Unit 6 Instruction Sets: Addressing Modes and Formats

Structure:

6.1 Introduction

The operation of a CPU is determined by the instructions it executes. These instructions are referred to as machine instructions.

The complete collection of instructions that are executed by a CPU is referred to as CPU's **Instruction Set**. The instructions are usually represented in binary and the program is usually written in assembly language.

Objectives:

By the end of Unit 6, you should be able to:

1. Discuss the elements of a machine instruction.
2. Discuss the data types supported by IBM and VAX machines.
3. Discuss different addressing modes.
4. Explain with example different types of operations.

Instruction Characteristics

Elements of a Machine Instruction: Each instruction must contain the information required by the CPU for execution. Elements of a machine Instruction are:

- **Operation Code:** Specifies the operation to be performed. 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 the inputs for the operation.
- **Result Operand Reference:** The operation may produce a result.
- **Next Instruction Reference:** This tells the CPU where to fetch the next instruction after the execution of the current instruction is complete.

The next instruction to be fetched is located in main memory or in case of virtual memory system it can be either in main memory or secondary memory. 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. But when explicit reference is needed, then the main memory or virtual memory address must be supplied. The form in which the address is supplied is discussed in this section.

Source and Result operands can be in one of the three areas:

- **Main (or virtual) memory:** The memory address must be supplied.
- **CPU register:** CPU 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 number, and the instruction must contain the number of the desired register.

- **I/O device:** The instruction must specify the I/O module or device for the operation. If memory-mapped I/O is used, this is just another memory address.

Instruction Representation

Each instruction is represented by a sequence of bits. The instruction is divided into fields, corresponding to the constituent elements of the instruction. This layout of the instruction is called **Instruction Format**. With most instruction sets, more than one format is used. It is difficult for a programmer and the reader to deal with binary representations of machine instructions. Hence usually the instructions are written in symbolic representations of machine code using English like language called **mnemonics**.

Thus operation codes, in short Opcodes are represented using mnemonics. The format for this instruction is given in figure 6.1.

Opcode	Operand Reference	Operand Reference

Figure 6.1: A simple instruction format

In most modern CPU's, the first byte contains the opcode, sometimes including the register reference in some of the instructions. The operand references are in the following bytes (byte 2, 3, etc...).

Examples of few mnemonics:

Table 6.1: Examples of mnemonics

Mnemonics	***Description***
ADD	add
SUB	subtract
MUL	Multiply
LOAD	Load data from memory
MOV A, B	Move contents of B register to A register, where A & B are user visible registers of CPU

Example 1: An instruction with a register and memory address operands.

8 bits	8 bits	16/32 bits
Opcode	**Operand Ref.**	**Operand Reference**
	register reference	memory address

Example 2: An instruction may span to the second byte. Eg: 2 byte instruction can be as follows:

12 bits	4 bits
Opcode	**Operand Ref.**
	register reference

For example consider an instruction,

ADD R, Y

Assuming that it means that add the contents of data location Y of the memory to the contents of register R. The operation is performed on the data present in location Y and not on its address i.e., Y itself.

Previous to the execution of add instruction we give a list of specifications like X= 153, Y=154 and R=20. And the content of memory at location 153 is 10 and at 154 it is 20.

After the execution of the given ADD R, Y the result will be in R and will be equal to {R=20+[Y=154]=20 }=40.

Instruction Types

Example: High level language statement: X = X + Y

If we assume a simple set of machine instructions, this operation could be accomplished with three instructions: (assume X is stored in memory location 624, and Y in memory loc. 625.)

1. Load a register with the contents of memory location 624.

2. Add the contents of memory location 625 to the register,
3. Store the contents of the register in memory location 624.

As seen, a simple "C" (or BASIC) instruction may require 3 machine instructions.

As we have seen before, the instructions fall into one the following four categories:

- **Data processing:** Arithmetic and logic instructions.
- **Data storage:** Memory instructions.
- **Data movement:** I/O instructions.
- **Control:** Test and branch instructions.

Number of Addresses

What is the maximum number of addresses one might need in an instruction?

Virtually all arithmetic and logic operations are either unary (one operand) or binary (two operands). The result of an operation must be stored, suggesting a third address. Finally, after the completion of an instruction, the next instruction must be fetched, and its address is needed.

This line of reasoning suggests that an instruction could be required to contain 4 address references: two operands, one result, and the address of the next instruction. In practice, the address of the next instruction is handled by the **Program Counter (PC)**; therefore most instructions have one, two or three operand addresses. Three-address instruction formats are not common, because they require a relatively long instruction format to hold three address references.

Table 6.2: Utilization of Instruction Addresses (Non-branching Instructions):

Number of Addresses	***Symbolic Representation***	***Interpretation***
3	OP A, B, C	A <- B OP C
2	OP A, B	A <- A OP B
1	OP A	AC <- AC OP A, (AC is Aaccumulator)
0	OP	T <- T OP (T-1)

Example 3: Program to execute Y = (A – B) / (C + D x E):

Solution:

A) With One-Address Instructions: (requires an accumulator AC)

Table 6.3: Solution to ex.3

LOAD D	AC <- D
MUL E	AC <- AC x E
ADD C	AC <- AC + C
STOR Y	Y <- AC
LOAD A	AC <- A
SUB B	AC <- AC – B
DIV Y	AC <- AC / Y
STOR Y	Y <- AC

B) With Two-Address Instructions:

Table 6.4: Solution to ex.3

MOVE Y, A	Y <- A
SUB Y, B	Y <- Y – B
MOVE T, D	T <- D
MUL T, E	T <- T x E
ADD T, C	T <- T + C
DIV Y, T	Y <- Y / T

C) With Three-Address Instructions:

Table 6.5: Solution to ex.3

SUB Y, A, D	Y <- A – B
MUL T, D, E	T <- D x E
ADD T, T, C	T <- T + C
DIV Y, Y, T	Y <- Y / T

Instruction Set Design

The design of an instruction set is very complex, since it affects many different aspects of the computer system. The instruction set defines many of the functions performed by CPU. The instruction set is a programmer's means of implementation of the CPU.

The fundamental design issues in designing an instruction set are:

- **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, sizes of various fields, and so on.
- **Registers:** Number of CPU registers that can be referenced by instructions and their use.
- **Addressing:** The mode or modes by which the address of an operand is specified.

These issues are highly interrelated and must be considered together in designing an instruction set.

6.2 Types of Operands

Data Types

The most important general categories of data are:

- Addresses
- Numbers
- Characters
- Logical data.

Numbers

All machine language include numeric data types. Even in non numeric data processing, there is a need of numbers to act as counters, field widths, and so on. The numbers stored in computer are limited. That is there is a limit to the magnitude of numbers that can be represented on a machine. Also the programmer is faced with understanding the consequences of rounding, overflow, and underflow.

Three types of numerical data are commonly used in computers. They are:

1. Integer or fixed point
2. Floating point (real numbers)
3. Decimal (Remember BCD; an instruction set may be able to process BCD numbers.)

1. Integer data type:

For unsigned integers: It is a straight forward, simple binary representation. In an N-bit word the N-bits holds the magnitude of the numbers.

For example

00000000	=	0
00000001	=	1
00101001	=	41
10000000	=	128
11111111	=	255

For signed integers: It involves treating the Most Significant Bit (MSB) which is nothing but leftmost bit in a word as a sign bit. That is if the MSB is 1 the number is considered as negative else positive. In an N-bit word, the (N-1) bits holds the magnitude of the numbers.

For example

00000000	=	0
00000001	=	1
00010010	=	18
10010010	=	-18
11111111	=	-127

2. **Floating point numbers:** These numbers are represented as $\pm S * B^{\pm E}$

Where Sign: plus or minus

S is significant

B is base for binary it is 2

E is exponent

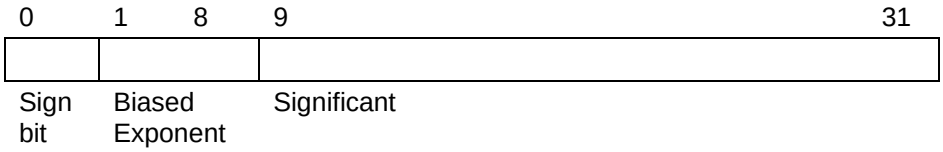


Figure 6.2: Format of floating point number

3. **Decimal:** Usual decimal system.

Characters

International Reference Alphabet (IRA) is referred to as ASCII in the USA. Each character in this code is represented by a unique 7-bit pattern, thus 128 different characters can be represented. Some of the patterns represent control characters. ASCII – encoded characters are usually stored and transferred as 8-bits per character. The eight bit may be set to 1 or 0 for even parity. EBCDIC character set is used in IBM 370 machines. It is an 8-bit code.

Logical Data

With logical data, memory can be used most efficiently for this storage. Logical values are also called as Boolean values which are 1 = true, 0 = false.

IBM 370 Data types

The IBM S/370 architecture provides the following data types.

- **Binary integer:** Binary integer may be either unsigned or signed. Signed binary integers are stored in 2's compliment form. Allowable lengths are 16 and 32 bits.
- **Floating point:** Floating point numbers of length 32, 64, and 128 bits are allowed. All use 7-bit exponent field.
- **Decimal:** Arithmetic on packed decimal integers is provided. The length is from 1 to 16 bytes. The rightmost 4 bits of the rightmost byte hold the sign. Hence signed numbers from 1 to 31 decimal digits can be represented.
- **Binary logical:** Operations are defined for data units of length 8, 32, and 64 bits. And variable length logical data of up to 256 bytes.
- **Character:** EBCDIC is used.

VAX Data types

The VAX provides an impressive array of data types. It is a byte oriented machine. All data types are in terms of bytes including 16-bit word, 32-bit long word, and the 64-bit quad-word, and even the 128 bit octa-word. The data type of VAX machine provides the following five types of data.

- **Binary integer:** Binary integers are usually considered as in 2's complement form. However they can be considered and operated on unsigned integers. Allowable lengths are 8, 16, 32, 64, and 128 bits.
- **Floating point:** It provides four different types of representations. They are

- F: 32 bits with an 8-bit exponent
- D: 64 bits with an 8-bit exponent
- G: 64 bits with an 11-bit exponent
- H: 128 bits with an 15-bit exponent

The F type is normal or default representation. D is usual double precision representation. G and H are provided for variety of applications to give successively increasing range and precision over F.

- **Decimal:** Arithmetic on packed decimal integers is provided. Two formats are provided.

Packed decimal strings: The length is from 1 to 16 bytes with 4 bits holding the sign.

Unpacked numeric strings: Stores 1 digit in ASCII representation, per byte with up to 31 bytes of length.

- **Variable bit field:** These are small integers packed together in large data unit. A bit field is specified by three operands: address of byte containing the start of field, starting bit position within the byte, the length in bits of the field. This data type is used to increase memory efficiency.
- **Character:** ASCII is used.

6.3 Types of Operations

A set of general types of operations is as follows:

Data transfer

It is the most fundamental type of machine instruction. The data transfer must specify

1) The location of the source and destination. Each location can be memory, register or top of stack.

2) The length of the data to be transferred.
3) The mode of addressing for each operand.

If both the operands are CPU registers, then the CPU simply causes data to be transferred from one register to another. This operation is internal to the CPU. If one or both operands are in memory, then the CPU must perform some or all of the following actions:

1. Calculate the memory address, based on the address mode.
2. If the address refers to virtual memory, translate from virtual to actual memory address.
3. Determine whether addressed item is in cache.
4. If not issue command to memory module.

For example: Move, Store, Load (Fetch), Exchange, Clear (Reset), Set, Push, Pop

Arithmetic

Most machines provide basic arithmetic functions like Add, Subtract, Multiply, and Divide. They are invariably provided for signed integer numbers. Often they are also provided for floating point and packed decimal numbers. Also some operations include only a single operand like Absolute, that takes only absolute value of the operand, Negate that takes the complement of the operands, Increment that increments the value of operand by 1, Decrement that decrements the value of operand by 1.

Logical

Machines also provide a variety of operations for manipulating individual bits of a word often referred to as *bit twiddling.* They are based on Boolean operations like AND, OR, NOT, XOR, Test, Compare, Shift, Rotate, Set control variables.

Conversion

These instructions are the ones that change the format or operate on the format of the data. Examples are Translate, Convert**.**

I/O, System control

There is a variety of approaches like isolated programmed I/O, memory mapped I/O, DMA and so on. Examples: Output (Write), Start I/O, Test I/O.

Transfer of Control

The instruction specifies the operation. In the normal course of events the next instruction to be performed is the one that immediately follows the current instruction in memory. But when the transfer of control type instruction occurs they specify the location of the next instruction that is to be executed. For example: Jump (Branch), Jump Conditional, Jump to Subroutine, Return, Execute, Skip, Skip Conditional, Halt, Wait (Hold), And No Operation.

System Control

These instructions are reserved for the use of Operating System. These instructions can only be executed while the processor is in a privileged state, or is executing a program in a special privileged area of memory.

6.4 Addressing Modes

In general, a program operates on data that reside in the computer's memory. These data can be organized in a variety of ways. If we want to keep track of students' names, we can write them in a list. If we want to associate information with each name, for example to record telephone numbers or marks in various courses, we may organize this information in the form of a table. Programmers use organizations called ***data structures*** to represent the data used in computations. These include lists, linked lists, arrays, queues, and so on.

Programs are normally written in a high-level language, which enables the programmer to use constants, local and global variables, pointers, and arrays. When translating a high-level language program into assembly language, the compiler must be able to implement these constructs using the facilities provided in the instruction set of the computer in which the program will be run. The different ways in which the location of an operand is specified in an instruction are referred to as ***addressing modes***.

There are the following addressing modes:

- Immediate Addressing
- Direct Addressing
- Indirect Addressing
- Register Addressing
- Register Indirect Addressing
- Displacement Addressing
- Stack Addressing

Notation:

A = Contents of an address field in the instruction.

R = Contents of an address field in the instruction that refers to a register.

EA = Effective (actual) Address of the location containing the referenced operand.

(X) = Contents of location X.

Table 6.6: Basic Addressing Modes

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 capability

Direct Addressing Mode

EA = A.

- Address field contains address of operand.
- Effective address (EA) = address field (A) e.g. ADD A
- Add contents of cell A to accumulator.
- Look in memory at address A for operand.
- Single memory reference to access data.
- No additional calculations to work out effective address.
- Limited address space

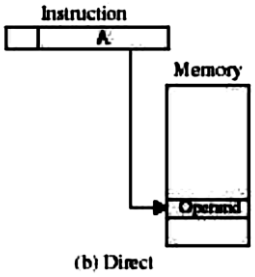


Figure 6.3

– The operand is in a memory location; the address of this location is given explicitly in the instruction. (In some assembly languages, this mode is called *Direct*.)

The instruction Move LOC, R2 uses these two modes. Processor registers are used as temporary storage locations where the data in a register are accessed using the Register mode. The Absolute mode can represent

global variables in a program. A declaration such as Integer A, B; in a high-level language program will cause the compiler to allocate a memory location to each of the variables A and B. Whenever they are referenced later in the program, the compiler can generate assembly language instructions that use the Absolute mode to access these variables. Next, let us consider the representation of constants. Address and data constants can be represented in assembly language using the immediate mode.

Immediate Addressing Mode

The operand is actually present in the instruction.

Operand = A

Can be used to define and use constants, or set initial values.

- Operand is part of instruction
- Operand = address field
- e.g. ADD 5
- Add 5 to contents of accumulator
- 5 is operand
- No memory reference to fetch data
- Fast
- Limited range

Figure 6.4: Immediate

For example, the instruction Move 200*immediate,* R0 places the value 200 in register R0. Clearly, the immediate mode is only used to specify the value of a source operand. Using a subscript to denote the immediate mode is not appropriate in assembly languages. A common convention is to use the sharp sign (#) in front of the value to indicate that this value is to be used as an immediate operand.

Hence, we write the instruction above in the form Move # 200, R0. Constant values are used frequently in high-level language programs. For example, the statement

A = B + 6 contains the constant 6. Assuming that A and B have been declared earlier as variables and may be accessed using the Absolute mode; this statement may be compiled as follows:

Move B, R1

Add #6, R1

Move R1, A

Constants are also used in assembly language to increment a counter, test for some bit pattern, and so on.

Indirect Addressing Mode

- Memory cell pointed to by address field contains the address of (pointer to) the operand

  EA = (A)
- Look in A, find address (A) and look there for operand

  e.g. ADD (A)
- Add contents of cell pointed to by contents of A to accumulator
- Large address space
- 2n where n = word length
- May be nested, multilevel, cascaded

  e.g. EA = (((A))) Draw the diagram yourself
- Multiple memory accesses to find operand

  Hence slower

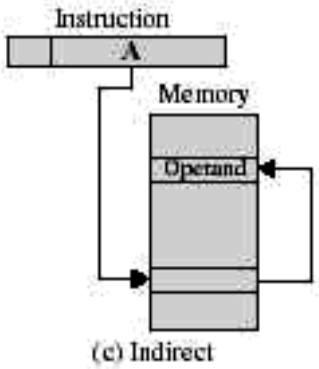


Figure 6.5

Indirect mode – The effective address of the operand is the contents of a register or memory location whose address appears in the instruction.

We denote indirection by placing the name of the register or the memory address given in the instruction in parentheses as illustrated in Figure and Table.

To execute the Add instruction in Figure 6.6a the processor uses the value B, which is in register R1, as the effective address of the operand. It requests a read operation from the memory to read the contents of location B. The value read is the desired operand, which the processor adds to the contents of register R0. Indirect addressing through a memory location is also possible as shown in Figure 6.6b. In this case, the processor first reads the contents of memory location A,

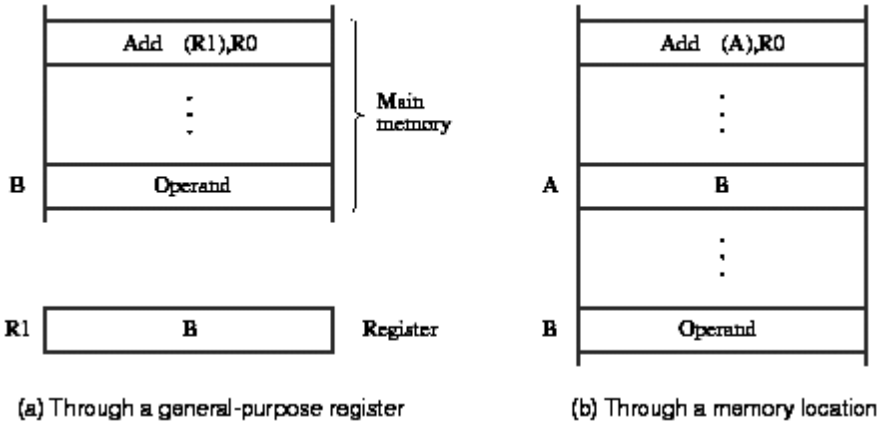


Figure 6.6

Address	Contents		
	Move	N,R1	Initialization
	Move	#NUM1,R2	
	Clear	R0	
LOOP	Add	(R2),R0	
	Add	#4,R2	
	Decrement	R1	
	Branch>0	LOOP	
	Move	R0,SUM	

Figure 6.7

then requests a second read operation using the value B as an address to obtain the operand. The register or memory location that contains the

address of an operand is called a *pointer*. Indirection and the use of pointers are important and powerful concepts in programming. Consider the analogy of a treasure hunt: In the instructions for the hunt you may be told to go to a house at a given address. Instead of finding the treasure there, you find a note that gives you another address where you will find the treasure.

By changing the note, the location of the treasure can be changed, but the instructions for the hunt remain the same. Changing the note is equivalent to changing the contents of a pointer in a computer program. For example, by changing the contents of register R1 or location A in Figure the same Add instruction fetches different operands to add to register R0.

For adding a list of numbers, indirect addressing can be used to access successive numbers in the list, resulting in the program shown in Figure 6.7. Register R2 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R2. The initialization section of the program loads the counter value *n* from memory location N into R1 and uses the immediate addressing mode to place the address value NUM1, which is the address of the first number in the list, into R2. Then it clears R0 to 0.

The instruction, Add (R2), R0 fetches the operand at location NUM1 and adds it to R0. The second Add instruction adds 4 to the contents of the pointer R2, so that it will contain the address value NUM2 when the above instruction is executed in the second pass through the loop.

Consider the C-language statement A=&B; where B is a pointer variable. This statement may be compiled into Move B, R1

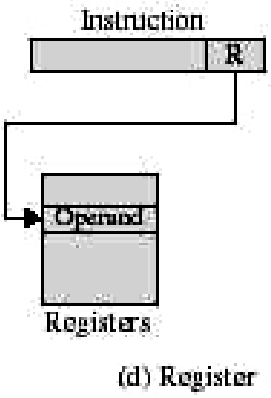
Move (R1), A

Using indirect addressing through memory, the same action can be achieved with Move (B), A. Despite its apparent simplicity, indirect addressing through memory has proven to be of limited usefulness as an addressing mode, and it is seldom found in modern computers.

Indirect addressing through registers is used extensively. The program in Figure shows the flexibility it provides. Also, when absolute addressing is not available, indirect addressing through registers makes it possible to access global variables by first loading the operand's address in a register.

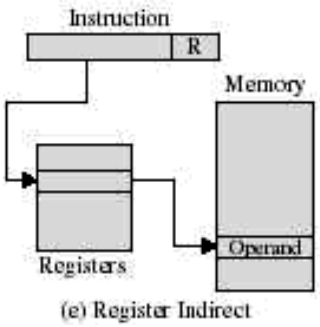
Register Addressing Mode

- Operand is held in register named in address field
- EA = R
- Limited number of registers
- Very small address field needed
- Shorter instructions
- Faster instruction fetch
- No memory access
- Very fast execution
- Very limited address space
- Multiple registers helps performance
- Requires good assembly programming or compiler writing
- N.B. C programming
- register into a;



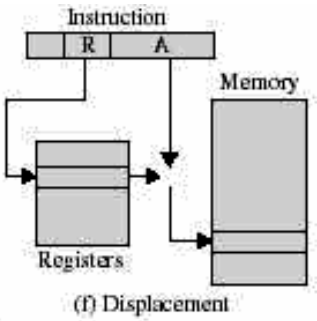
(d) Register

Figure 6.8



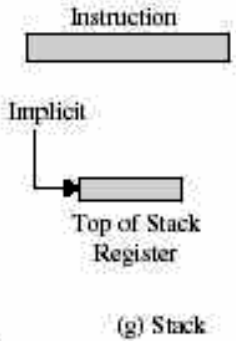
(e) Register Indirect

Figure 6.9



(f) Displacement

Figure 6.10



(g) Stack

Figure 6.11

Register Indirect Addressing Mode

- Register indirect addressing
- EA = (R)
- Operand is in memory cell pointed to by contents of register R
- Large address space (2n)
- Fewer memory access than indirect addressing

Indexing and pointers

The next addressing mode we discuss provides a different kind of flexibility for accessing operands. It is useful in dealing with lists and arrays.

Index mode – The effective address of the operand is generated by adding a constant value to the contents of a register. The register used may be either a special register provided for this purpose, or, more commonly, it may be any one of a set of general-purpose registers in the processor. In either case, it is referred to as an ***index register***. We indicate the Index mode symbolically as X(R*i)* where X denotes the constant value contained in the instruction and R*i* is the name of the register involved. The effective address of the operand is given by EA = X + [R*i*]. The contents of the index register are not changed in the process of generating the effective address.

In an assembly language program, the constant X may be given either as an explicit number or as a symbolic name representing a numerical value. When the instruction is translated into machine code, the constant X is given as a part of the instruction and is usually represented by fewer bits than the word length of the computer. Since X is a signed integer, it must be sign-extended to the register length before being added to the contents of the register.

Figure 6.12 illustrates two ways of using the Index mode. In Figure 6.12a, the index register, R1, contains the address of a memory location, and the

value X defines an ***offset*** (also called a ***displacement***) from this address to the location where the operand is found.

An alternative use is illustrated in Figure 6.12b. Here, the constant X corresponds to a memory address, and the contents of the index register define the offset to the operand. In either case, the effective address is the sum of two values; one is given explicitly in the instruction, and the other is stored in a register.

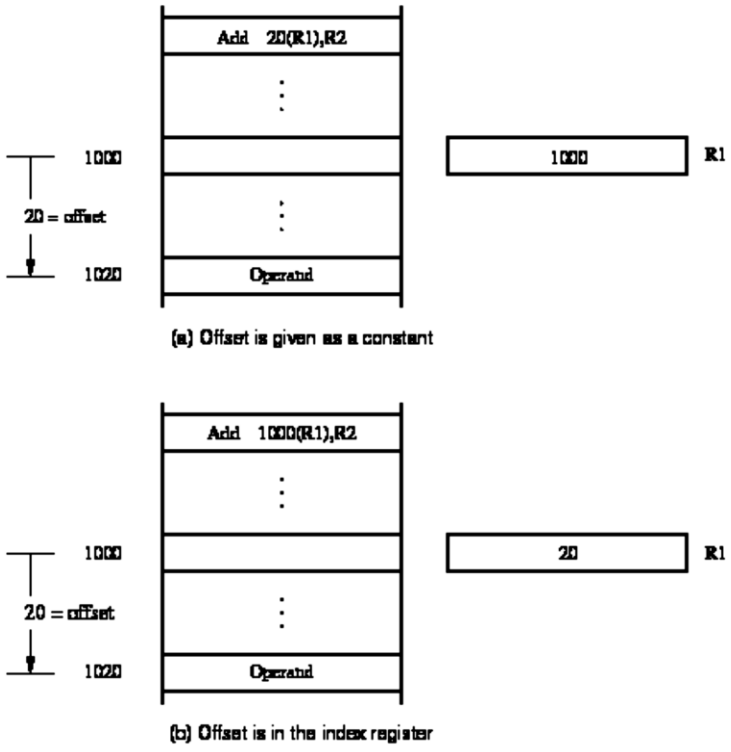


Figure 6.12

Displacement Addressing Mode

- EA = A + (R)
- Address field hold two values
- A = base value

- R = register that holds displacement
- or vice versa

Relative Addressing Mode

- A version of displacement addressing
- R = Program counter, PC
- EA = A + (PC)
- i.e. get operand from A cells from current location pointed to by PC

Base-Register Addressing

- A holds displacement
- R holds pointer to base address
- R may be explicit or implicit
- e.g. segment registers in 80x86

Indexing

- A = base
- R = displacement
- EA = A + R
- Good for accessing arrays
- EA = A + R
- R++

Stack Addressing

- Operand is (implicitly) on top of stack
- e.g. ADD Pop top two items from stack and add

Other additional addressing modes

The two modes described next are useful for accessing data items in successive locations in the memory.

Autoincrement mode – The effective address of the operand is the contents of a register specified in the instruction. After accessing the

operand, the contents of this register are automatically incremented to point to the next item in a list.

We denote the Autoincrement mode by putting the specified register in parentheses, to show that the contents of the register are used as the effective address, followed by a plus sign to indicate that these contents are to be incremented after the operand is accessed. Thus, the Autoincrement mode is written as *(*R*i)*+.

6.5 Instruction Formats

Instruction Format is defined as the layout of bits in an instruction in terms of its constituent parts. An Instruction Format must include opcode implicitly or explicitly and one or more operand(s). For, most instruction sets have usually more than one instruction format.

Instruction Length

Most important design issue is the length of an instruction. It is affected by and affects Memory size, Memory organization, Bus structure, CPU complexity, CPU speed. There is a trade off between powerful instruction repertoire and saving space.

Apart from this tradeoff there are other considerations. Either the instruction length should be equal to the memory transfer length or one should be a multiple of the other. A related consideration is the memory transfer rate.

Allocation of Bits

The factors that determine the use of addressing bits are:

- **Number of addressing modes:** Some times addressing mode is implicit in the instruction or may be certain opcodes call for indexing. In other cases the addressing mode must be explicit and one or more bits are needed.

- **Number of operands:** Typically today's machines provide two operands. Each operand may require its own mode indicator or the use of indicator is limited to one of the address fields.
- **Register versus memory:** A machine must have registers so that the data can be brought into the CPU for processing. One operand address is implicit. The more the registers are used to specify the operands less the number of bits needed.
- **Number of register sets:** Almost all machines have a set of general purpose registers, with typically 8 or 16 registers in it. These registers can be used to store data or addresses for displacement addressing etc.
- **Address range:** For addresses that refer to the memory locations, the range of addresses is related to the number of address bits. Because of this limitation direct addressing is rarely used.
- **Address granularity:** It is concerned with addresses that refer to the memory other than registers. In a system with 16 or 32 bit words, an address can refer to a word or a byte at the designer's choice. Byte addressing is convenient for character manipulation but requires fixed size memory, and hence more address bits.

Variable-Length Instruction

The designer might provide a variety of instruction formats of different lengths. Addressing can be more flexible, with various combinations of registers and memory references plus addressing modes.

The price that needs to be paid is an increase in the complexity of the CPU.

1. Due to varying number of operands,
2. Due to varying lengths of opcode in some CPUs.

6.6 Stacks and Subroutines

Stacks

A computer program often needs to perform a particular subtask using the familiar subroutine structure. In order to organize the control and information linkage between the main program and the subroutine, a data structure called a stack is used.

This section will describe stacks, as well as a closely related data structure called a queue.

Data operated on by a program can be organized in a variety of ways. We have already encountered a data structure called list. Now, we consider an important data structure known as a stack. A ***stack*** is a list of data elements, usually words or bytes, with the accessing restriction that elements can be added or removed at one end of the stack only. This end is called the ***top*** of the stack, and the other end is called the ***bottom***.

The structure is sometimes referred to as a ***pushdown*** stack. Imagine a pile of trays in a cafeteria; customers pick up new trays from the top of the pile, and clean trays are added to the pile by placing them onto the top of the pile. Another descriptive phrase, ***last-in-first-out*** (LIFO) stack, is also used to describe this type of storage mechanism; the last data item placed on the stack is the first one removed when retrieval begins. The terms ***push*** and ***pop*** are used to describe placing a new item on the stack and removing the top item from the stack, respectively.

Data stored in the memory of a computer can be organized as a stack, with successive elements occupying successive memory locations. Assume that the first element is placed in location BOTTOM, and when new elements are pushed onto the stack, they are placed in successively lower address locations. We use a stack that grows in the direction of decreasing memory addresses in our discussion, because this is a common practice.

Figure 6.13 shows a stack of word data items in the memory of a computer. It contains numerical values, with 43 at the bottom and 28 at the top. A processor register is used to keep track of the address of the element of the stack that is at the top at any given time. This register is called the ***stack pointer*** **(SP)**. It could be one of the general-purpose registers or a register dedicated to this function. If we

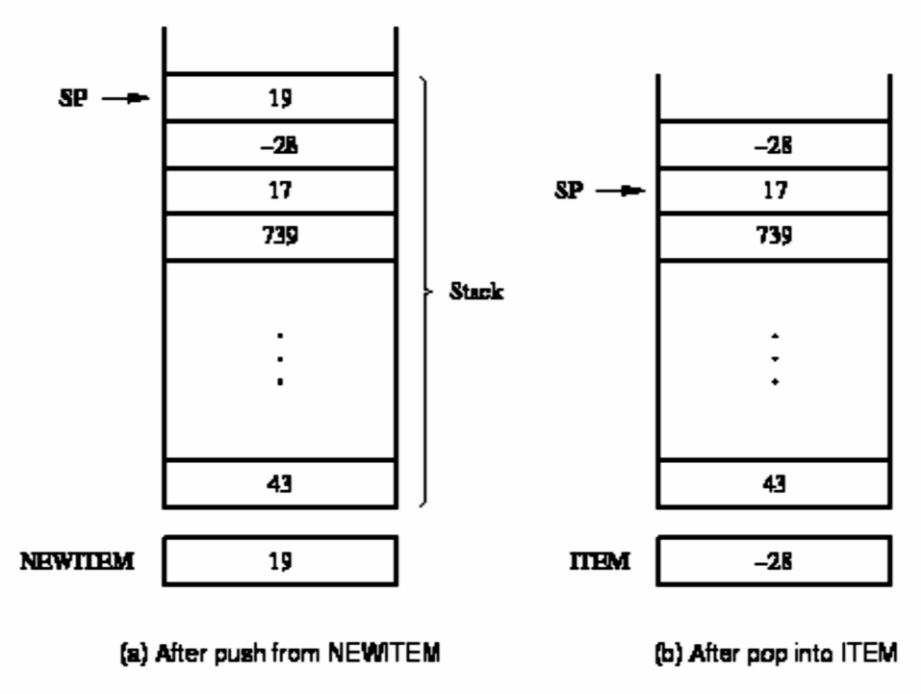


Figure 6.13

assume a byte-addressable memory with a 32-bit word length; the push operation can be implemented as

Subtract #4, SP

Move NEWITEM, (SP)

where the Subtract instruction subtracts the source operand 4 from the destination operand contained in SP and places the result in SP. These two instructions move the word from location NEWITEM onto the top of the

stack, decrementing the stack pointer by 4 before the move. The pop operation can be implemented as

Move (SP), ITEM

Add #4, SP

These two instructions move the top value from the stack into location ITEM and then increment the stack pointer by 4 so that it points to the new top element. Figure 6.14 shows the effect of each of these operations on the stack in Figure 6.13. If the processor has the Autoincrement and Autodecrement addressing modes, then the push operation can be performed by the single instruction Move NEWITEM,.(SP) and the pop operation can be performed by Move (SP)+, ITEM.

When a stack is used in a program, it is usually allocated a fixed amount of space in the memory. In this case, we must avoid pushing an item onto the stack when the stack has reached its maximum size. Also, we must avoid attempting to pop an item off an empty stack, which could result from a programming error. Suppose that a stack runs from location 2000 (BOTTOM) down no further than location 1500. The stack pointer is loaded initially with the address value 2004. Recall that SP is decremented by 4 before new data are stored on the stack. Hence, an initial value of 2004 means that the first item pushed onto the stack will be at location 2000. To prevent either pushing an item on a full stack or popping an item off an empty stack, the single-instruction push and pop operations can be replaced by the instruction sequences shown in Figure 6.15.

Compare instruction: Compare src, dst performs the operation [dst] . [src] and sets the condition code flags according to the result. It does not change the value of either operand.

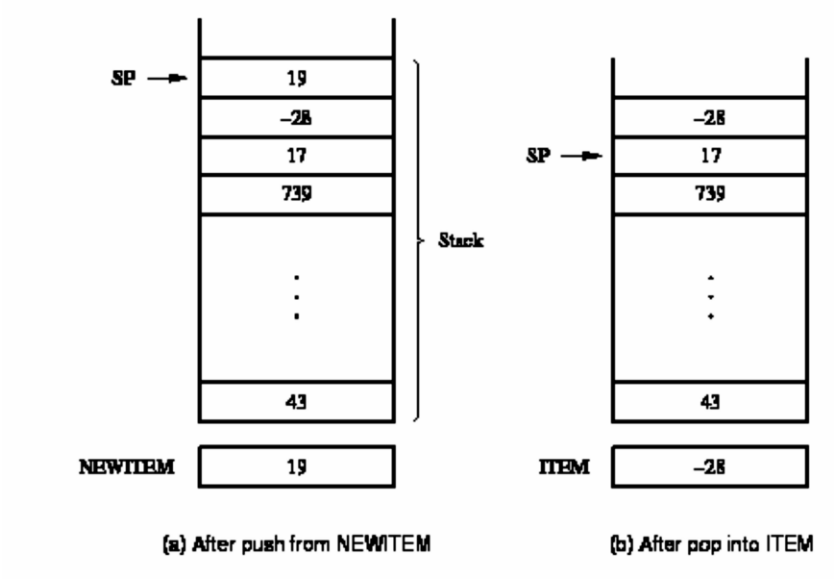


Figure 6.14

SAFEPOP	Compare Branch>0	#2000,SP EMPTYERROR	Check to see if the stack pointer contains an address value greater than 2000. If it does, the stack is empty. Branch to the routine EMPTYERROR for appropriate action.
	Move	(SP)+,ITEM	Otherwise, pop the top of the stack into memory location ITEM.

(a) Routine for a safe pop operation

SAFEPUSH	Compare Branch≤0	#1500,SP FULLERROR	Check to see if the stack pointer contains an address value equal to or less than 1500. If it does, the stack is full. Branch to the routine FULLERROR for appropriate action.
	Move	NEWITEM,−(SP)	Otherwise, push the element in memory location NEWITEM onto the stack.

(b) Routine for a safe push operation

Figure 6.15

Subroutines

In a given program, it is often necessary to perform a particular subtask many times on different data values. Such a subtask is usually called a ***subroutine***. For example, a subroutine may evaluate the *sine* function or sort a list of values into increasing or decreasing order. It is possible to include the block of instructions that constitute a subroutine at every place where it is needed in the program. However, to save space, only one copy of the instructions that constitute the subroutine is placed in the memory, and any program that requires the use of the subroutine simply branches to its starting location. When a program branches to a subroutine we say that it is *calling* the subroutine. The instruction that performs this branch operation is named a Call instruction.

After a subroutine has been executed, the calling program must resume execution, continuing immediately after the instruction that called the subroutine. The subroutine is said to *return* to the program that called it by executing a Return instruction. Since the subroutine may be called from different places in a calling program, provision must be made for returning to the appropriate location. The location where the calling program resumes execution is the location pointed to by the updated PC while the Call instruction is being executed. Hence, the contents of the PC must be saved by the Call instruction to enable correct return to the calling program.

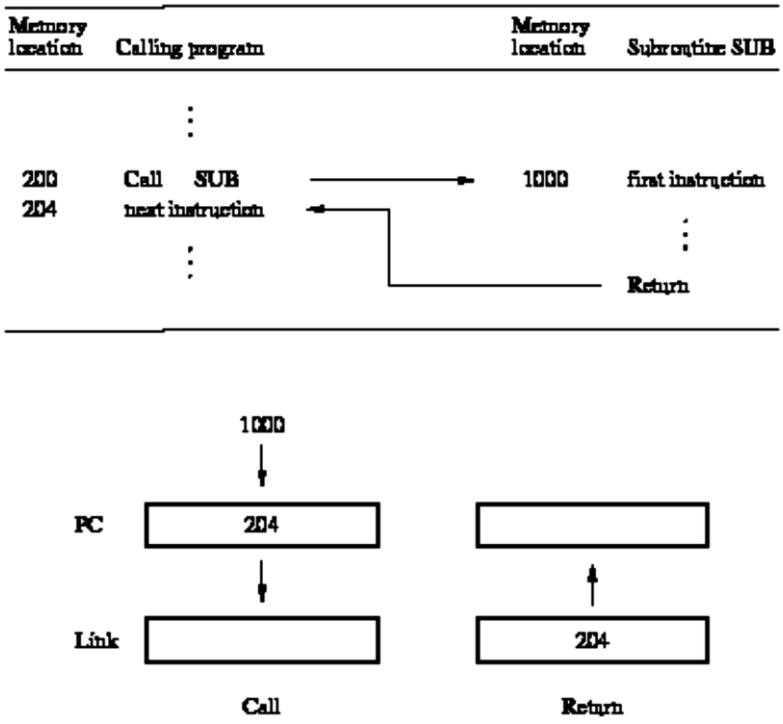


Figure 6.16: An example of CALL and RETURN

The way in which a computer makes it possible to call and return from subroutines is referred to as its *subroutine linkage* method. The simplest subroutine linkage method is to save the return address in a specific location, which may be a register dedicated to this function. Such a register is called the *link register*. When the subroutine completes its task, the Return instruction returns to the calling program by branching indirectly through the link register.

The Call instruction is just a special branch instruction that performs the following operations:

- Store the contents of the PC in the link register
- Branch to the target address specified by the instruction

The Return instruction is a special branch instruction that performs the operation:

Branch to the address contained in the link register. Figure 6.16 illustrates this procedure.

6.7 Summary

This chapter covers the representation of instructions and discusses Arithmetic and logic instructions, Memory instructions, I/O instructions, and Test and branch instructions. We have also seen different types of addressing modes like direct, indirect, immediate and register, including the important concepts of pointers and indexed addressing. Here we also see the different data types like Addresses, Numbers, Characters, and Logical data. Numbers can be integers or decimal (floating point). As an example we have studied data types for two machines IBM and VAX. Finally the concept of subroutine with the application of the stack data structure is discussed.

Self Assessment Questions

1. _____ specifies the operation to be performed.
2. Opcodes are represented using _____.
3. Floating point numbers are _____.
4. _____ is referred as ASCII in USA.
5. _____ instructions are reserved for the use of operating system.

6.8 Terminal Questions

1. Define the following:
   a. Op-code.
   b. Machine instruction.
2. Discuss the different categories of instructions.

3. Write a note on different data types with examples for each.
4. Give the details of Data types specified for VAX & IBM 370 machines.
5. Discuss in detail, with examples, the addressing modes that do not use memory.
6. Explain the calculation of effective address in case of Displacement addressing and relative addressing modes.
7. Write a note on:
   a. Stack
   b. Subroutines

6.9 Answers

Self Assessment Questions:

1. operation code or opcode
2. mnemonics
3. real numbers
4. international reference alphabet (IRA)
5. system control

Terminal Questions:

1. Refer Section 6.1
2. Refer Section 6.1
3. Refer Section 6.2
4. Refer Section 6.2
5. Refer Section 6.4
6. Refer Section 6.4
7. Refer Section 6.6