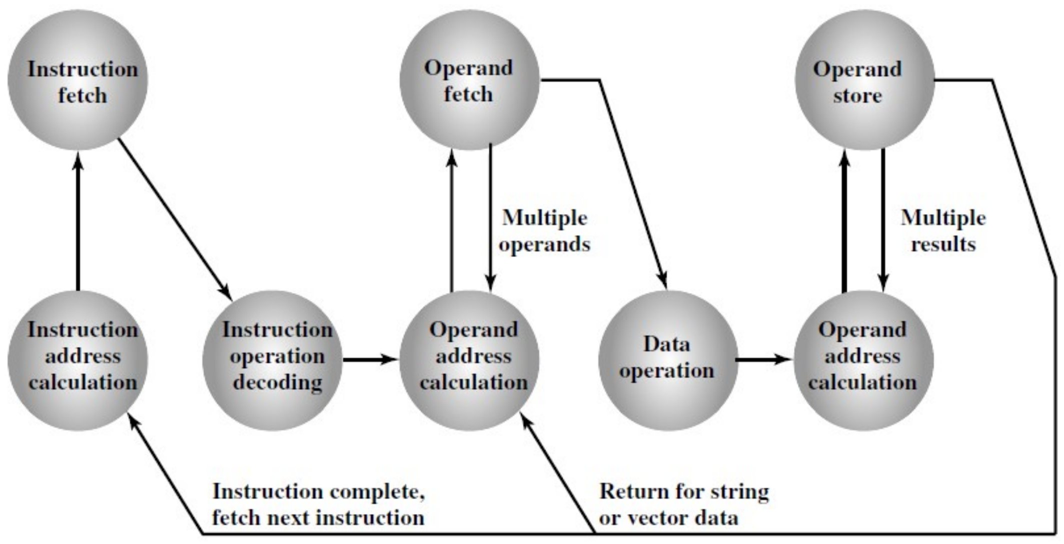
Instruction code: A CPU instruction code is a group of bits that instruct the computer to perform a specific task.

Micro operation: In computer central processing units, micro-operations (also known as micro-ops or μops, historically also as micro-actions) are **detailed low-level instructions used in some designs to implement complex machine instructions** (sometimes termed macro-instructions in this context)

MACHINE INSTRUCTION CHARACTERISTICS

The operation of the processor is determined by the instructions it executes, referred to as *machine instructions* or *computer instructions.* The collection of different instructions that the processor can execute is referred to as the processor's *instruction set.*

Instruction Cycle State Diagram


Instruction fetch
Operand fetch
Operand store
Multiple operands
Multiple results
Instruction address calculation
Instruction operation decoding
Operand address calculation
Data operation
Operand address calculation
Instruction complete, fetch next instruction
Return for string or vector data

Elements of a Machine Instruction

Each instruction must contain the information required by the processor for execution. Above figure, shows the steps involved in instruction execution and, by implication, defines the elements of a machine instruction.These elements are as follows:

• **Operation code:** Specifies the operation to be performed (e.g., ADD, I/O).
The operation is specified by a binary code, known as the operation code, or **opcode**.
• **Source operand reference:** The operation may involve one or more source operands, that is, operands that are inputs for the operation.
• **Result operand reference:** The operation may produce a result.
• **Next instruction reference:** This tells the processor where to fetch the next instruction after the execution of this instruction is complete.

The address of the next instruction to be fetched could be either a real address or a virtual address, depending on the architecture. Generally, the distinction is transparent to the instruction set architecture. In most cases, the next instruction to be fetched immediately follows the current instruction. In those cases, there is no explicit reference to the next instruction. When an explicit reference is needed, then the main memory or virtual memory address must be supplied.

Source and result operands can be in one of four areas:

• **Main or virtual memory:** As with next instruction references, the main or virtual memory address must be supplied.
• **Processor register:** With rare exceptions, a processor contains one or more registers that may be referenced by machine instructions. If only one register exists, reference to it may be implicit. If more than one register exists, then each register is assigned a unique name or number, and the instruction must contain the number of the desired register.
• **Immediate:** The value of the operand is contained in a field in the instruction being executed.
• **I/O device:** The instruction must specify the I/O module and device for the operation. If memory-mapped I/O is used, this is just another main or virtual memory address.

Instruction Representation

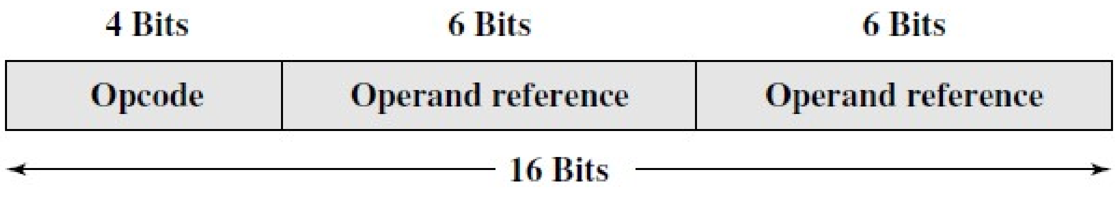


Fig: A simple instruction Format

Within the computer, each instruction is represented by a sequence of bits.The instruction is divided into fields, corresponding to the constituent elements of the instruction. A simple example of an instruction format is shown in the above figure.
With most instruction sets, more than one format is used. During instruction execution, an instruction is read into an instruction register (IR) in the processor. The processor must be able to extract the data from the various instruction fields to perform the required operation.

It is difficult for both the programmer and the reader of textbooks to deal with binary representations of machine instructions. Thus, it has become common practice to use a *symbolic representation* of machine instructions.

Opcodes are represented by abbreviations, called *mnemonics that* indicate the operation. Common examples include

ADD	Add
SUB	Subtract
MUL	Multiply
DIV	Divide
LOAD	Load data from memory
STOR	Store data to memory

Operands are also represented symbolically. For example, the instruction

ADD R, Y

may mean add the value contained in data location Y to the contents of register R. In this example, Y refers to the address of a location in memory, and R refers to a particular register. Note that the operation is performed on the contents of a location, not on its address.

Thus, it is possible to write a machine-language program in symbolic form. Each symbolic opcode has a fixed binary representation, and the programmer specifies the location of each symbolic operand. For example, the programmer might begin with a list of definitions:

$$X = 513$$

$$Y = 514$$

and so on. A simple program would accept this symbolic input, convert opcodes and operand references to binary form, and construct binary machine instructions.

Machine-language programmers are rare to the point of nonexistence. Most programs today are written in a high-level language or, failing that, assembly language. However, symbolic machine language remains a useful tool for describing machine instructions, and we will use it for that purpose.

Instruction Types

Consider a high-level language instruction that could be expressed in a language such as BASIC or FORTRAN. For example,

$$X = X + Y$$

This statement instructs the computer to add the value stored in Y to the value stored in X and put the result in X. How might this be accomplished with machine instructions? Let us assume that the variables X and Y correspond to locations 513 and 514. If we assume a simple set of machine instructions, this operation could be accomplished with three instructions:

1. Load a register with the contents of memory location 513.
2. Add the contents of memory location 514 to the register.
3. Store the contents of the register in memory location 513.

As can be seen, the single BASIC instruction may require three machine instructions.This is typical of the relationship between a high-level language and a machine language. A high-level language expresses operations in a concise algebraic form, using variables. A machine language expresses operations in a basic form involving the movement of data to or from registers.
With this simple example to guide us, let us consider the types of instructions that must be included in a practical computer. A computer should have a set of instructions that allows the user to formulate any data processing task. Another way to view it is to consider the capabilities of a high-level programming language. Any program written in a high-level language must be translated into machine language to be executed. Thus, the set of machine instructions must be sufficient to express any of the instructions from a high-level language. With this in mind we can categorize instruction types as follows:

• **Data processing:** Arithmetic and logic instructions
• **Data storage:** Movement of data into or out of register and or memory locations
• **Data movement:** I/O instructions
• **Control:** Test and branch instructions

Arithmetic instructions provide computational capabilities for processing numeric data. *Logic* (Boolean) instructions operate on the bits of a word as bits rather than as numbers; thus, they provide capabilities for processing any other type of data the user may wish to employ. These operations are performed primarily on data in processor registers. Therefore, there must be *memory* instructions for moving data between memory and the registers. *I/O* instructions are needed to transfer programs and data into memory and the results of computations back out to the user.

Test instructions are used to test the value of a data word or the status of a computation.

Branch instructions are then used to branch to a different set of instructions depending on the decision made.

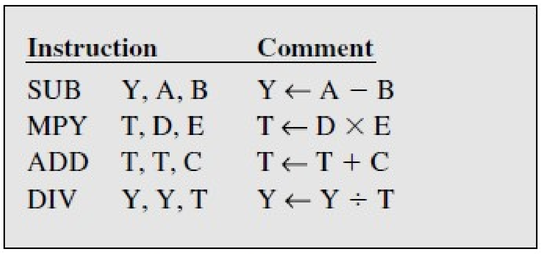
Number of Addresses

One of the traditional ways of describing processor architecture is in terms of the number of addresses contained in each instruction. This dimension has become less significant with the increasing complexity of processor design. Nevertheless, it is useful at this point to draw and analyze this distinction.
What is the maximum number of addresses one might need in an instruction? Evidently, arithmetic and logic instructions will require the most operands. Virtually all arithmetic and logic operations are either unary (one source operand) or binary (two source operands). Thus, we would need a maximum of two addresses to reference source operands. The result of an operation must be stored, suggesting a third address, which defines a destination operand. Finally, after completion of an instruction, the next instruction must be fetched, and its address is needed.

Processor architecture can be described in four categories on the basis of number of addresses in the instructions:

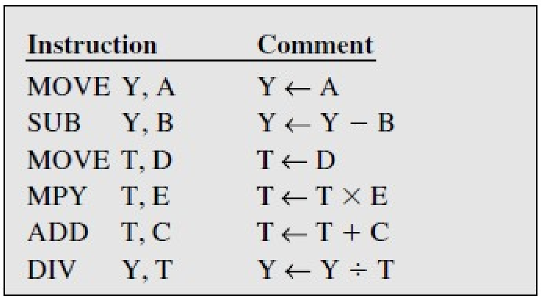
- Three Address instructions
- Two Address instructions
- One Address instructions
- Zero Address instructions

To Execute Y = (A – B)/[C + (D * E)] Three, Two and One address instruction processors follow following instruction set:



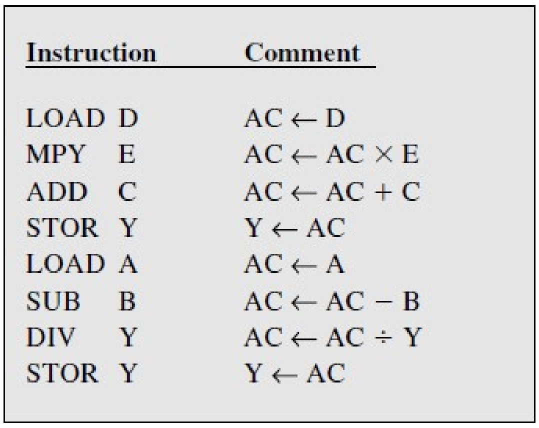
Instruction		Comment
SUB	Y, A, B	$Y \leftarrow A - B$
MPY	T, D, E	$T \leftarrow D \times E$
ADD	T, T, C	$T \leftarrow T + C$
DIV	Y, Y, T	$Y \leftarrow Y \div T$

(a) Three-address instructions



Instruction		Comment
MOVE	Y, A	$Y \leftarrow A$
SUB	Y, B	$Y \leftarrow Y - B$
MOVE	T, D	$T \leftarrow D$
MPY	T, E	$T \leftarrow T \times E$
ADD	T, C	$T \leftarrow T + C$
DIV	Y, T	$Y \leftarrow Y \div T$

(b) Two-address instructions



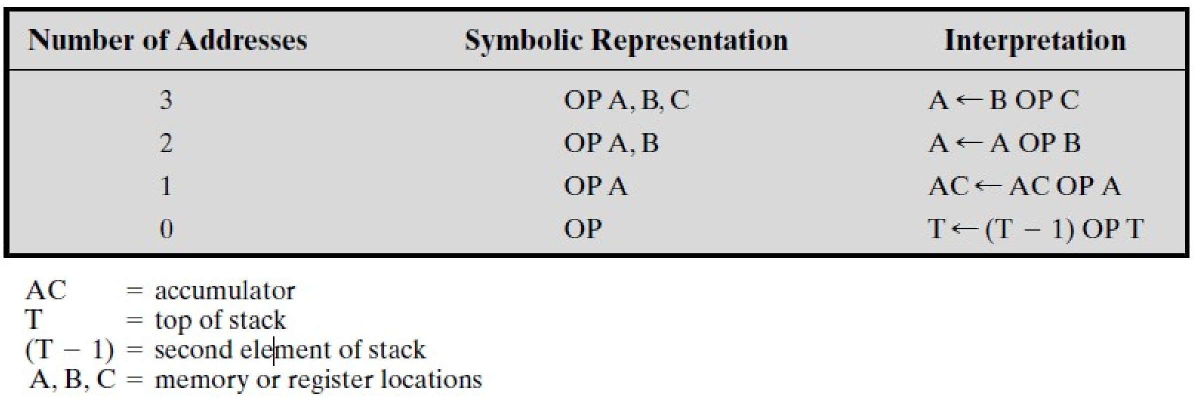
Instruction		Comment
LOAD	D	$AC \leftarrow D$
MPY	E	$AC \leftarrow AC \times E$
ADD	C	$AC \leftarrow AC + C$
STOR	Y	$Y \leftarrow AC$
LOAD	A	$AC \leftarrow A$
SUB	B	$AC \leftarrow AC - B$
DIV	Y	$AC \leftarrow AC \div Y$
STOR	Y	$Y \leftarrow AC$

(c) One-address instructions

Three-address instruction formats are not common because they require a relatively long instruction format to hold the three address references. With two address instructions, and for binary operations, one address must do double duty as both an operand and a result. Thus, the instruction SUB Y, B carries out the calculation Y-B and stores the result in Y. The two-address format reduces the space requirement but also introduces some awkwardness. To avoid altering the value of an operand, a MOVE instruction is used to move one of the values to a result or temporary location before performing the operation. Our sample program expands to six instructions.

Simpler yet is the one-address instruction. For this to work, a second address must be implicit. This was common in earlier machines, with the implied address being a processor register known as the **accumulator** (AC). The accumulator contains one of the operands and is used to store the result. In our example, eight instructions are needed to accomplish the task.

It is, in fact, possible to make do with zero addresses for some instructions. Zero-address instructions are applicable to a special memory organization, called a *stack.* A stack is a last-in-first-out set of locations. The stack is in a known location and, often, at least the top two elements are in processor registers. Thus, zero address instructions would reference the top two stack elements.



Number of Addresses	Symbolic Representation	Interpretation
3	OP A, B, C	$A \leftarrow B$ OP C
2	OP A, B	$A \leftarrow A$ OP B
1	OP A	$AC \leftarrow AC$ OP A
0	OP	$T \leftarrow (T - 1)$ OP T

AC	= accumulator
T	= top of stack
(T − 1)	= second element of stack
A, B, C	= memory or register locations

Fig: Utilization of Instruction Addresses (Nonbranching Instructions)

Instruction Set Design

One of the most interesting, and most analyzed, aspects of computer design is instruction set design. The design of an instruction set is very complex because it affects so many aspects of the computer system. The instruction set defines many of the functions performed by the processor and thus has a significant effect on the implementation of the processor. The instruction set is the programmer's means of controlling the processor. Thus, programmer requirements must be considered in designing the instruction set.

It may surprise you to know that some of the most fundamental issues relating to the design of instruction sets remain in dispute. Indeed, in recent years, the level of disagreement concerning these fundamentals has actually grown. The most important of these fundamental design issues include the following:

- **Operation repertoire:** How many and which operations to provide, and how complex operations should be
- **Data types:** The various types of data upon which operations are performed
- **Instruction format:** Instruction length (in bits), number of addresses, size of various fields, and so on
- **Registers:** Number of processor registers that can be referenced by instructions, and their use
- **Addressing:** The mode or modes by which the address of an operand is specified

TYPES OF OPERANDS

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

• Addresses
• Numbers
• Characters
• Logical data

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. An important distinction between numbers used in ordinary mathematics and numbers stored in a computer is that the latter are limited. This is true in two senses. First, there is a limit to the magnitude of numbers representable on a machine and second, in the case of floating-point numbers, a limit to their precision. Thus, the programmer is faced with understanding the consequences of rounding, overflow, and underflow. 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.**

(Some Books often refer to this as binary coded decimal (BCD). Strictly speaking, BCD refers to the encoding of each decimal digit by a unique 4-bit sequence. Packed decimal refers to the storage of BCDencoded digits using one byte for each two digits)

With packed decimal, each decimal digit is represented by a 4-bit code, in the obvious way, with two digits stored per byte. Thus, 0 = 0000, 1 = 0001, …, 8 = 1000 and 9 = 1001. 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 (-).

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; see Appendix F). 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).

Logical Data

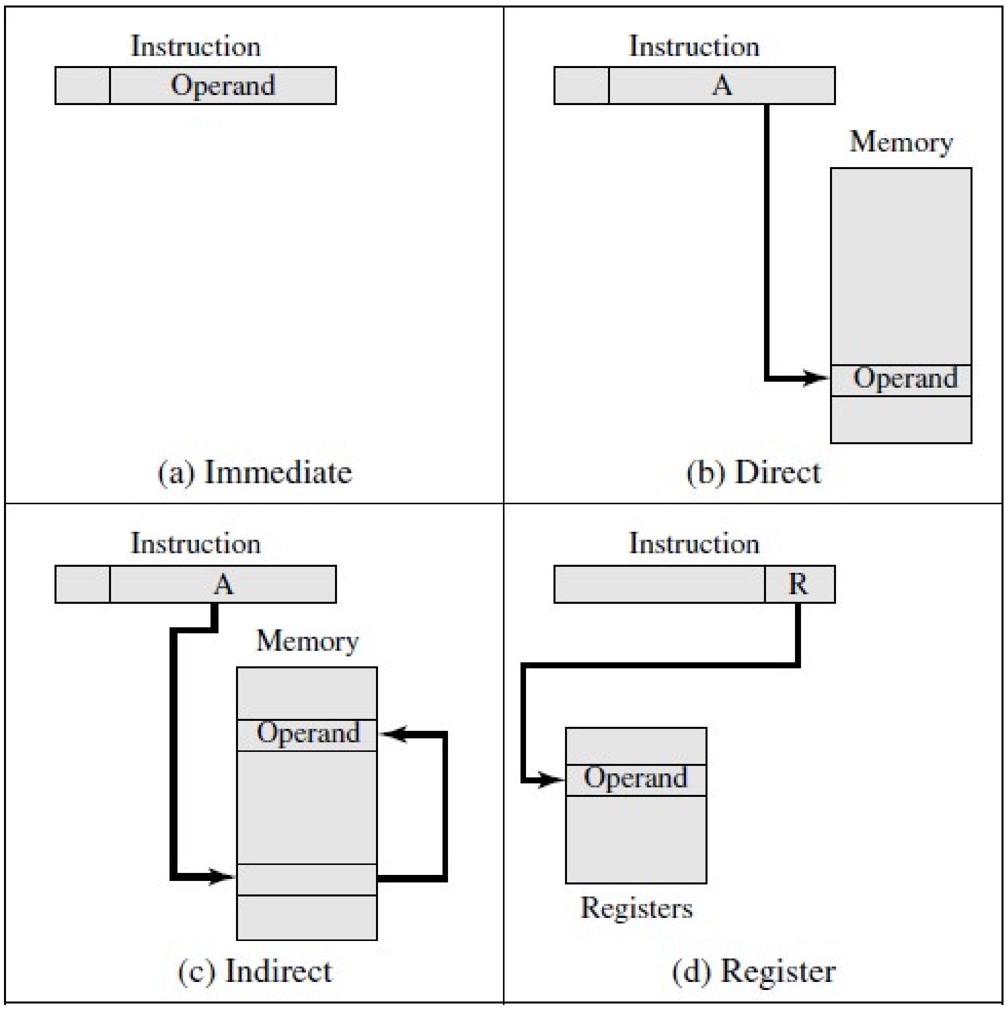
Normally, each word or other addressable unit (byte, halfword, and so on) is treated as a single unit of data. It is sometimes useful, however, to consider an n-bit unit as consisting of n 1-bit items of data, each item having the value 0 or 1.When data are viewed this way, they are considered to be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

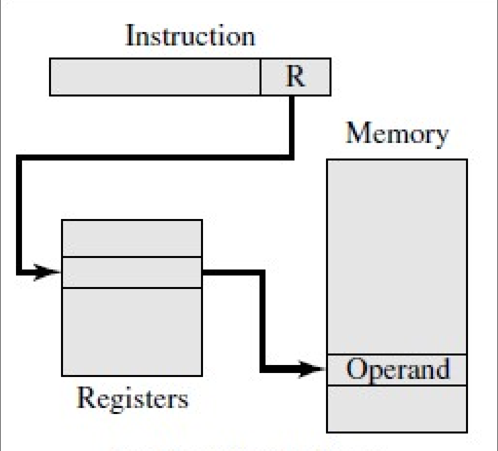
ADDRESSING

The address field or fields in a typical instruction format are relatively small. We would like to be able to reference a large range of locations in main memory or, for some systems, virtual memory. To achieve this objective, a variety of addressing techniques has been employed. They all involve some trade-off between address range and/or addressing flexibility, on the one hand, and the number of memory references in the instruction and/or the complexity of address calculation, on the other. In this section, we examine the most common addressing techniques:

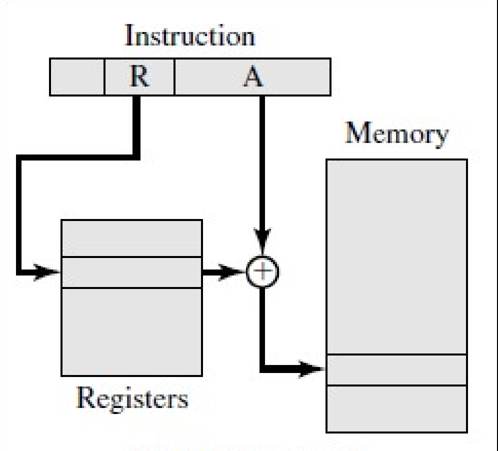
- Immediate
- Direct
- Indirect
- Register
- Register indirect
- Displacement
- Stack

Different Addressing modes:

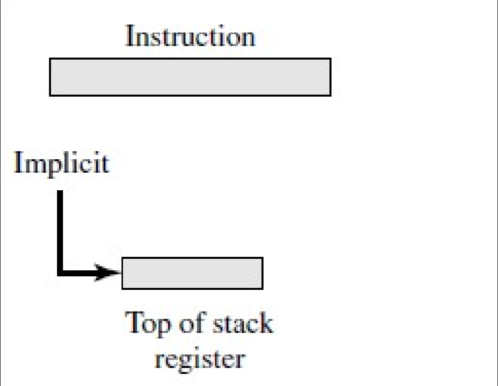




(e) Register indirect



(f) Displacement



(g) Stack

A = contents of an address field in the instruction
R = contents of an address field in the instruction that refers to a register
EA = actual (effective) address of the location containing the referenced operand
(X) = contents of memory location X or register X

Addressing modes comparison:

Mode	Algorithm	Principal Advantage	Principal Disadvantage
Immediate	Operand = A	No memory reference	Limited operand magnitude
Direct	EA = A	Simple	Limited address space
Indirect	EA = (A)	Large address space	Multiple memory references
Register	EA = R	No memory reference	Limited address space
Register indirect	EA = (R)	Large address space	Extra memory reference
Displacement	EA = A + (R)	Flexibility	Complexity
Stack	EA = top of stack	No memory reference	Limited applicability

Reference: **COMPUTER ORGANIZATION AND ARCHITECTURE *DESIGNING FOR PERFORMANCE, By William Stallings***