University of Babylon, College of science for women
Dept. of Computer science

Computer Architecture

Second year

Dr. Salah Al-Obaidi

Lecture #5: Types of Operands Spring 2025

Contents

4.4 x86 Addressing Modes

The x86 is equipped with a variety of addressing modes intended to allow the efficient execution of high- level languages. Figure 4.1 indicates the logic involved.

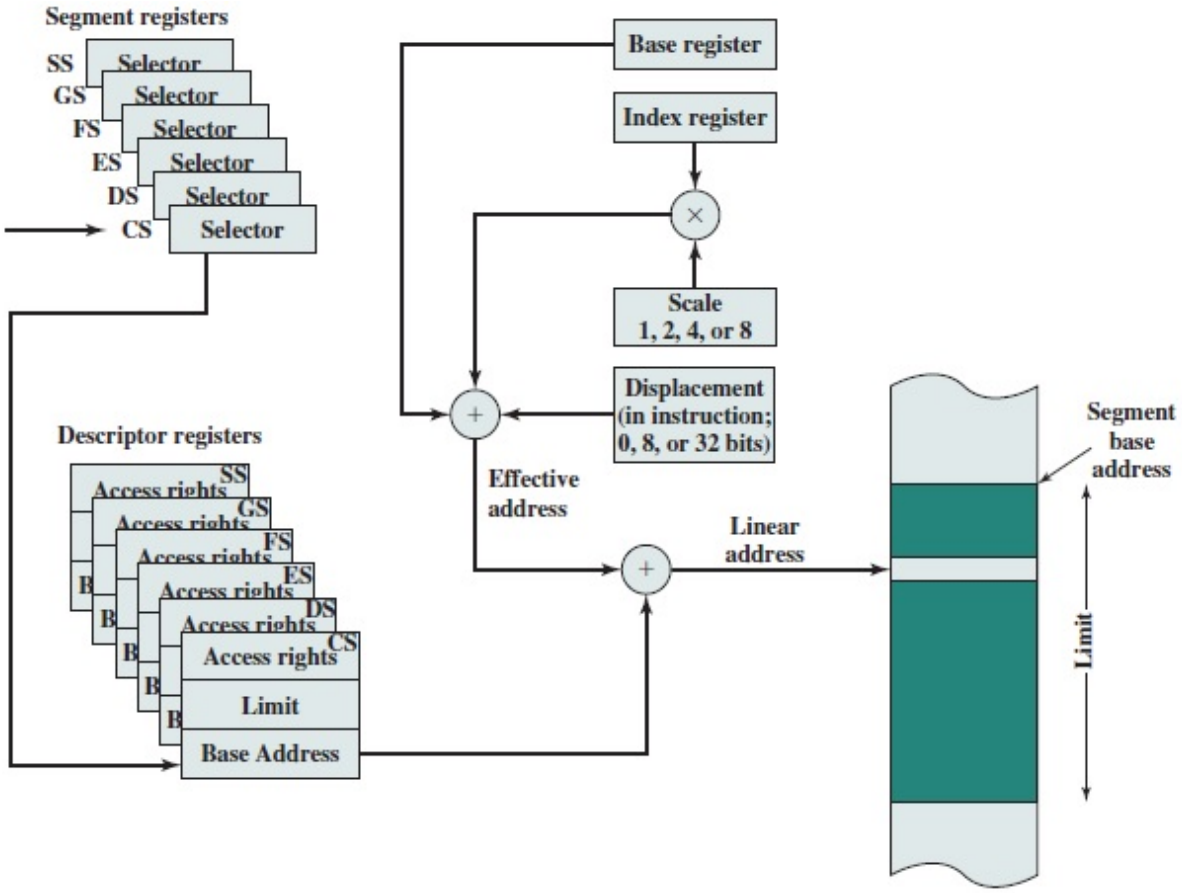

Figure 4.1: x86 Addressing Mode Calculation.

Table 4.1 lists the x86 addressing modes. Let us consider each of these in turn.

For the **immediate mode**, the operand is included in the instruction. The operand can be a byte, word, or doubleword of data.

For **register operand mode**, the operand is located in a register. For general instructions, such as data transfer, arithmetic, and logical instructions, the operand can be one of the 32-bit general registers (EAX, EBX, ECX, EDX, ESI,

Table 4.1: x86 Addressing Modes.

Mode	**Algorithm**
Immediate	Operand = A
Register Operand	LA = R
Displacement	LA = (SR) + A
Base	LA = (SR) + (B)
Base with Displacement	LA = (SR) + (B) + A
Scaled Index with Displacement	LA = (SR) + (I) * S + A
Base with Index and Displacement	LA = (SR) + (B) + (I) + A
Base with Scaled Index and Displacement	LA = (SR) + (I) * S + (B) + A
Relative	LA = (PC) + A

LA = linear address
(X) = contents of X
SR = segment register
PC = program counter
A = contents of an address field in the instruction
R = register
B = base register
I = index register
S = scaling factor

EDI, ESP, EBP), one of the 16-bit general registers (AX, BX, CX, DX, SI, DI, SP, BP), or one of the 8-bit general registers (AH, BH, CH, DH, AL, BL, CL, DL). There are also some instructions that reference the segment selector registers (CS, DS, ES, SS, FS, GS).

The remaining addressing modes reference locations in memory. The memory location must be specified in terms of the segment containing the location and the offset from the beginning of the segment. In some cases, a segment is specified explicitly; in others, the segment is specified by simple rules that assign a segment by default.

In the **displacement mode**, the operand's offset (the effective address of Figure 4.1) is contained as part of the instruction as an 8-, 16-, or 32-bit displacement. With segmentation, all addresses in instructions refer merely to an offset in a segment. The displacement addressing mode is found on few machines because it leads to long instructions. In the case of the x86, the displacement value can be as long as 32 bits, making for a 6-byte instruction. Displacement addressing can be useful for referencing global variables.

The remaining addressing modes are indirect, in the sense that the address portion of the instruction tells the processor where to look to find the address.

The **base mode** specifies that one of the 8-, 16-, or 32-bit registers contains the effective address.This is equivalent to what we have referred to as register indirect addressing.

In the **base with displacement mode**, the instruction includes a displacement to be added to a base register, which may be any of the general-purpose registers. Examples of uses of this mode are as follows:

- Used by a compiler to point to the start of a local variable area.
- Used to index into an array when the element size is not 1, 2, 4, or 8 bytes and which therefore cannot be indexed using an index register.
- Used to access a field of a record.

In the **scaled index with displacement mode**, the instruction includes a displacement to be added to a register, in this case called an *index register*. The index register may be any of the general-purpose registers except the one called **ESP**, which is generally used for stack processing. In calculating the effective address, the contents of the index register are multiplied by a scaling factor of 1, 2, 4, or 8, and then added to a displacement.This mode is very convenient for indexing arrays.

The **base with index and displacement mode** sums the contents of the base register, the index register, and a displacement to form the effective address. Again, the base register can be any general-purpose register and the index register can be any general-purpose register except **ESP**. As an example, this addressing mode could be used for accessing a local array on a stack frame. This mode can also be used to support a two-dimensional array.

The **based scaled index with displacement mode** sums the contents of the index register multiplied by a scaling factor, the contents of the base register, and the displacement. This is useful if an array is stored in a stack frame; in this case, the array elements would be 2, 4, or 8 bytes each in length. This

mode also provides efficient indexing of a two-dimensional array when the array elements are 2, 4, or 8 bytes in length.

Finally, **relative addressing** can be used in transfer-of-control instructions. A displacement is added to the value of the program counter, which points to the next instruction. In this case, the displacement is treated as a signed byte, word, or doubleword value, and that value either increases or decreases the address in the program counter

5. Types of Operands

Machine instructions operate on data. The most important general categories of data are

- Addresses
- Numbers
- Characters
- Logical data

The addresses are, in fact, a form of data. In many cases, some calculation must be performed on the operand reference in an instruction to determine the main or virtual memory address. In this context, addresses can be considered to be unsigned integers.

Other common data types are numbers, characters, and logical data, and each of these is briefly examined in this section. Beyond that, some machines define specialized data types or data structures. For example, there may be machine operations that operate directly on a list or a string of characters.

Numbers

All machine languages include numeric data types. Even in nonnumeric data processing, there is a need for numbers to act as counters, field widths, and so forth.

Three types of numerical data are common in computers:

- Binary integer or binary fixed point.
- Binary floating point.
- Decimal.

Although all internal computer operations are binary in nature, the human users of the system deal with decimal numbers. Thus, there is a necessity to convert from decimal to binary on input and from binary to decimal on output. For applications in which there is a great deal of I/O and comparatively little, comparatively simple computation, it is preferable to store and operate on the numbers in decimal form. The most common representation for this purpose is packed decimal.

With packed decimal, each decimal digit is represented by a 4-bit code, in the obvious way, with two digits stored per byte. Thus, and Note that this is a rather inefficient code because only 10 of 16 possible 4-bit values are used. To form numbers, 4-bit codes are strung together, usually in multiples of 8 bits. Thus, the code for 246 is 0000 0010 0100 0110. This code is clearly less compact than a straight binary representation, but it avoids the conversion overhead. Negative numbers can be represented by including a 4-bit sign digit at either the left or right end of a string of packed decimal digits. Standard sign values are 1100 for positive and 1101 for negative.

Many machines provide arithmetic instructions for performing operations directly on packed decimal numbers.

Characters

A common form of data is *text* or *character strings*. While textual data are most convenient for human beings, they cannot, in character form, be easily stored or transmitted by data processing and communications systems. Such systems are designed for binary data. Thus, a number of codes have been devised by which characters are represented by a sequence of bits. Perhaps the earliest common example of this is the Morse code. Today, the most commonly used

character code in the International Reference Alphabet (IRA), referred to in the United States as the American Standard Code for Information Interchange (**ASCII**). Each character in this code is represented by a unique 7-bit pattern; thus, 128 different characters can be represented. This is a larger number than is necessary to represent printable characters, and some of the patterns represent control characters. Some of these control characters have to do with controlling the printing of characters on a page. Others are concerned with communications procedures. IRA-encoded characters are almost always stored and transmitted using 8 bits per character. The eighth bit may be set to 0 or used as a parity bit for error detection. In the latter case, the bit is set such that the total number of binary 1s in each octet is always odd (*odd parity*) or always even (*even parity*).

Another code used to encode characters is the Extended Binary Coded Decimal Interchange Code (**EBCDIC**). EBCDIC is used on IBM mainframes. It is an 8-bit code. As with IRA, EBCDIC is compatible with packed decimal. In the case of EBCDIC, the codes **11110000** through **11111001** represent the digits 0 through 9.

Logical Data

Normally, each word or other addressable unit (byte, halfword, and so on) is treated as a single unit of data. It is sometimes useful, however, to consider an *n-bit* unit as consisting of *n* 1-bit items of data, each item having the value 0 or 1. When data are viewed this way, they are considered to be *logical data*.

There are two advantages to the bit-oriented view:

- First, we may sometimes wish to store an array of Boolean or binary data items, in which each item can take on only the values 1 (true) and 0 (false).With logical data, memory can be used most efficiently for this storage.

- Second, there are occasions when we wish to manipulate the bits of a data

item. For example, if floating-point operations are implemented in software, we need to be able to shift significant bits in some operations. Another example: To convert from IRA to packed decimal, we need to extract the rightmost 4 bits of each byte.