit Mode

In 64-bit mode, a memory operand can be referenced by a segment selector and an offset. The offset can be 16 bits, 32 bits or 64 bits (see Figure 3-10).

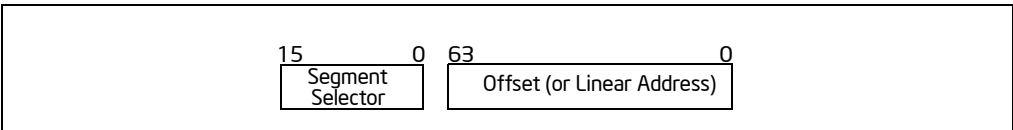


Figure 3-10. Memory Operand Address in 64-Bit Mode

3.7.4 Specifying a Segment Selector

The segment selector can be specified either implicitly or explicitly. The most common method of specifying a segment selector is to load it in a segment register and then allow the processor to select the register implicitly, depending on the type of operation being performed. The processor automatically chooses a segment according to the rules given in Table 3-5.

When storing data in memory or loading data from memory, the DS segment default can be overridden to allow other segments to be accessed. Within an assembler, the segment override is generally handled with a colon ":" operator. For example, the following MOV instruction moves a value from register EAX into the segment pointed to by the ES register. The offset into the segment is contained in the EBX register:

```
MOV ES:[EBX], EAX;
```

Table 3-5. Default Segment Selection Rules

Reference Type	Register Used	Segment Used	Default Selection Rule
Instructions	CS	Code Segment	All instruction fetches.
Stack	SS	Stack Segment	All stack pushes and pops. Any memory reference which uses the ESP or EBP register as a base register.
Local Data	DS	Data Segment	All data references, except when relative to stack or string destination.
Destination Strings	ES	Data Segment pointed to with the ES register	Destination of string instructions.

At the machine level, a segment override is specified with a segment-override prefix, which is a byte placed at the beginning of an instruction. The following default segment selections cannot be overridden:

- Instruction fetches must be made from the code segment.
- Destination strings in string instructions must be stored in the data segment pointed to by the ES register.
- Push and pop operations must always reference the SS segment.

Some instructions require a segment selector to be specified explicitly. In these cases, the 16-bit segment selector can be located in a memory location or in a 16-bit register. For example, the following MOV instruction moves a segment selector located in register BX into segment register DS:

```
MOV DS, BX
```

Segment selectors can also be specified explicitly as part of a 48-bit far pointer in memory. Here, the first double-word in memory contains the offset and the next word contains the segment selector.

3.7.4.1 Segmentation in 64-Bit Mode

In IA-32e 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 IA-32 mode, using the 16-bit or 32-bit protected mode semantics described above.

In 64-bit mode, segmentation is generally (but not completely) disabled, creating a flat 64-bit linear-address space. The processor treats the segment base of CS, DS, ES, SS as zero, creating a linear address that is equal to the effective address. The exceptions are the FS and GS segments, whose segment registers (which hold the segment base) can be used as additional base registers in some linear address calculations.

3.7.5 Specifying an Offset

The offset part of a memory address can be specified directly as a static value (called a **displacement**) or through an address computation made up of one or more of the following components:

- **Displacement** — An 8-, 16-, or 32-bit value.
- **Base** — The value in a general-purpose register.
- **Index** — The value in a general-purpose register.
- **Scale factor** — A value of 2, 4, or 8 that is multiplied by the index value.

The offset which results from adding these components is called an **effective address**. Each of these components can have either a positive or negative (2s complement) value, with the exception of the scaling factor. Figure 3-11 shows all the possible ways that these components can be combined to create an effective address in the selected segment.

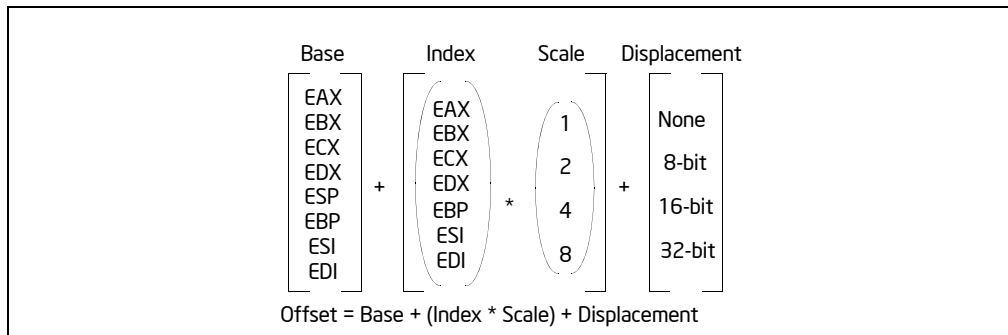


Figure 3-11. Offset (or Effective Address) Computation

The uses of general-purpose registers as base or index components are restricted in the following manner:

- The ESP register cannot be used as an index register.
- When the ESP or EBP register is used as the base, the SS segment is the default segment. In all other cases, the DS segment is the default segment.

The base, index, and displacement components can be used in any combination, and any of these components can be NULL. A scale factor may be used only when an index also is used. Each possible combination is useful for data structures commonly used by programmers in high-level languages and assembly language.

The following addressing modes suggest uses for common combinations of address components.

- **Displacement** — A displacement alone represents a direct (uncomputed) offset to the operand. Because the displacement is encoded in the instruction, this form of an address is sometimes called an absolute or static address. It is commonly used to access a statically allocated scalar operand.
- **Base** — A base alone represents an indirect offset to the operand. Since the value in the base register can change, it can be used for dynamic storage of variables and data structures.
- **Base + Displacement** — A base register and a displacement can be used together for two distinct purposes:
 - As an index into an array when the element size is not 2, 4, or 8 bytes—The displacement component encodes the static offset to the beginning of the array. The base register holds the results of a calculation to determine the offset to a specific element within the array.
 - To access a field of a record: the base register holds the address of the beginning of the record, while the displacement is a static offset to the field.

An important special case of this combination is access to parameters in a procedure activation record. A procedure activation record is the stack frame created when a procedure is entered. Here, the EBP register is the best choice for the base register, because it automatically selects the stack segment. This is a compact encoding for this common function.

- **(Index * Scale) + Displacement** — This address mode offers an efficient way to index into a static array when the element size is 2, 4, or 8 bytes. The displacement locates the beginning of the array, the index register holds the subscript of the desired array element, and the processor automatically converts the subscript into an index by applying the scaling factor.
- **Base + Index + Displacement** — Using two registers together supports either a two-dimensional array (the displacement holds the address of the beginning of the array) or one of several instances of an array of records (the displacement is an offset to a field within the record).
- **Base + (Index * Scale) + Displacement** — Using all the addressing components together allows efficient indexing of a two-dimensional array when the elements of the array are 2, 4, or 8 bytes in size.

3.7.5.1 Specifying an Offset in 64-Bit Mode

The offset part of a memory address in 64-bit mode can be specified directly as a static value or through an address computation made up of one or more of the following components:

- **Displacement** — An 8-bit, 16-bit, or 32-bit value.
- **Base** — The value in a 32-bit (or 64-bit if REX.W is set) general-purpose register.
- **Index** — The value in a 32-bit (or 64-bit if REX.W is set) general-purpose register.
- **Scale factor** — A value of 2, 4, or 8 that is multiplied by the index value.

The base and index value can be specified in one of sixteen available general-purpose registers in most cases. See Chapter 2, “Instruction Format,” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A.

The following unique combination of address components is also available.

- **RIP + Displacement** — In 64-bit mode, RIP-relative addressing uses a signed 32-bit displacement to calculate the effective address of the next instruction by sign-extend the 32-bit value and add to the 64-bit value in RIP.

3.7.6 Assembler and Compiler Addressing Modes

At the machine-code level, the selected combination of displacement, base register, index register, and scale factor is encoded in an instruction. All assemblers permit a programmer to use any of the allowable combinations of these addressing components to address operands. High-level language compilers will select an appropriate combination of these components based on the language construct a programmer defines.

3.7.7 I/O Port Addressing

The processor supports an I/O address space that contains up to 65,536 8-bit I/O ports. Ports that are 16-bit and 32-bit may also be defined in the I/O address space. An I/O port can be addressed with either an immediate operand or a value in the DX register. See Chapter 16, “Input/Output,” for more information about I/O port addressing.

