Data Types and Addressing Modes 29

This section describes data types and addressing modes available to programmers of the Intel Architecture processors.

29.1 Fundamental Data Types

The fundamental data types of the Intel Architecture are bytes, words, doublewords, and quadwords (see Figure 29-1). A byte is eight bits, a word is 2 bytes (16 bits), a doubleword is 4 bytes (32 bits), and a quadword is 8 bytes (64 bits).

Figure 29-1. Fundamental Data Types

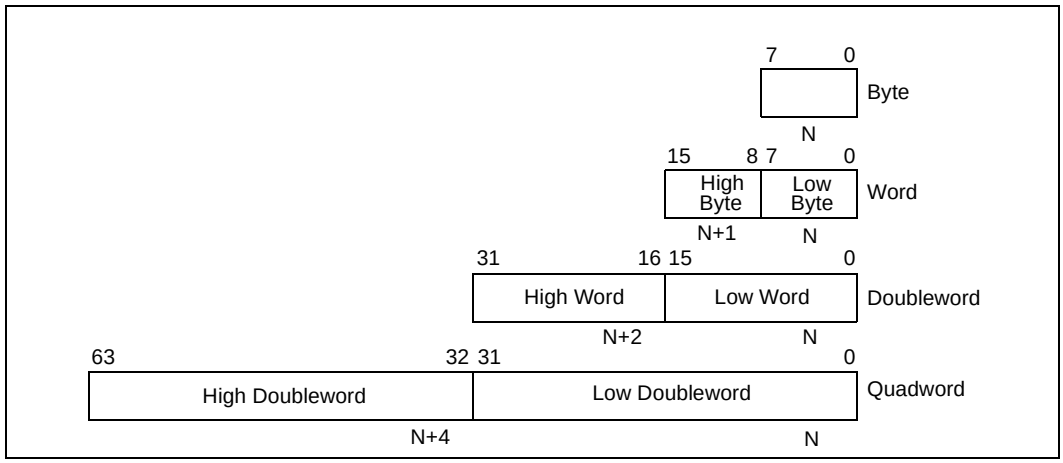
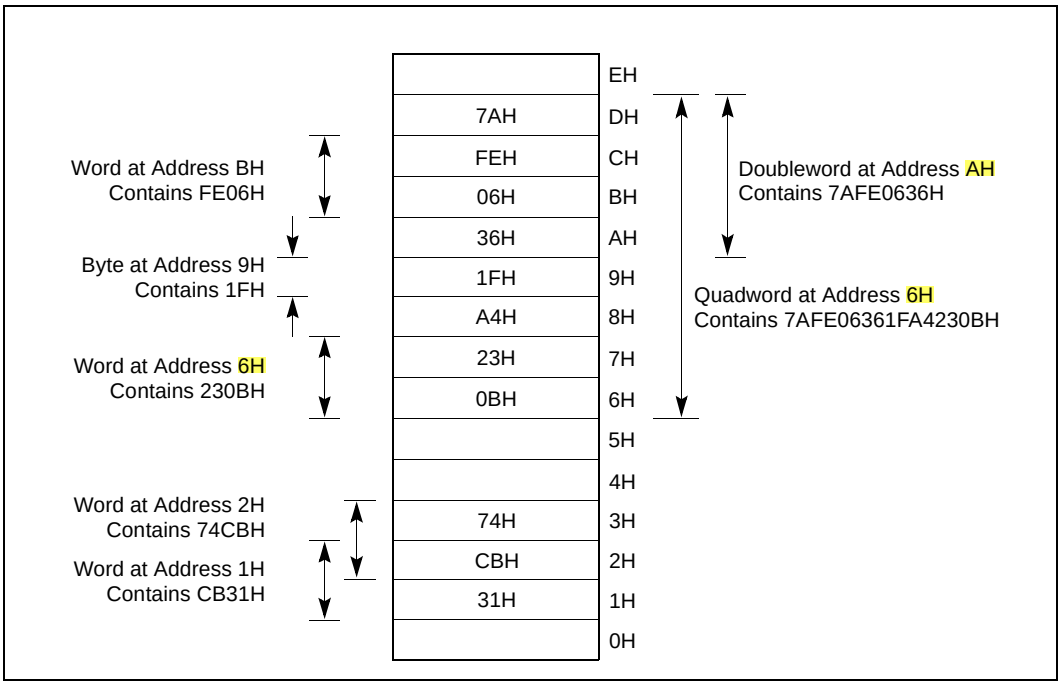


Figure 29-2 shows the byte order of each of the fundamental data types when referenced as operands in memory. The low byte (bits 0 through 7) of each data type occupies the lowest address in memory and that address is also the address of the operand.

29.1.1 Alignment of Words, Doublewords, and Quadwords

Words, doublewords, and quadwords do not need to be aligned in memory on natural boundaries. (The natural boundaries for words, double words, and quadwords are even-numbered addresses, addresses evenly divisible by four, and addresses evenly divisible by eight, respectively.) However, to improve the performance of programs, data structures (especially stacks) should be aligned on natural boundaries whenever possible. The reason for this is that the processor requires two memory accesses to make an unaligned memory access; whereas, aligned accesses require only one memory access. A word or doubleword operand that crosses a 4-byte boundary or a quadword operand that crosses an 8-byte boundary is considered unaligned and requires two separate memory bus cycles to access it; a word that starts on an odd address but does not cross a word boundary is considered aligned and can still be accessed in one bus cycle.

Figure 29-2. Bytes, Words, Doublewords and Quadwords in Memory



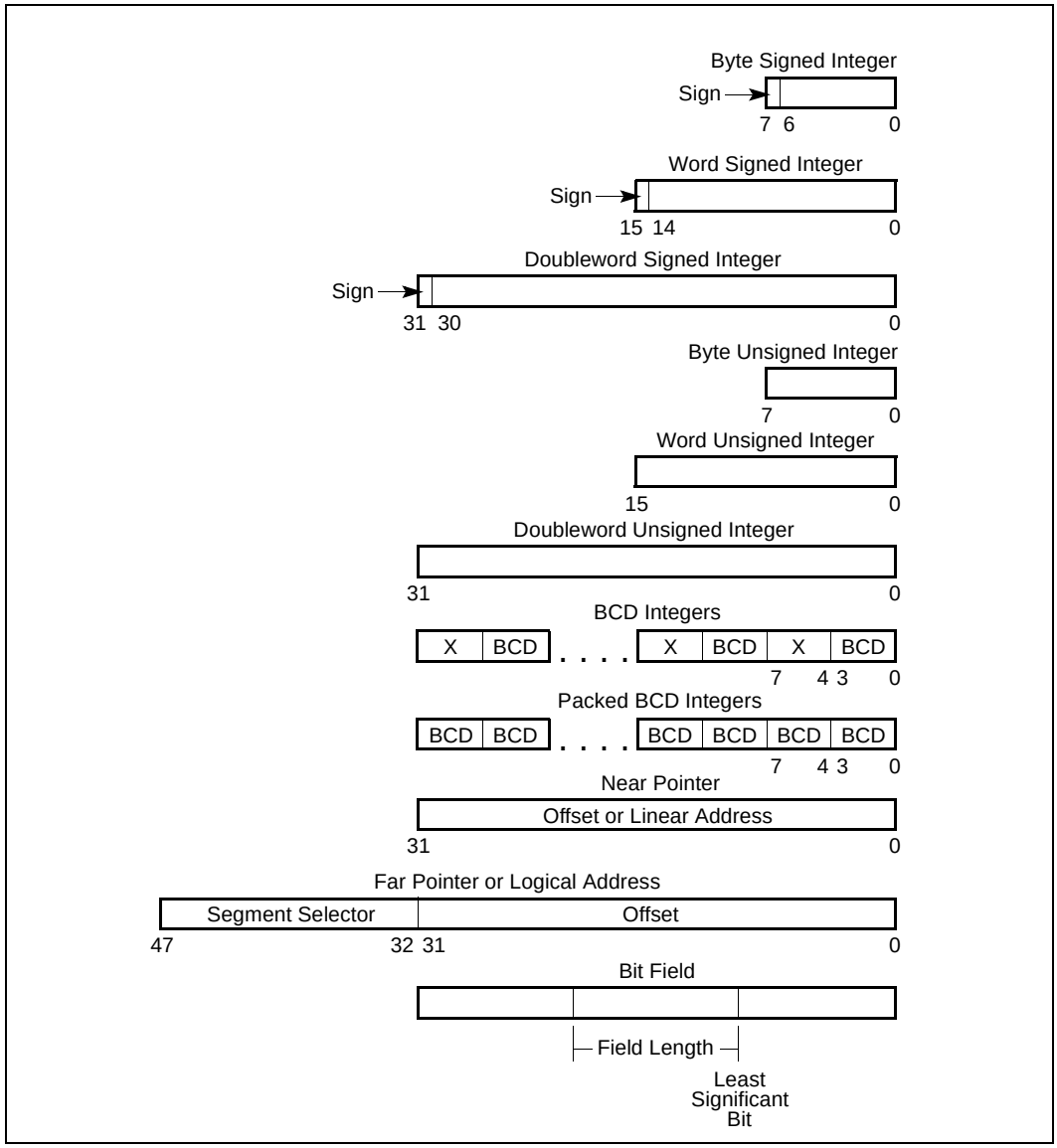
29.2 Numeric, Pointer, Bit Field, and String Data Types

Although bytes, words, and doublewords are the fundamental data types of the Intel Architecture, some instructions support additional interpretations of these data types to allow operations to be performed on numeric data types (signed and unsigned integers and BCD integers). See Figure 29-3. Also, some instructions recognize and operate on additional pointer, bit field, and string data types. The following sections describe these additional data types.

29.2.1 Integers

Integers are signed binary numbers held in a byte, word, or doubleword. All operations assume a two's complement representation. The sign bit is located in bit 7 in a byte integer, bit 15 in a word integer, and bit 31 in a doubleword integer. The sign bit is set for negative integers and cleared for positive integers and zero. Integer values range from –128 to +127 for a byte integer, from –32,768 to +32,767 for a word integer, and from -2^{31} to $+2^{31} - 1$ for a doubleword integer.

Figure 29-3. Numeric, Pointer, and Bit Field Data Types



29.2.2 Unsigned Integers

Unsigned integers are unsigned binary numbers contained in a byte, word, or doubleword. Unsigned integer values range from 0 to 255 for an unsigned byte integer, from 0 to 65,535 for an unsigned word integer, and from 0 to $2^{32} - 1$ for an unsigned doubleword integer. Unsigned integers are sometimes referred to as **ordinals**.

29.2.3 BCD Integers

Binary-coded decimal integers (BCD integers) are unsigned 4-bit integers with valid values ranging from 0 to 9. BCD integers can be unpacked (one BCD digit per byte) or packed (two BCD digits per byte). The value of an unpacked BCD integer is the binary value of the low half-byte (bits 0 through 3). The high half-byte (bits 4 through 7) can be any value during addition and subtraction, but must be zero during multiplication and division.

Packed BCD integers allow two BCD digits to be contained in one byte. Here, the digit in the high half-byte is more significant than the digit in the low half-byte.

29.2.4 Pointers

Pointers are addresses of locations in memory. The Pentium Pro processor recognizes two types of pointers: a **near pointer** (32 bits) and a **far pointer** (48 bits). A near pointer is a 32-bit offset (also called an **effective address**) within a segment. Near pointers are used for all memory references in a flat memory model or for references in a segmented model where the identity of the segment being accessed is implied. A far pointer is a 48-bit logical address, consisting of a 16-bit segment selector and a 32-bit offset. Far pointers are used for memory references in a segmented memory model where the identity of a segment being accessed must be specified explicitly.

29.2.5 Bit Fields

A **bit field** is a contiguous sequence of bits. It can begin at any bit position of any byte in memory and can contain up to 32 bits.

29.2.6 Strings

Strings are continuous sequences of bits, bytes, words, or doublewords. A **bit string** can begin at any bit position of any byte and can contain up to $2^{32}-1$ bits. A **byte string** can contain bytes, words, or doublewords and can range from zero to $2^{32}-1$ bytes (4 gigabytes).

29.2.7 Floating-Point Data Types

The processor's floating-point instructions recognize a set of real, integer, and BCD integer data types. See Floating-Point Data Types and Formats, for a description of FPU data types.

29.2.8 MMX™ Technology Data Types

Intel Architecture processors that implement the Intel MMX technology recognize a set of packed 64-bit data types. See MMX™ Data Types, for a description of the MMX data types.

29.3 Operand Addressing

An Intel Architecture machine-instruction acts on zero or more operands. Some operands are specified explicitly in an instruction and others are implicit to an instruction. An operand can be located in any of the following places:

- The instruction itself (an immediate operand).
- A register.
- A memory location.
- An I/O port.

29.3.1 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 the 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}).

29.3.2 Register Operands

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

- The 32-bit general-purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP).
- The 16-bit general-purpose registers (AX, BX, CX, DX, SI, DI, SP, or BP).
- The 8-bit general-purpose registers (AH, BH, CH, DH, AL, BL, CL, or DL).
- The segment registers (CS, DS, SS, ES, FS, and GS).
- The EFLAGS register.
- System registers, such as the global descriptor table (GDTR) or the interrupt descriptor table register (IDTR).

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.

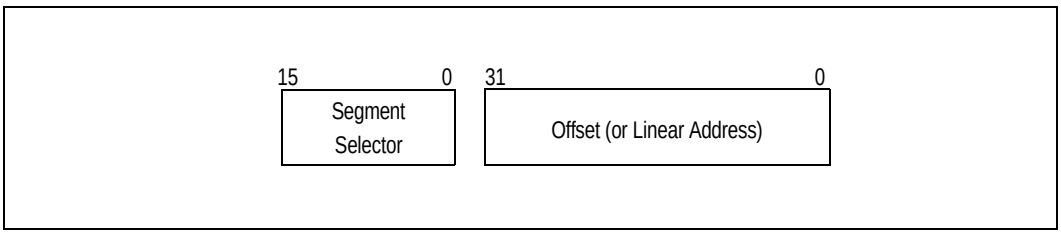
Several instructions (such as the PUSHFD and POPFD instructions) are provided to load and store the contents of the EFLAGS register or to set or clear individual flags in this register. Other instructions (such as the J*cc* instructions) use the state of the status flags in the EFLAGS register as condition 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.

29.3.3 Memory Operands

Source and destination operands in memory are referenced by means of a segment selector and an offset (see Figure 29-4). The segment selector specifies the segment containing the operand and the offset (the number of bytes from the beginning of the segment to the first byte of the operand) specifies the linear or effective address of the operand.

Figure 29-4. Memory Operand Address



29.3.3.1 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 29-1.

Table 29-1. Default Segment Selection Rules

Type of Reference	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.

When storing data in 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;
```

(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 doubleword in memory contains the offset and the next word contains the segment selector.

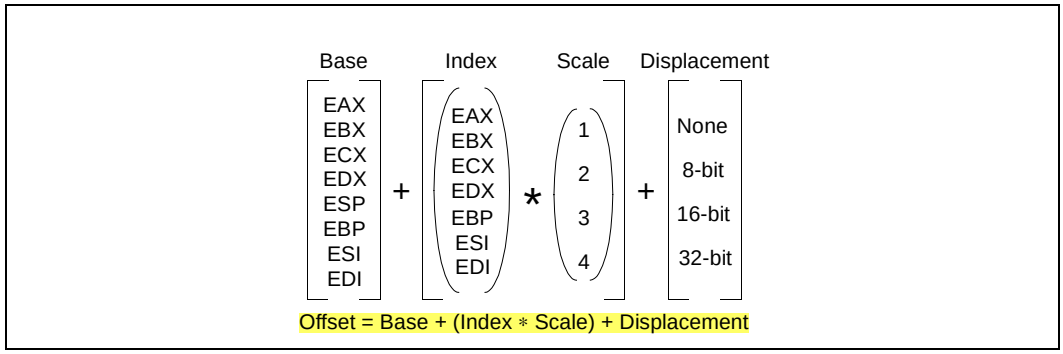
29.3.3.2 Specifying an Offset

The offset part of a memory address can be specified either directly as an 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 29-5 shows all the possible ways that these components can be combined to create an effective address in the selected segment.

Figure 29-5. 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 an 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

if you use any constant like [epb+4] [esp-4] the 4 , i mean the constant value is encoded as displacement displacement is only used to access memory

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.

29.3.3.3 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 (HLL) compilers will select an appropriate combination of these components based on the HHL construct a programmer defines.

29.3.4 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 “Input/Output” for more information about I/O port addressing.