# Basic Execution Environment 27

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

The execution environment for the floating-point unit (FPU) is described in “Floating-Point Unit”.

## 27.1 Modes of Operation

The Intel Architecture supports three operating modes: protected mode, real-address mode, and system management mode. The operating mode determines which instructions and architectural features are accessible:

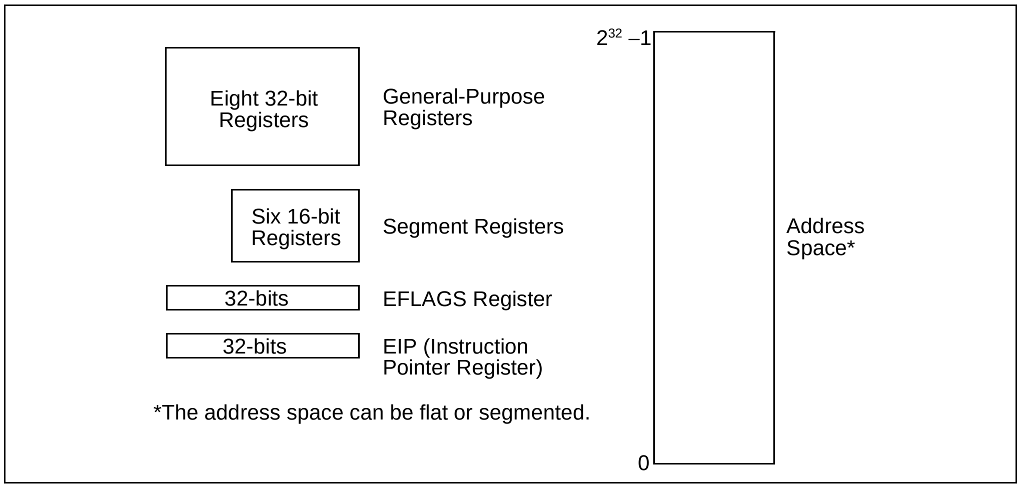
- **Protected mode.** The native state of the processor. In this mode all instructions and architectural features are available, providing the highest performance and capability. This is the recommended mode for all new applications and operating systems.
- Among the capabilities of protected mode is 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.** Provides the programming environment of the Intel 8086 processor with a few 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.** A standard architectural feature unique to all Intel processors, beginning with the Intel386 SL processor. 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 entire 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.

The basic execution environment is the same for each of these operating modes, as is described in the remaining sections of this chapter.

## 27.2 Overview of the Basic Execution Environment

Any program or task running on an Intel Architecture processor is given a set of resources for executing instructions and for storing code, data, and state information. These resources (shown in Figure 27-1) include an address space of up to $2^{32}$ bytes, a set of general data registers, a set of segment registers, and a set of status and control registers. When a program calls a procedure, a procedure stack is added to the execution environment. (Procedure calls and the procedure stack implementation are described in Chapter 4, *Procedure Calls, Interrupts, and Exceptions*.)

**Figure 27-1. Pentium® Pro Processor Basic Execution Environment**



## 27.3 Memory Organization

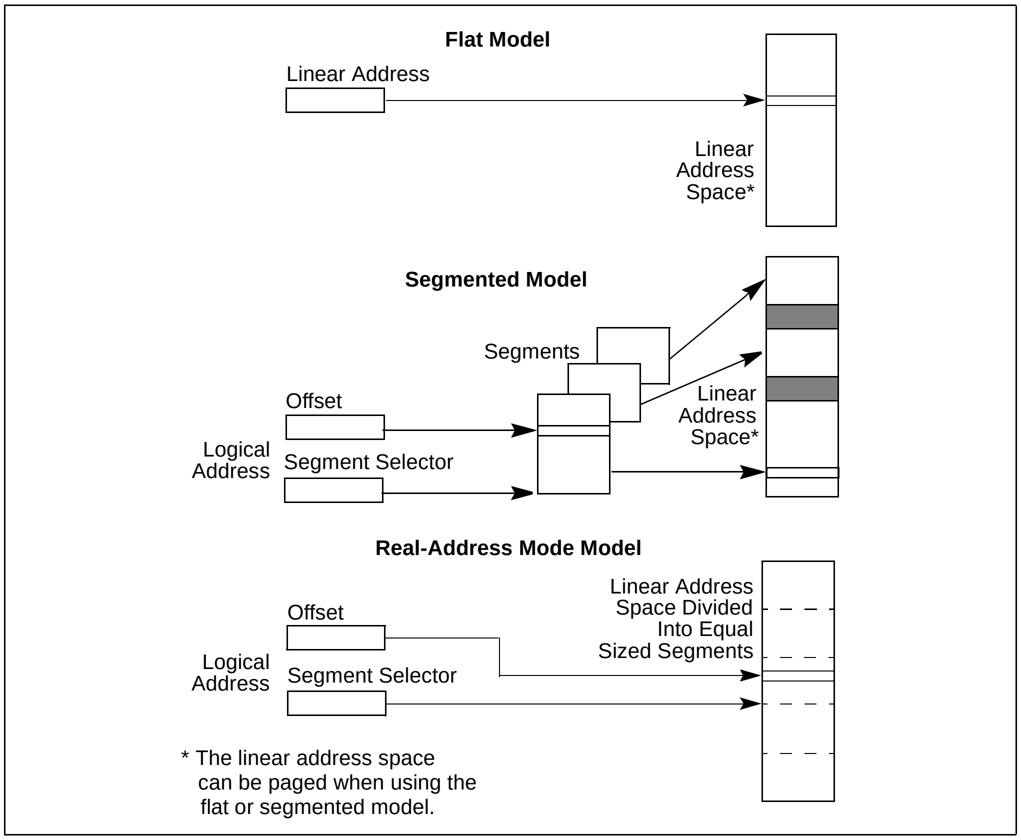
The memory that the processor addresses on its bus is called **physical memory**. Physical memory is organized as a sequence of 8-bit bytes. Each byte is assigned a unique address, called a **physical address**. The **physical address space** ranges from zero to a maximum of $2^{32}-1$ (4 gigabytes).

Virtually any operating system or executive designed to work with an Intel Architecture 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*, of the *Intel Architecture Software Developer's Manual, Volume 3*. The following paragraphs describe the basic methods of addressing memory when memory management is used.

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

With the **flat** memory model (see Figure 27-2), memory appears to a program as a single, continuous address space, called a **linear address space**. Code (a program's instructions), data, and the procedure stack are all contained in this address space. The linear address space is byte addressable, with addresses running contiguously from 0 to $2^{32} - 1$. An address for any byte in the linear address space is called a **linear address**.

**Figure 27-2. Three Memory Management Models**



With the **segmented** memory model, memory appears to a program as a group of independent address spaces called **segments**. When using this model, code, data, and stacks are typically contained in separate segments. To address a byte in a segment, a program must issue a **logical address**, which consists of a segment selector and an offset. (A logical address is often referred to as a **far pointer**.) The **segment selector** identifies the segment to be accessed and the offset identifies a byte in the address space of the segment. The programs running on an Intel Architecture 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. So, the processor translates each logical address into a linear address to access a memory location. 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. And placing the operating system's or executive's code, data, and stack in separate segments protects them from the application program and vice versa.

With either the flat or segmented model, the Intel Architecture provides facilities for dividing the linear address space into pages and mapping the pages into virtual memory. If an operating system/executive uses the Intel Architecture's paging mechanism, the existence of the pages is transparent to an application program.

The **real-address mode** model uses the memory model for the Intel 8086 processor, the first Intel Architecture processor. It was provided in all the subsequent Intel Architecture processors for 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 64K bytes in size each. The maximum size of the linear address space in real-address mode is $2^{20}$ bytes. (See Chapter 15, *8086 Emulation*, in the *Intel Architecture Software Developer's Manual, Volume 3*, for more information on this memory model.)

## 27.4 Modes of Operation

When writing code for the Pentium Pro 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 11, in the *Intel Architecture Software Developer's Manual, Volume 3*, for more information on the memory model used in SMM.)

## 27.5 32-Bit Vs. 16-Bit Address and Operand Sizes

The processor 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 ($2^{32}$), and 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 ($2^{16}$), and 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, it 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 address is still 0000FFFFH ($2^{16}$).

## 27.6 Registers

The processor provides 16 registers for use in general system and application programing. As shown in Figure 27-3, these registers can be grouped as follows:

- **General-purpose data registers**. These eight registers are available for storing operands and pointers.
- **Segment registers**. These registers hold up to six segment selectors.
- **Status and control registers**. These registers report and allow modification of the state of the processor and of the program being executed.

### 27.6.1 General-Purpose Data Registers

The 32-bit general-purpose data 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.

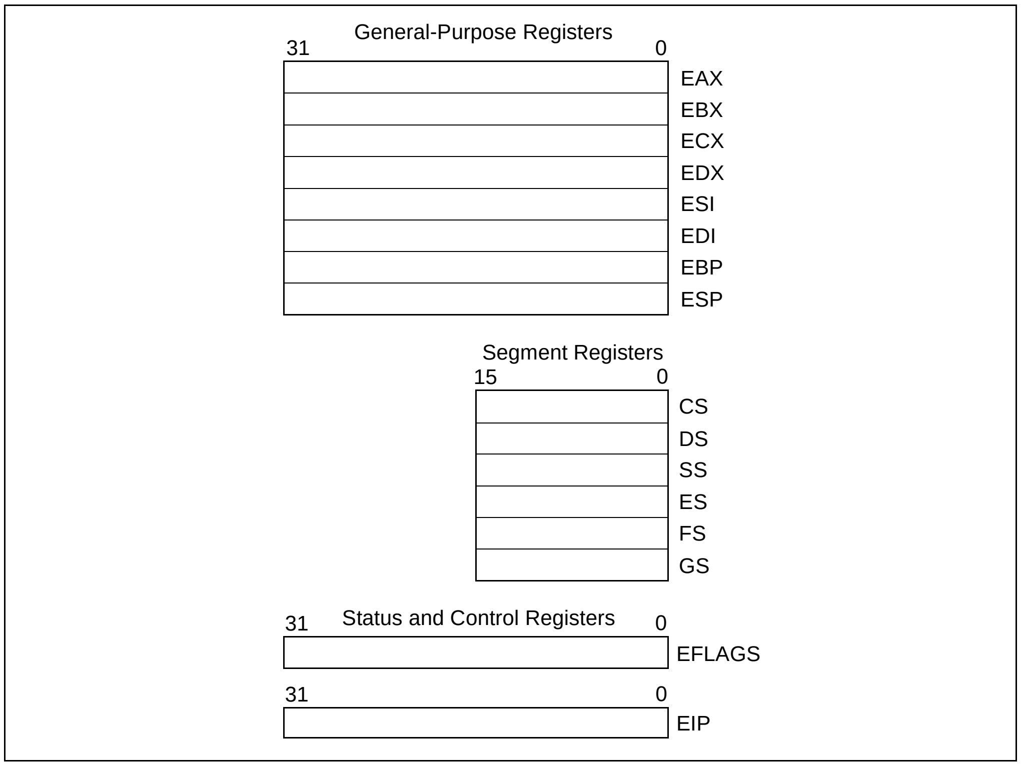
Although all of these registers are available for general storage of operands, results, and pointers, caution should be used when referencing the ESP register. The ESP register holds the stack pointer and as a general rule should not be used for any other purpose.

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.

The special uses of general-purpose registers by instructions are described in “Instruction Page Key”. The following is a summary of these special uses:

- EAX—Accumulator for operands and results data.
- EBX—Pointer to data in the DS segment.

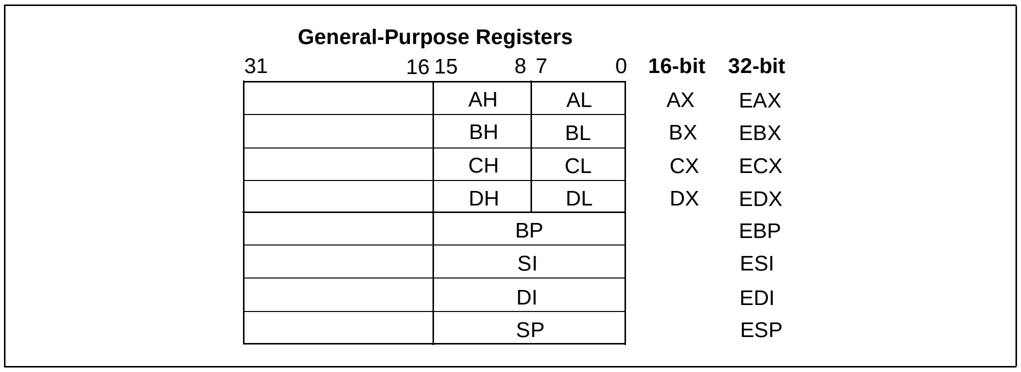
**Figure 27-3. Application Programming Registers**



- 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 27-4, 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, SP, SI, and DI. 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).

**Figure 27-4. Alternate General-Purpose Register Names**



### 27.6.2 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, you 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 you are writing system code, you may need to create segment selectors directly. (A detailed description of the segment-selector data structure is given in Chapter 3, *Protected-Mode Memory Management*, of the *Intel Architecture Software Developer's Manual, Volume 3*.)

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, the segment registers are loaded with segment selectors that point to overlapping segments, each of which begins at address 0 of the linear address space (as shown in Figure 27-5). 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 (as shown in Figure 27-6). 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 27-5. Use of Segment Registers for Flat Memory Model**

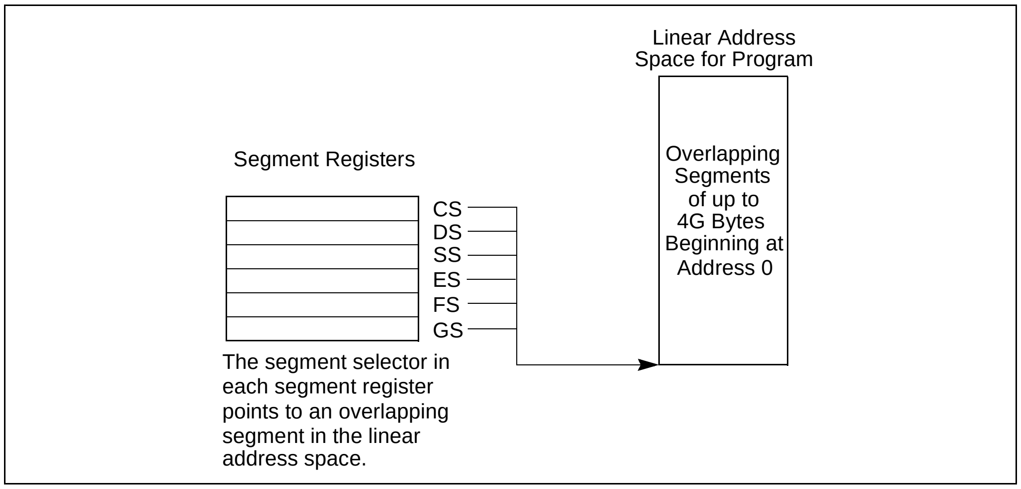
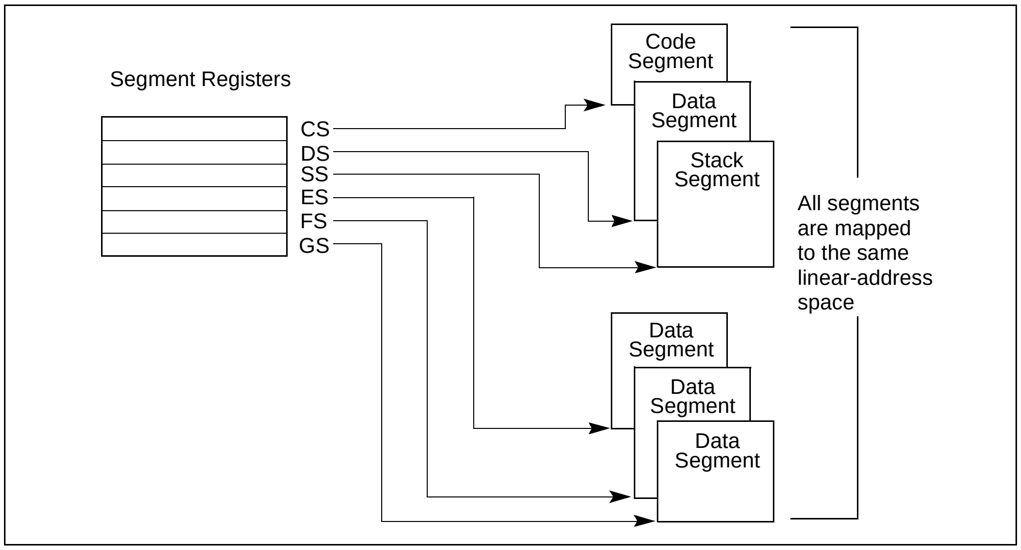

Linear Address Space for Program
Segment Registers
CS
DS
SS
ES
FS
GS
Overlapping Segments of up to 4G Bytes Beginning at Address 0
The segment selector in each segment register points to an overlapping segment in the linear address space.

**Figure 27-6. 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 linear address 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 a **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 “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 Intel Architecture with the Intel386 family of processors.

### 27.6.3 EFLAGS Register

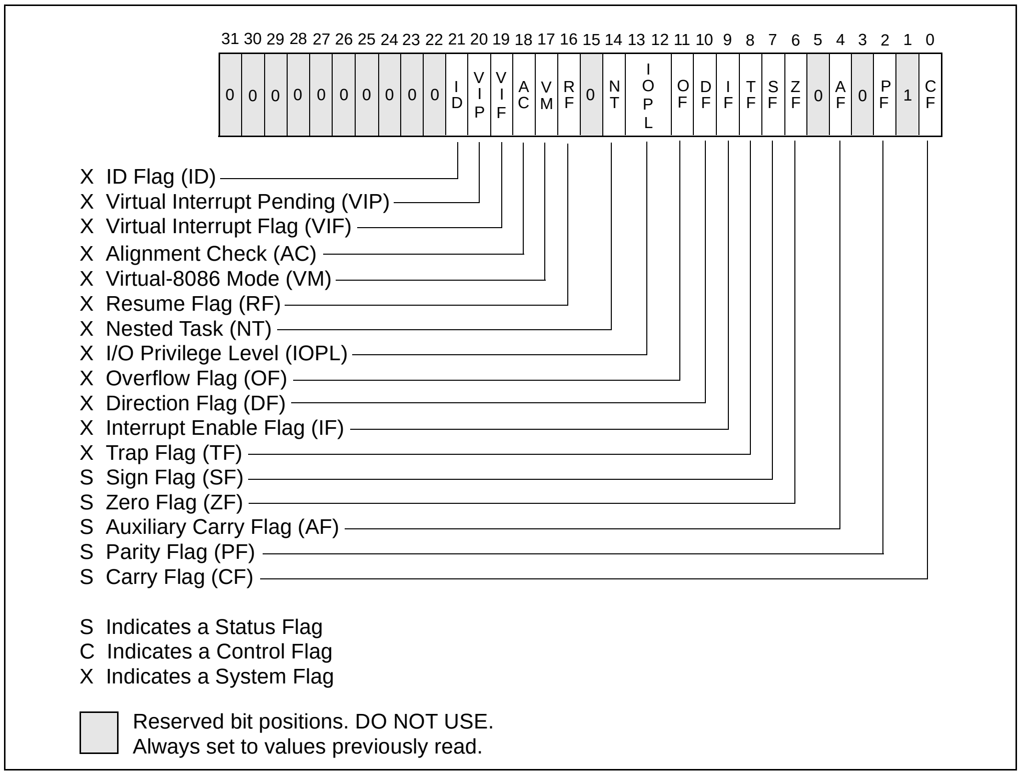
The 32-bit EFLAGS register contains a group of status flags, a control flag, and a group of system flags. Figure 27-7 defines the flags within this register. Following initialization of the processor (either by asserting the RESET pin or the INIT pin), the state of the EFLAGS register is 00000002H. Bits 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. However, 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 27-7. EFLAGS Register**


31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 0 0 0 0 0 0 0 0 ID VIP VIF AC VM RF 0 NT IOPL OF DF IF TF SF ZF 0 AF 0 PF 1 CF
X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
X Overflow Flag (OF)
X Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)
S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag
Reserved bit positions. DO NOT USE.
Always set to values previously read.

As the Intel 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 Intel Architecture processors to the next. As a result, code that accesses or modifies these flags for one family of Intel Architecture processors works as expected when run on later families of processors.

## 27.6.3.1 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 functions of the status flags are as follows:

| | |
|---|---|
| **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 J*cc* (jump on condition code *cc*), SET*cc* (byte set on condition code *cc*), LOOP*cc*, and CMOV*cc* (conditional move) use one or more of the status flags as condition codes and test them for branch, set-byte, or end-loop conditions.

#### 27.6.3.2 DF Flag

The direction flag (DF, located in bit 10 of the EFLAGS register) controls the string instructions (MOVS, CMPS, SCAS, LODS, and STOS). Setting the DF flag causes the string instructions to auto-decrement (that is, 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.

### 27.6.4 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 status flags are as follows:

| | |
|---|---|
| **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. |
| **TF (bit 8)** | **Trap flag.** Set to enable single-step mode for debugging; clear to disable single-step mode. |
| **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. This field can only be modified by the POPF and IRET instructions 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. |
| **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 to that an interrupt is pending; clear when no interrupts are 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. |

See Chapter 3, *Protected-Mode Memory Management*, in the *Intel Architecture Software Developer's Manual, Volume 3,* for a detail description of these flags.

## 27.7 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, J*cc*, 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, J*cc*, 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).

All Intel Architecture 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 EIP register to direct program flow remains fully compatible with all software written to run on Intel Architecture processors.

## 27.8 Operand-Size and Address-Size Attributes

When 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 Architecture Software Developer's Manual, Volume 3*). 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 sizes of operands that instructions operate on. 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 that can be addressed to 64 KBytes. When the 32-bit address-size attribute is in force, segment offsets and displacements are 32-bits, allowing segments of 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 "Instruction Prefixes" in Chapter 2 of the *Intel Architecture Software Developer's Manual, Volume 3*). The effect of this prefix applies only to the instruction it is attached to.

Table 27-1 shows effective operand size and address size (when executing in protected mode) depending on the settings of the B flag and the operand-size and address-size prefixes.

**Table 27-1. 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 |

**NOTE:**

| | |
|---|---|
| Y | Yes, this instruction prefix is present. |
| N | No, this instruction prefix is not present. |