Module I

BASIC OPERATIONAL CONCEPTS

- The processor contains ***Arithmetic Logic Unit (ALU)***, control-circuitry and many registers.
- The ***Instruction Register (IR)*** holds the instruction that is currently being executed.
- The instruction is then passed to the ***Control Unit (CU)***, which generates the timing-signals that determine when a given action is to take place
- The ***Program Counter (PC)*** contains the memory-address of the next-instruction to be fetched & executed.
- During the execution of an instruction, the contents of PC are updated to point to next instruction.
- The processor also contains ***N General Purpose Registers*** R0 through Rn-1, which is used to store temporary results, operands, memory address etc.
- The ***Memory Address Register (MAR)*** holds the address of the memory-location to be accessed.
- The ***Memory Data Register (MDR)*** contains the data to be written into or read out of the addressed location.

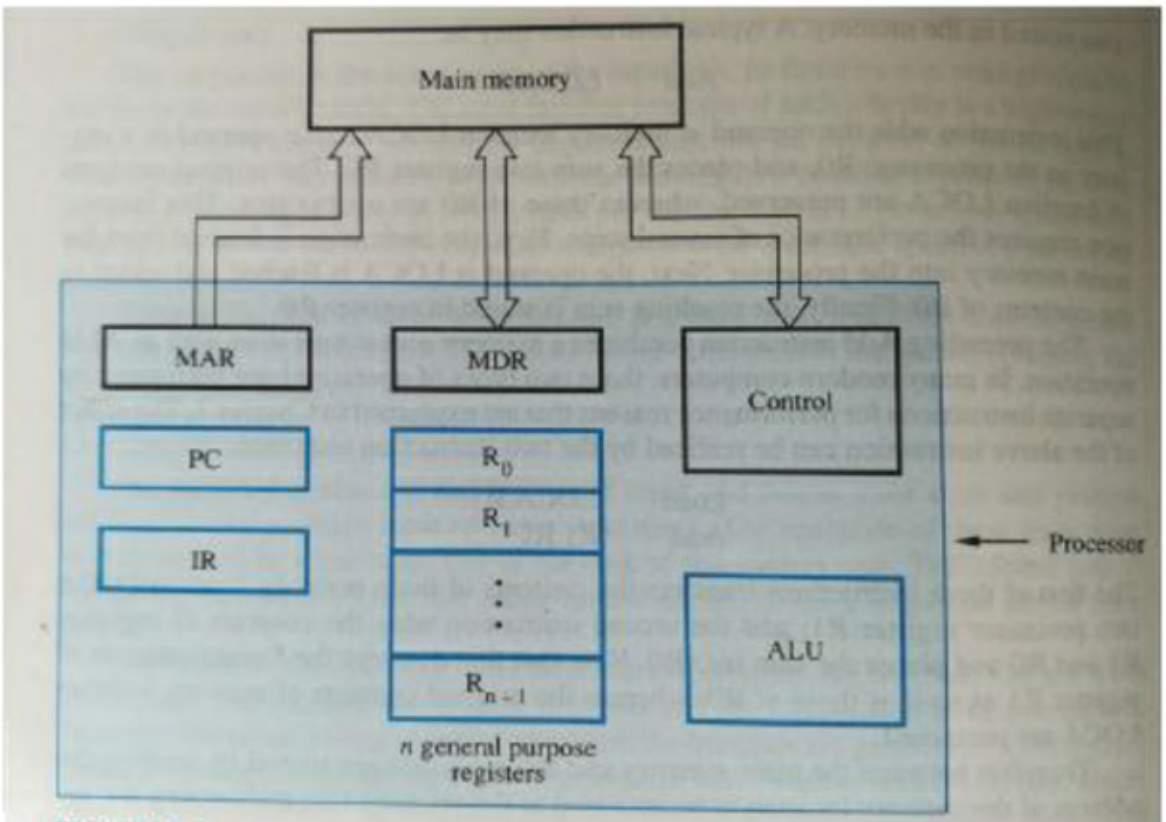


FIGURE 1.3
Connections between the processor and the main memory.

Following are the steps that take place to execute an instruction

Load LOCA, R1

Add R1, R0

1. The address of first instruction (to be executed) gets loaded into PC.

2. The contents of PC (i.e. address) are transferred to the MAR & control-unit issues Read signal to memory.
3. After certain amount of elapsed time, the first instruction is read out of memory and placed into MDR.
4. Next, the contents of MDR are transferred to IR. At this point, the instruction can be decoded & executed.
5. To fetch an operand, it's address is placed into MAR & control-unit issues Read signal. As a result, the operand is transferred from memory into MDR, and then it is transferred from MDR to ALU.
6. Likewise required number of operands is fetched into processor.
7. Finally, ALU performs the desired operation.
8. If the result of this operation is to be stored in the memory, then the result is sent to the MDR.
9. The address of the location where the result is to be stored is sent to the MAR and a Write cycle is initiated.

At some point during execution, contents of PC are incremented to point to next instruction in the program. [The instruction is a combination of opcode and operand].

BUS STRUCTURE

- A ***bus*** is a group of lines that serves as a connecting path for several devices.
- Bus must have lines for data transfer, address & control purposes.
- Because the bus can be used for only one transfer at a time, only 2 units can actively use the bus at any given time.
- Bus control lines are used to arbitrate multiple requests for use of the bus.
- Main advantage of single bus: Low cost and flexibility for attaching peripheral devices.
- Systems that contain multiple buses achieve more concurrency in operations by allowing 2 or more transfers to be carried out at the same time. Advantage: better performance. Disadvantage: increased cost.
- The devices connected to a bus vary widely in their speed of operation. To synchronize their operational speed, the approach is to include buffer registers with the devices to hold the information during transfers.
- Buffer registers prevent a high-speed processor from being locked to a slow I/O device during a sequence of data transfers.

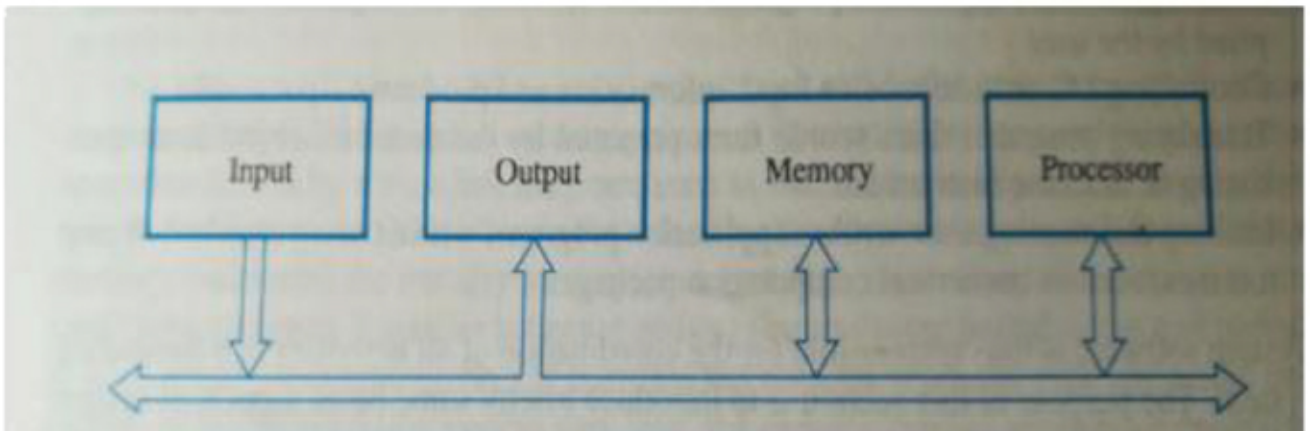


FIGURE 1.4
Single-bus structure.

PERFORMACNE

Cache

- Cache is a fastest memory between main memory and processor.
- Processor time to execute a program depends on the hardware involved in the execution of individual machine instructions.
- The processor and a relatively small cache memory can be fabricated on a single integrated circuit chip

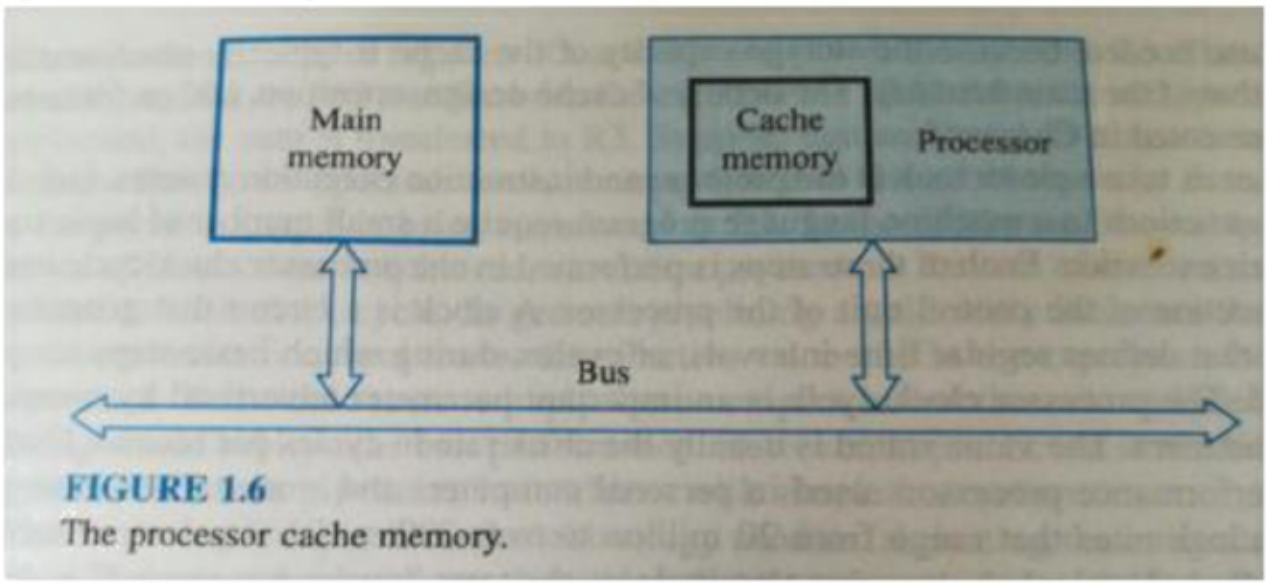


FIGURE 1.6
The processor cache memory.

PROCESSOR CLOCK

- Processor circuits are controlled by a timing signal called a clock.
- The clock defines regular time intervals called *clock cycles.*
- To execute a machine instruction, the processor divides the action to be performed into a sequence of basic steps such that each step can be completed in one clock cycle.
- Let **P**=length of one clock cycle R=clock rate. Relation between **P** and **R** is given by

$$\mathbf{R}=\frac{\mathbf{1}}{P}$$

This is measured in cycles per second.

- Cycles per second is also called hertz(Hz)

BASIC PERFORMANCE EQUATION

- Let
- **T**=processor time required to executed a program N=actual number of instruction executions
- **S**=average number of basic steps needed to execute one machine instruction R=clock rate in cycles per second
- The program execution time is given by

$$T = \frac{N \times S}{R}$$

- *To achieve high performance, the computer designer must reduce the value of T, which means reducing N and S, and increasing R.*
  - → The value of N is reduced if source program is compiled into fewer machine instructions.
  - → The value of S is reduced if instructions have a smaller number of basic steps to perform.
  - → The value of R can be increased by using a higher frequency clock.
- Care has to be taken while modifying the values since changes in one parameter may affect the other.

Problems:

1. A program contains 1000 instruction Out of that 25% instructions requires 4 clock cycles. 40% instructions require 5 clocks and remaining 3 clock cycles for execution. Find the total time required to execute the program running in a 1GHz machine. (5 Marks)

Solution:

N=1000

25% of N= 250 instructions require 4 clock cycles,

40% of N =400 instructions require 5 clock cycles,

35% of N=350 instructions require 3 clock cycles

T = (N*S)/R =

250*4+400*5+350*3/1*10^9

=1000*(1000+2000+1050)/1*10^9

=

2. The Effective value of S is 1.25 and the average value of N is 200. If the clock rate is 500MHz, calculate the total program execution time required.

T = (N*S)/R

= (200*1.25)/5*10^6

= (250)/5*10^6

=

PERFORMANCE MEASUREMENT

- ***SPEC*** *(System Performance Evaluation Corporation)* selects & publishes the standard programs along with their test results for different application domains
- The SPEC rating is computed as follows

$$\text{SPEC rating} = \frac{\text{Running time on the reference computer}}{\text{Running time on the computer under test}}$$

- If SPEC rating=50 means that the computer under test is 50times as fast as reference computer.
- The test is repeated for all the programs in the SPEC suite, and the geometric mean of the results is computed.
- Let $SPEC_i$ be the rating for program i in the suite. The overall SPEC rating for the computer is given by

Where **n**=number of programs in the suite

$$\text{SPEC rating} = \left(\prod_{i=1}^{n} \text{SPEC}_i\right)^{\frac{1}{n}}$$

MEMORY LOCATIONS & ADDRESSES

- The memory consists of many millions of storage cells (flip-flops), each of which can store a bit of information having the value 0 or 1 (Figure 2.5).
- Each group of n bits is referred to as a word of information, and n is called the word length.
- The word length can vary from 8 to 64bits.
- A unit of 8 bits is called a byte.

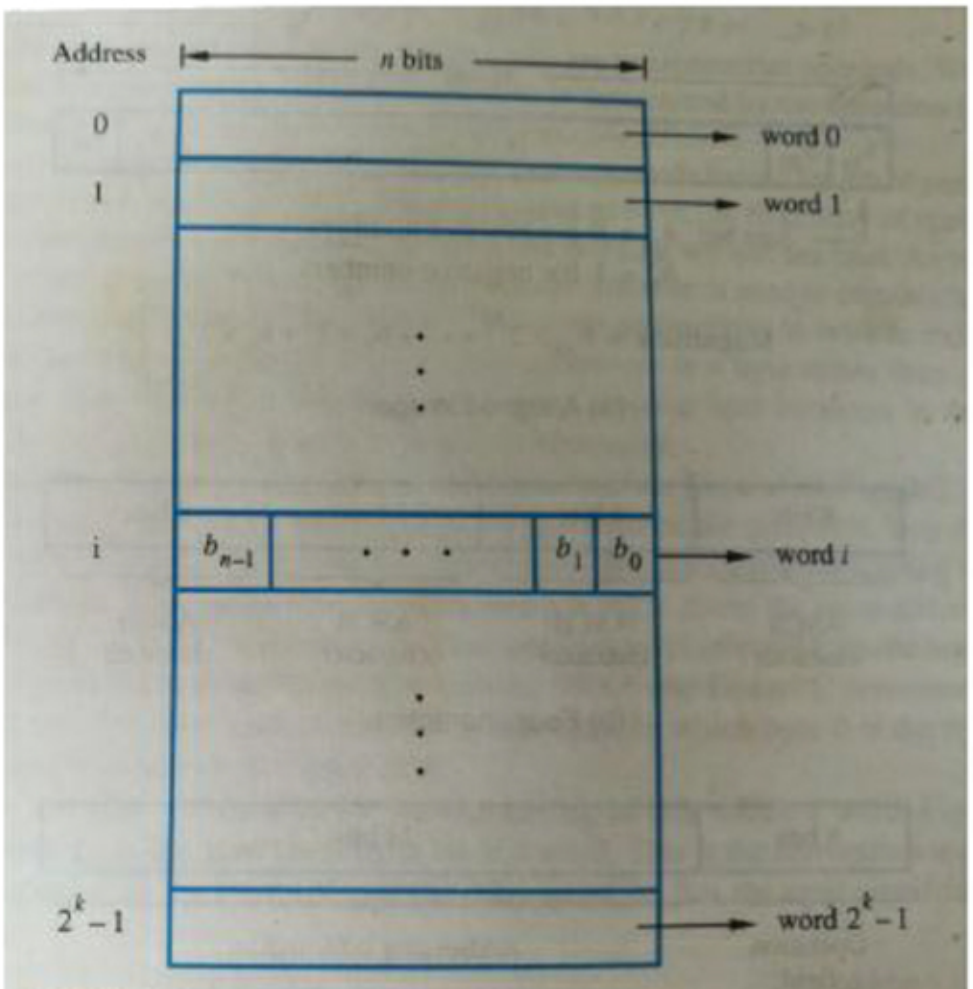


FIGURE 2.1
Main memory addresses.

- Accessing the memory to store or retrieve a single item of information (either a word or a byte) requires distinct addresses for each item location. (It is customary to use numbers from 0 through 2^k-1 as the addresses of successive locations in the memory).
- If 2^k=number of addressable locations, then 2^k addresses constitute the address space of the computer. For example, a 24-bit address generates an address space of 2^{24} locations (16MB).

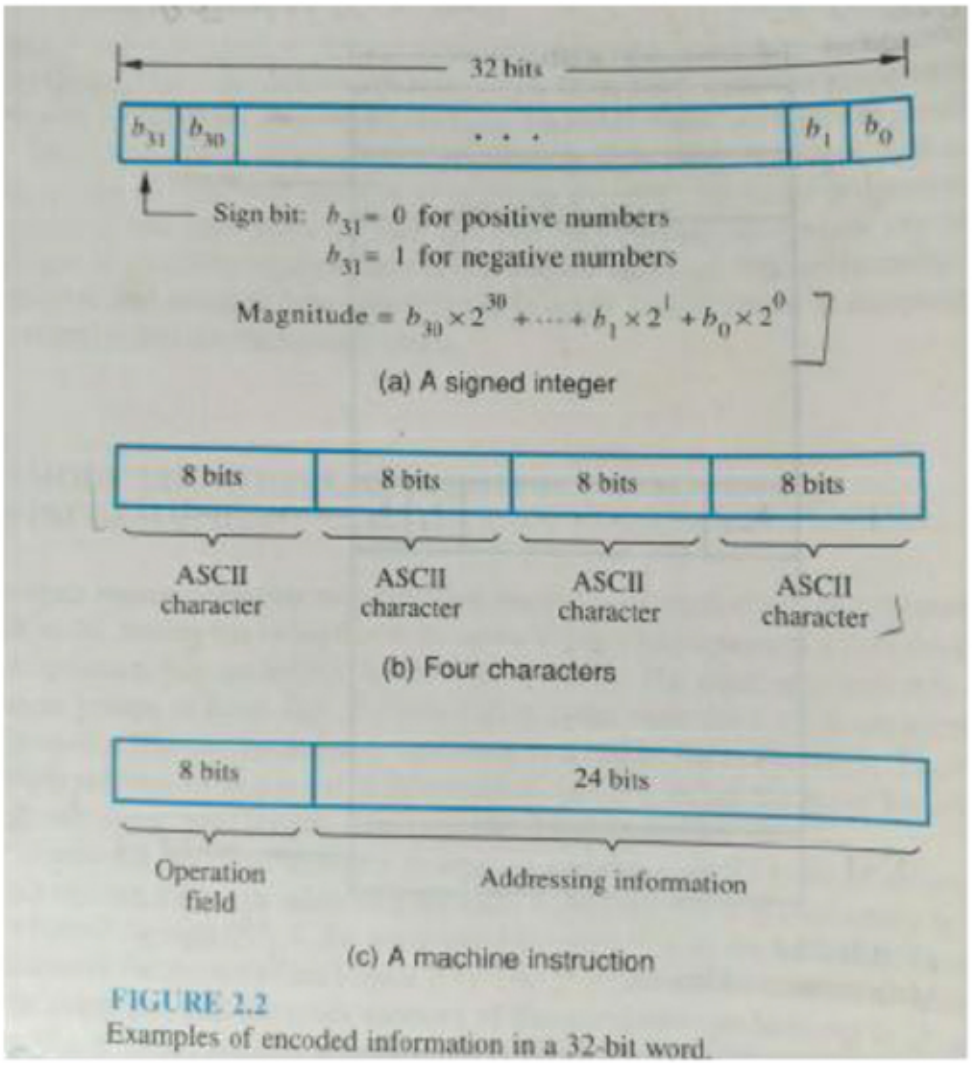


FIGURE 2.2
Examples of encoded information in a 32-bit word.

Note:

- ✓ Characters can be letters of the alphabet, decimal digits, and punctuation marks etc.
- ✓ Characters are represented by codes that are usually 8 bits long i.e. ASCII code
- ✓ The three basic information quantities are: bit, byte and word.
- ✓ A byte is always 8 bits, but the word length typically ranges from 1 to 64 bits.
- ✓ It is impractical to assign distinct addresses to individual bit locations in the memory.

BYTE ADDRESSABILITY

- ➢ In byte addressable memory, successive addresses refer to successive byte locations in the memory.
- ➢ Byte locations have addresses 0, 1, 2. . . . .
- ➢ If the word length is 32 bits, successive words are located at addresses 0, 4, 8. .with each word having 4 bytes.

BIG-ENDIAN AND LITTLE-ENDIAN ASSIGNMENTS

There are two ways in which byte addresses are arranged.

1. **Big-endian assignment**: lower byte addresses are used for the more significant bytes of the word (Figure 2.3).
2. **Little-endian**: lower byte addresses are used for the less significant bytes of the word

In both cases, byte addresses 0, 4, 8. . . . . are taken as the addresses of successive words in the memory.

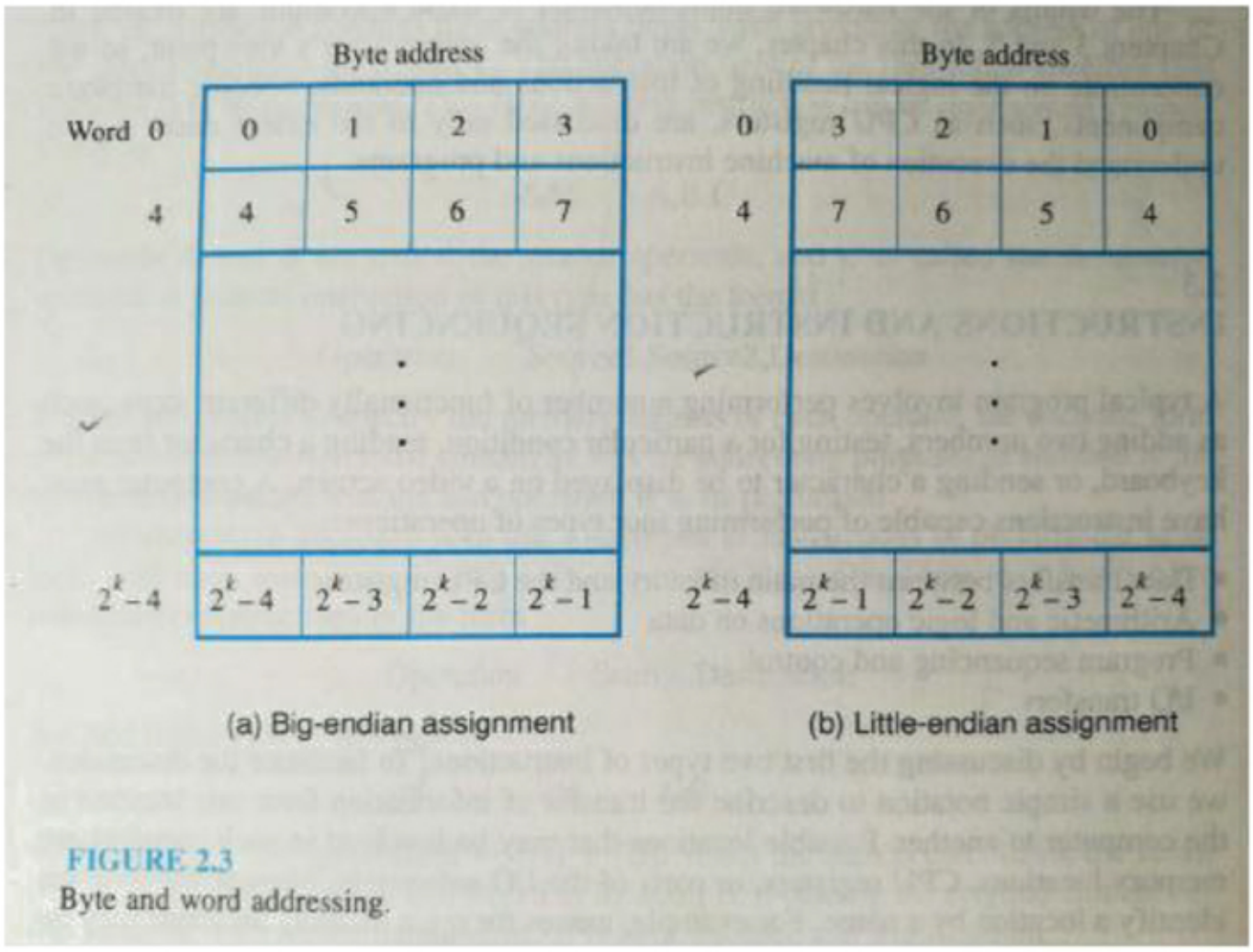


FIGURE 2.3
Byte and word addressing.

WORD ALIGNMENT

- Words are said to be aligned in memory if they begin at a byte address that is a multiple of the number of bytes in a word.
- For example, if the word length is 16(2 bytes), aligned words begin at byte addresses 0, 2, 4 . . . . . And for a word length of 64, aligned words begin at byte addresses 0, 8, 16. . . . . .
- Words are said to have unaligned addresses, if they begin at an arbitrary byte address.

ACCESSING NUMBERS, CHARACTERS & CHARACTERS STRINGS

- A number usually occupies one word. It can be accessed in the memory by specifying its word address. Similarly, individual characters can be accessed by their byte address.
- There are two ways to indicate the length of the string
  → a **special control character** with the meaning "end of string" can be used as the last character in the string, or
  → a **separate memory word location** or processor register can contain a number indicating the length of the string in bytes

MEMORY OPERATIONS

- Two basic operations involving the memory are: Load(Read/Fetch) and Store(Write).
- The Load operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged.

- The steps for Load operation:
    1. Processor sends the address of the desired location to the memory
    2. Processor issues "read" signal to memory to fetch the data
    3. Memory reads the data stored at that address
    4. Memory sends the read data to the processor

- The Store operation transfers the information from the processor register to the specified memory location. This will destroy the original contents of that memory location.
- The steps for Store operation are:
    1. Processor sends the address of the memory location where it wants to store data
    2. Processor issues "write" signal to memory to store the data
    3. Content of register (MDR) is written into the specified memory location

INSTRUCTIONS & INSTRUCTION SEQUENCING

A computer must have instructions capable of performing 4 types of operations:

1. **Data transfers** between the memory and the processor registers (MOV, PUSH, POP, XCHG),
2. **Arithmetic and logic operations** on data (ADD, SUB, MUL, DIV, AND, OR, NOT),
3. **Program sequencing and control** (CALL.RET, LOOP, INT),
4. **I/O transfers** (IN, OUT),

REGISTER TRANSFER NOTATION (RTN)

We identify a memory location by a symbolic name (in uppercase alphabets). For example, LOC, PLACE, NUM etc indicate memory locations.R0, R5 etc indicate processor register. DATAIN, OUTSTATUS etc indicate I/O registers.

Example,

$$R \leftarrow [LOC]$$

Means that the contents of memory location LOC are transferred into processor register R1 (The contents of a location are denoted by placing **[ ]** square brackets around the name of the location).

$$R3 \leftarrow [R1] + [R2]$$

Indicates the operation that adds the contents of registers R1 and R2 ,and then places their sum into register R3.

- This type of notation is known as **RTN** (Register Transfer Notation).

ASSEMBLY LANGUAGE NOTATION

To represent machine instructions and programs, assembly language format can be used.

Example,

Move LOC, R1;

This instruction transfers data from memory-location LOC to processor-register R1. The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1 are overwritten.

Add R1, R2, R3;

This instruction adds 2 numbers contained in processor-registers R1 and R2, and places their sum in R3.

Note:

- ✓ A computer performs its task according to the program stored in memory. A program is a collection of instructions which tell the processor to perform a basic operation like addition, reading from keyboard etc.
- ✓ Possible locations that may be involved in data transfers are memory locations, processor registers or registers in the I/O subsystem.

BASIC INSTRUCTION TYPES

C = A + B;

- ➢ This statement is a command to the computer to add the current values of the two variables A and B, and to assign the sum to a third variable C.
- ➢ When the program is compiled, each variable is assigned a distinct address in memory.
- ➢ The contents of these locations represent the values of the three variables
- ➢ The statement ***C←[A]+[B]*** indicates that the contents of memory locations A and B are fetched from memory, transferred to the processor, sum is computed and then result is stored in memory location C.

1. Three-Address Instruction

The instruction has general format

Operation Source1, Source2, Destination

Example

Add A, B, C;

Performs the operation

C ← [A] + [B]

Operands A and B are called the source operands, C is called the destination operand, and Add is the operation to be performed.

2. Two-Address Instruction

The instruction has general format

Operation Source, Destination

Example,

Add A, B;

Performs the operation

B ← [A] + [B]

When the sum is calculated, the result is sent to the memory and stored in location B, replacing the original contents of this location. This means that operand B is both a source and a destination.

The operation ***C ← [A] + [B]*** can be performed by the two-instruction sequence

Move B, C
Add A, C

One-Address Instruction

The instruction has general format

Operation Source/Destination

Example,

Add A;

Performs

AC ← [A] + [AC]

Adds the contents of memory location A to the contents of the accumulator (AC) register and place the sum back into the accumulator.

Load A;

This instruction copies the contents of memory location A into the accumulator and

Store A;

This instruction copies the contents of the accumulator into memory location A.

The operation ***C ← [A] + [B]*** can be performed by executing the sequence of instructions

Load A
Add B
Store C

The operand may be a source or a destination depending on the instruction. In the Load instruction, address A specifies the source operand, and the destination location, the accumulator, is implied. On the other hand, C denotes the destination location in the Store instruction, whereas the source, the accumulator, is implied.

Zero-Address Instruction

The instruction has general format

Operation

The locations of all operands are defined implicitly. The operands are stored in a structure called pushdown stack. In this case, the instructions are called zero-address instructions.

Example

ADD

Performs

TOC ← [TOC] + [TOC-1]

Where both operands for addition are present in Top of Stack (TOC), Top two elements are popped and final sum is pushed back to Stack.

Note:

→ Access to data in the registers is much faster than to data stored in memory locations because the registers are inside the processor.

Let Ri represent a general-purpose register. The instructions are generalizations of the Load, Store and Add Instructions for the single-accumulator case, in which register Ri performs the function of the accumulator.

Load A,Ri

Store Ri,A
Add A,Ri

In processors where arithmetic operations as allowed only on operands that are in processor registers, the C=A+B task can be performed by the instruction sequence

Move A,Ri
Move B,Rj
Add Ri,Rj
Move Rj,C

INSTRUCTION EXECUTION & STRAIGHT LINE SEQUENCING

The program is executed as follows:

1. Initially, the address of the first instruction is loaded into PC (Program counter is a register which holds the address of the next instruction to be executed)
2. Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called *straight-line sequencing* (Figure 2.4)
3. During the execution of each instruction, the PC is incremented by 4 to point to the next instruction.
4. Executing given instruction is a two-phase procedure.
    i. In fetch phase, the instruction is fetched from the memory location (whose address is in the PC) and placed in the IR of the processor
    ii. In execute phase, the contents of IR is examined to determine which operation is to be performed. The specified operation is then performed by the processor.

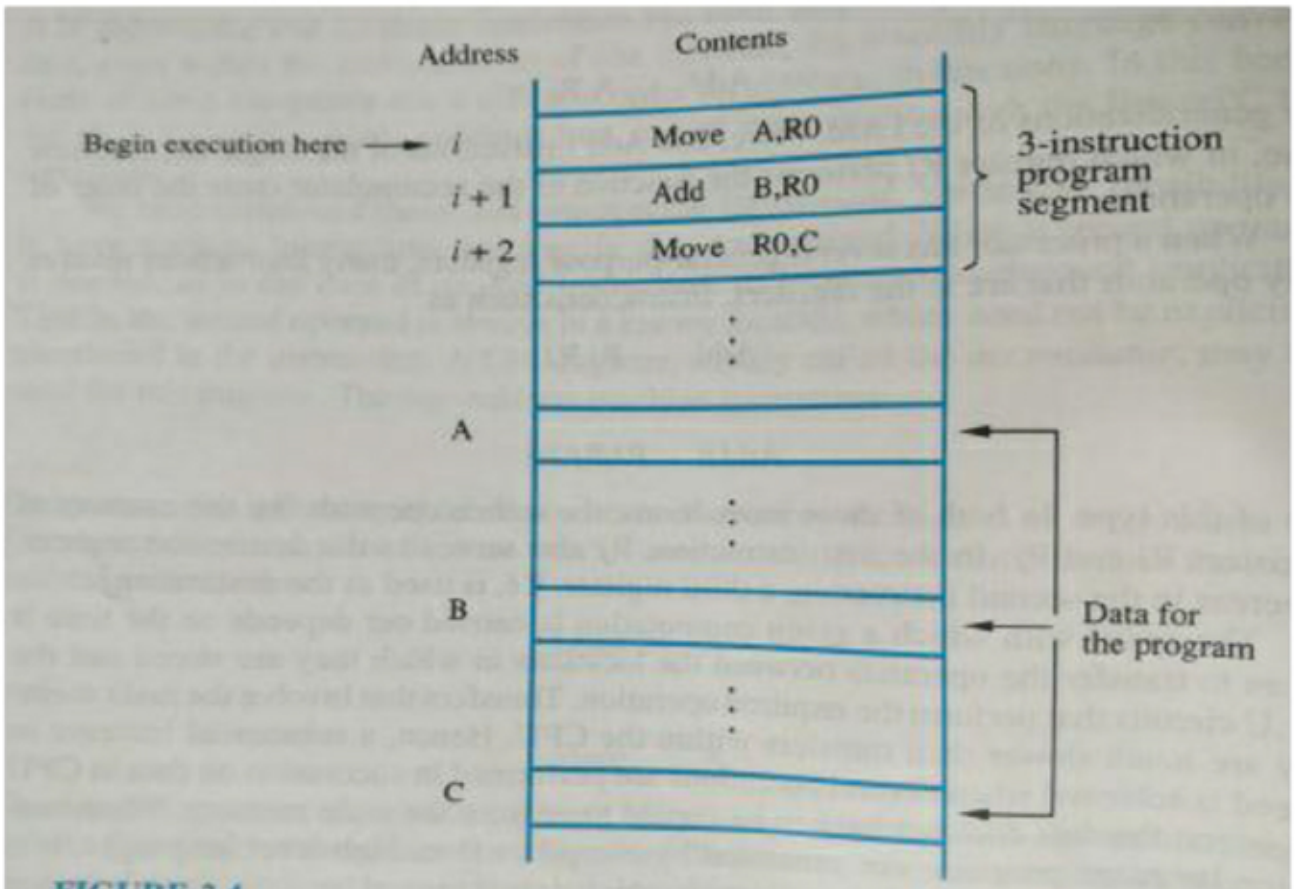


FIGURE 2.4
A program for C ← [A] + [B].

BRANCHING

- Consider the task of adding a list of n numbers (Figure 2.10).
- The loop is a straight line sequence of instructions executed as many times as needed. It starts at location LOOP and ends at the instruction Branch > 0.
- During each pass through this loop, the address of the next list entry is determined, and that entry is fetched and added to R0.
- Register R1 is used as a counter to determine the number of times the loop is executed. Hence, the contents of location N are loaded into register R1 at the beginning of the program.
- Within the body of the loop, the instruction *Decrement R1* reduces the contents of R1 by 1 each time through the loop.
- Then Branch instruction loads a new value into the program counter. As a result, the processor fetches and executes the instruction at this new address called the branch target.
- A conditional branch instruction causes a branch only if a specified condition is satisfied. If the condition is not satisfied, the PC is incremented in the normal way, and the next instruction in sequential address order is fetched and executed.

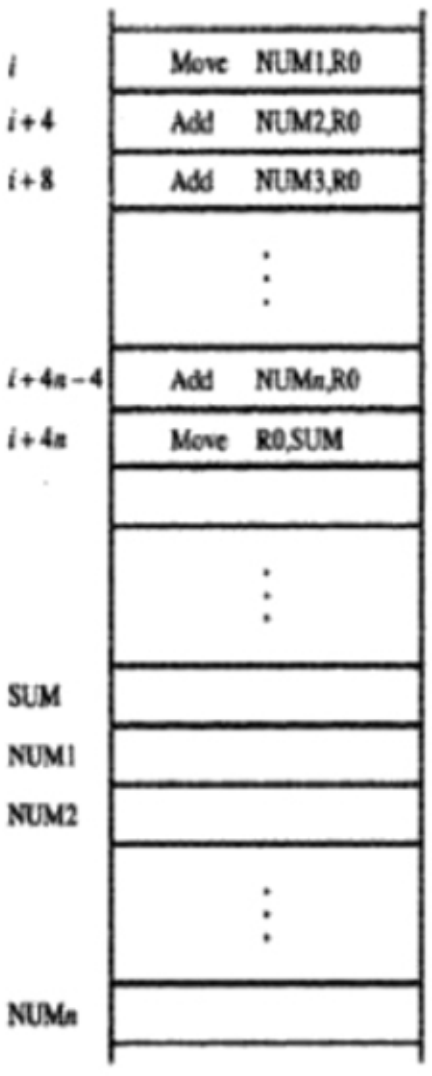


Figure 2.9 A straight-line program for adding *n* numbers.

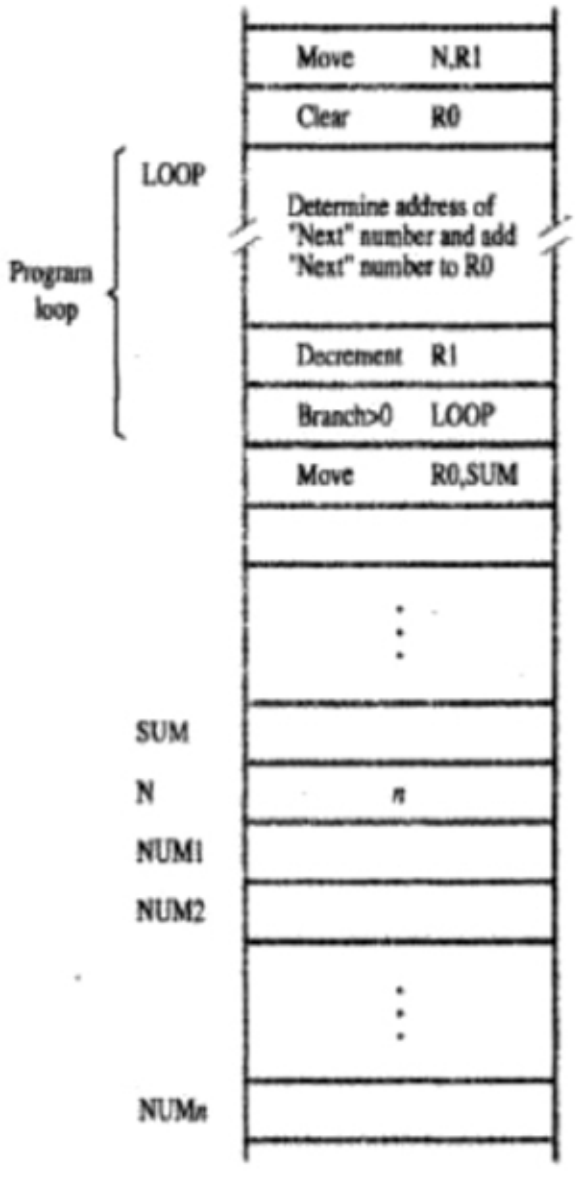


Figure 2.10 Using a loop to add *n* numbers.

CONDITION CODES

- The processor keeps track of information about the results of various operations. This is accomplished by recording the required information in individual bits, called ***condition code flags***.
- These flags are grouped together in a special processor-register called the *condition code register* (or statue register).
- Four commonly used flags are
  - → **N (negative)** set to 1 if the result is negative, otherwise cleared to 0
  - → **Z (zero)** set to 1 if the result is 0; otherwise, cleared to 0
  - → **V (overflow)** set to 1 if arithmetic overflow occurs; otherwise, cleared to 0
  - → **C (carry)** set to 1 if a carry-out results from the operation; otherwise cleared to 0

ADDRESSING MODES

The different ways in which the location of an operand is specified in an instruction are referred to as ***addressing modes.*** The different generic addressing modes are listed in Table 2.1.

Table 2.1 Generic addressing modes

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC
Indirect	(R*i*)	EA = [R*i*]
	(LOC)	EA = [LOC]
Index	X(R*i*)	EA = [R*i*] + X
Base with index	(R*i*,R*j*)	EA = [R*i*] + [R*j*]
Base with index and offset	X(R*i*,R*j*)	EA = [R*i*] + [R*j*] + X
Relative	X(PC)	EA = [PC] + X
Autoincrement	(R*i*)+	EA = [R*i*]; Increment R*i*
Autodecrement	−(R*i*)	Decrement R*i*; EA = [R*i*]

EA = effective address
Value = a signed number

Implementation of Variable and Constants

- Variables & constants are the simplest data-types and are found in almost every computer program.
- In assembly language, a variable is represented by allocating a register (or memory-location) to hold its value. Thus, the value can be changed as needed using appropriate instructions.

Register Mode

- The operand is the contents of a register.
- The name (or address) of the register is given in the instruction.
- Registers are used as temporary storage locations where the data in a register are accessed.

For example, the instruction,

Move R1, R2 Copy content of register R1 into register R2

Absolute (Direct) Mode

- The operand is in a memory-location.
- The address of memory-location is given explicitly in the instruction.

For example, the instruction,

Move LOC, R2 Copy content of memory-location LOC into register R2

Immediate Mode

- The operand is given explicitly in the instruction.
- The immediate value is prefixed with #.
- The immediate mode is only used to specify the value of a source-operand.

For example, the instruction,

Move #200, R0 Place the value 200 in register R0

INDIRECTION AND POINTERS

In this case, the instruction does not give the operand or its address explicitly; instead, it provides information from which the memory-address of the operand can be determined. We refer to this address as the ***effective address (EA)*** of the operand.

Indirect Mode

- The EA of the operand is the contents of a register (or memory-location) whose address appears in the instruction.
- The register (or memory-location) that contains the address of an operand is called a *pointer*. {The indirection is denoted by ( ) sign around the register or memory-location}.

For example, the instruction

Add (R1), R0

The operand is in memory. Register R1 gives the effective-address (B) of the operand. The data is read from location B and added to contents of register R0

- To execute the Add instruction in fig (a), the processor uses the value which is in register R1, as the EA 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 fig (b). In this case, the processor first reads the contents of memory location A,
- then requests a second read operation using the value B as an address to obtain the operand

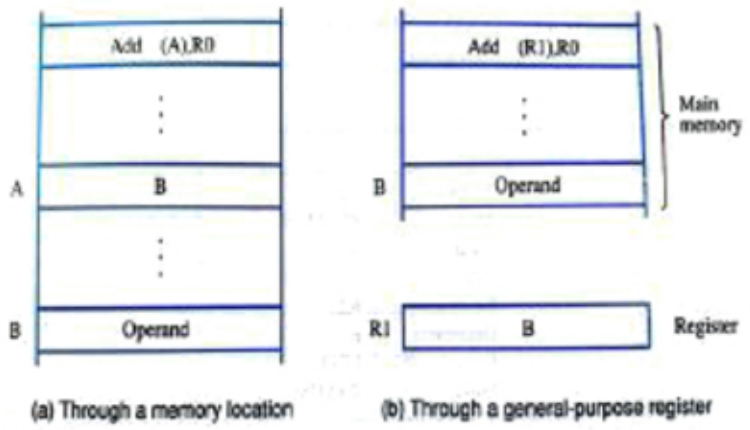


FIGURE 2.7
Indirect addressing.

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

FIGURE 2.8
Indirect addressing used in the program of Figure 2.6.

- In above program, 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 first two instructions in the loop implement the unspecified instruction block starting at LOOP.
- The first time through the loop, 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.

INDEXING AND ARRAYS

A different kind of flexibility for accessing operands is useful in dealing with lists and arrays.

Index mode

The operation is indicated as X(Ri)

Where

X = the constant value contained in the instruction.

Ri = the name of the index register

The effective-address of the operand is given by $EA = X + [Ri]$

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.

- The Figure 2.13 illustrates two ways of using the Index mode. In Figure 2.13 (a), 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 2.13 (b). 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.

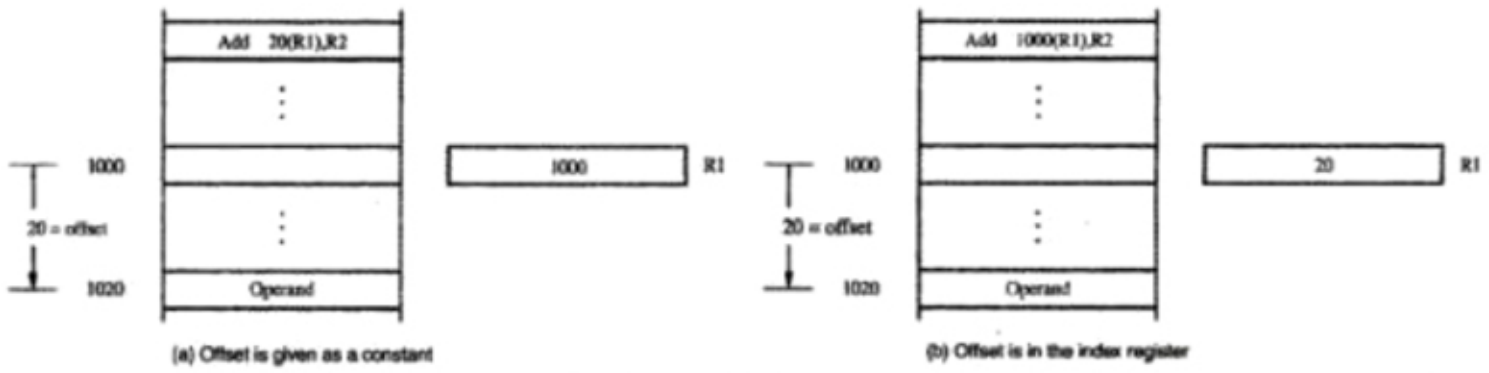


Figure 2.13 Indexed addressing.

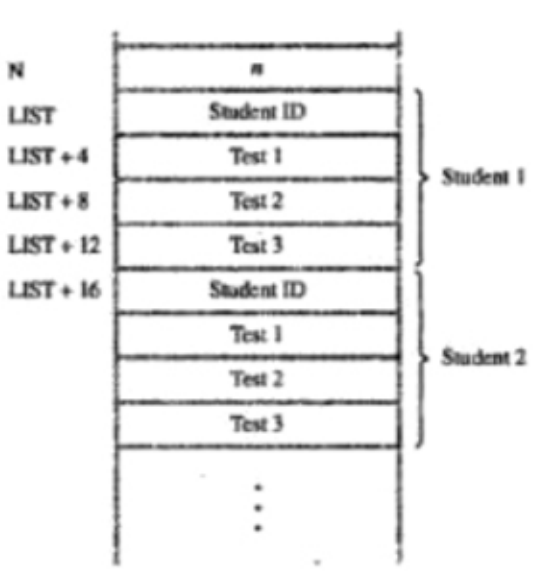


Figure 2.14 A list of students' marks.

```
        Move    #LIST,R0
        Clear   R1
        Clear   R2
        Clear   R3
        Move    N,R4
LOOP    Add     4(R0),R1
        Add     8(R0),R2
        Add     12(R0),R3
        Add     #16,R0
        Decrement R4
        Branch>0 LOOP
        Move    R1,SUM1
        Move    R2,SUM2
        Move    R3,SUM3
```

Figure 2.15 Indexed addressing used in accessing test scores in the list in Figure 2.14.

Base with Index Mode

- Another version of the Index mode uses 2 registers which can be denoted as (Ri, Rj)
- Here, a second register may be used to contain the offset X.
- The second register is usually called the *base register*.
- The effective-address of the operand is given by $EA = [Ri] + [Rj]$
- This form of indexed addressing provides more flexibility in accessing operands, because both components of the effective address can be changed.

Base with Index & Offset Mode

- Another version of the Index mode uses 2 registers plus a constant, which can be denoted as *X(Ri, Rj)*
- The effective-address of the operand is given by $EA = X + [Ri] + [Rj]$

- This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the (Ri, Rj) part of the addressing-mode. In other words, this mode implements a 3-dimensional array.

RELATIVE MODE

- This is similar to index-mode with an exception: The effective address is determined using the PC in place of the general purpose register Ri.
- The operation is indicated as X (PC).
- X(PC) denotes an effective-address of the operand which is X locations above or below the current contents of PC.
- Since the addressed-location is identified "relative" to the PC, the name Relative mode is associated with this type of addressing.
- This mode is used commonly in conditional branch instructions.

An instruction such as

Branch > 0 LOOP

Causes program execution to go to the branch target location identified by name LOOP if branch condition is satisfied.

ADDITIONAL ADDRESSING MODES

The following 2 modes are useful for accessing data items in successive locations in the memory.

Auto-increment Mode

- The effective-address of operand is the contents of a register specified in the instruction (Fig: 2.16).
- After accessing the operand, the contents of this register are automatically incremented to point to the next item in a list.
- Implicitly, the increment amount is 1.

This mode is denoted as

(Ri)+ where Ri = pointer register

Auto-decrement Mode

The contents of a register specified in the instruction are first automatically decremented and are then used as the effective address of the operand.

This mode is denoted as

-(Ri) where Ri = pointer register

These 2 modes can be used together to implement an important data structure called a stack.

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

FIGURE 2.8
Indirect addressing used in the program of Figure 2.6.

Register Transfer Notation (RTN)

- Identify a location by a symbolic name standing for its hardware binary address (LOC, R0,…)
- Contents of a location are denoted by placing square brackets around the name of the location

$$R1 \leftarrow [LOC]$$
$$R3 \leftarrow [R1] + [R2]$$

ASSEMBLY LANGUAGE

- A complete set of symbolic names (mnemonic code) and rules for their use constitute an assembly language.
- The set of rules for using the mnemonics in the specification of complete instructions and programs is called the *syntax* of the language.
- Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an *assembler*.
- The user program in its original alphanumeric text formal is called a *source program*, and the assembled machine language program is called an *object program*.

Move instruction is written is

MOVE R0, SUM

The mnemonic MOVE represents the binary pattern, or OP code, for the operation performed by the instruction.
The instruction

ADD #5, R3

Adds the number 5 to the contents of register R3 and puts the result back into register R3.

ASSEMBLER DIRECTIVES

Assembler directives are instructions that direct the assembler to do something.

- EQU informs the assembler about the value of an identifier (Figure: 2.18).

Ex*ample*

SUM EQU 200

This statement informs the assembler that the name SUM should be replaced by the value 200 wherever it appears in the program.

- ORIGIN tells the assembler about the starting-address of memory-area to place the data block.
- DATAWORD directive tells the assembler to load a value (say 100) into the location (say 204).

Example

N DATAWORD 100

- RESERVE directive declares that a memory-block of 400 bytes is to be reserved for data and that the name NUM1 is to be associated with address 208.

Example

NUM1 RESERVE 400

- END directive tells the assembler that this is the end of the source-program text.
- RETURN directive identifies the point at which execution of the program should be terminated.

Any statement that makes instructions or data being placed in a memory-location may be given a label. The label (say N or NUM1) is assigned a value equal to the address of that location.

GENERAL FORMAT OF A STATEMENT

Most assembly languages require statements in a source program to be written in the form:

Label Operation Operands Comment

→*Label* is an optional name associated with the memory-address where the machine language instruction produced from the statement will be loaded.

→The *Operation* field contains the OP-code mnemonic of the desired instruction or assembler

→The *Operand*s field contains addressing information for accessing one or more operands, depending on the type of instruction.

→The *Comment* field is used for documentation purposes to make the program easier to understand.

ASSEMBLY AND EXECUTION OF PRGRAMS

Programs written in an assembly language are automatically translated into a sequence of machine instructions by the assembler.

• Assembler program

✓Replaces all symbols denoting operations & addressing-modes with binary-codes used in machine instructions.

	Memory address label	Operation	Addressing or data information
Assembler directives	SUM	EQU	200
		ORIGIN	204
	N	DATAWORD	100
	NUM1	RESERVE	400
		ORIGIN	100
Statements that generate machine instructions	START	MOVE	N,R1
		MOVE	#NUM1,R2
		CLR	R0
	LOOP	ADD	(R2),R0
		ADD	#4,R2
		DEC	R1
		BGTZ	LOOP
		MOVE	R0,SUM
Assembler directives		RETURN	
		END	START

Figure 2.18 Assembly language representation for the program in Figure 2.17.

- ✓Replaces all names and labels with their actual values.
- ✓Assigns addresses to instructions & data blocks, starting at the address given in the ORIGIN directive.
- ✓Inserts constants that may be given in DATAWORD directives.
- ✓Reserves memory-space as requested by RESERVE directives.

- ➢ As the assembler scans through a source-program, it keeps track of all names of numerical-values that correspond to them in a symbol-table. Thus, when a name appears a second time, it is replaced with its value from the table. Hence, such an assembler is called a *two-pass assembler.*
- ➢ The assembler stores the object-program on a magnetic-disk. The object-program must be loaded into the memory of the computer before it is executed. For this, a *loader program* is used.
- ➢ *Debugger program* is used to help the user find the programming errors.

➢Debugger program enables the user

→ to stop execution of the object-program at some points of interest and

→ to examine the contents of various processor registers and memory-location

BASIC INPUT/OUTPUT OPERATIONS

The Input and Output operation can be done in two ways

1. Program controlled.
2. Memory Mapped.

Program Controlled IO Operation.

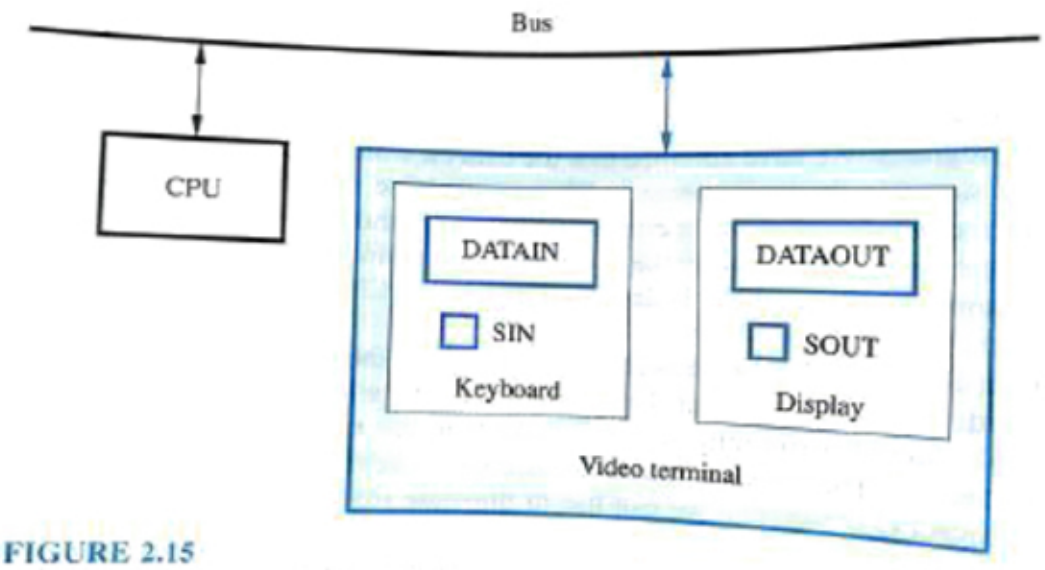


FIGURE 2.15
A video terminal connected to a CPU.

- ➢ Consider the problem of moving a character-code from the keyboard to the processor. For this transfer, buffer-register (DATAIN) & a status control flags (SIN) are used.
- ➢ Striking key stores the corresponding character-code in an 8-bit buffer-register (DATAIN) associated with the keyboard (Figure: 2.19).
- ➢ To inform the processor that a valid character is in DATAIN, a SIN is set to 1.
- ➢ A program monitors SIN, and when SIN is set to 1, the processor reads the contents of DATAIN.
- ➢ When the character is transferred to the processor, SIN is automatically cleared to 0.

- If a second character is entered at the keyboard, SIN is again set to 1 and the process repeats.
- An analogous process takes place when characters are transferred from the processor to the display. A buffer-register, DATAOUT, and a status control flag, SOUT are used for this transfer.
- When SOUT=1, the display is ready to receive a character.
- The transfer of a character to DATAOUT clears SOUT to 0.
- The buffer registers DATAIN and DATAOUT and the status flags SIN and SOUT are part of circuitry commonly known as a *device interface*.

Program in Figure 2.20 reads a line of characters and display (echos) it

```
        Move        #LOC,R0          Initialize pointer register R0 to point to the
                                       address of the first location in memory
                                       where the characters are to be stored.
READ    TestBit     #3,INSTATUS      Wait for a character to be entered
        Branch=0    READ              in the keyboard buffer DATAIN.
        MoveByte    DATAIN,(R0)      Transfer the character from DATAIN into
                                       the memory (this clears SIN to 0).
ECHO    TestBit     #3,OUTSTATUS     Wait for the display to become ready.
        Branch=0    ECHO
        MoveByte    (R0),DATAOUT     Move the character just read to the display
                                       buffer register (this clears SOUT to 0).
        Compare     #CR,(R0)+        Check if the character just read is CR
                                       (carriage return). If it is not CR, then
        Branch≠0    READ              branch back and read another character.
                                       Also, increment the pointer to store the
                                       next character.
```

Figure 2.20 A program that reads a line of characters and displays it.

MEMORY-MAPPED I/O

- Some address values are used to refer to peripheral device buffer-registers such as DATAIN and DATAOUT.
- No special instructions are needed to access the contents of the registers; data can be transferred between these registers and the processor using instructions such as Move, Load or Store.
- For example, contents of the keyboard character buffer DATAIN can be transferred to register R1 in the processor by the instruction

  MoveByte DATAIN, R1
- The MoveByte operation code signifies that the operand size is a byte.
- The Testbit instruction tests the state of one bit in the destination, where the bit position to be tested is indicated by the first operand.

STACKS

- A stack is a list of data elements with the accessing restriction that elements can be added or removed at one end of the list only. This end is called the top of the stack, and the other end is called the bottom (Figure: 2.21).
- 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.
- 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 SP (Stack Pointer).
- If we assume a byte-addressable memory with a 32-bit word length,

The push operation can be implemented as

Subtract #4, SR

Move NEWITEM, (SP)

The pop operation can be implemented as

Move (SP), ITEM

Add #4,SP

Routine for a safe pop and push operation as follows

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 2.23 Checking for empty and full errors in pop and push operations.

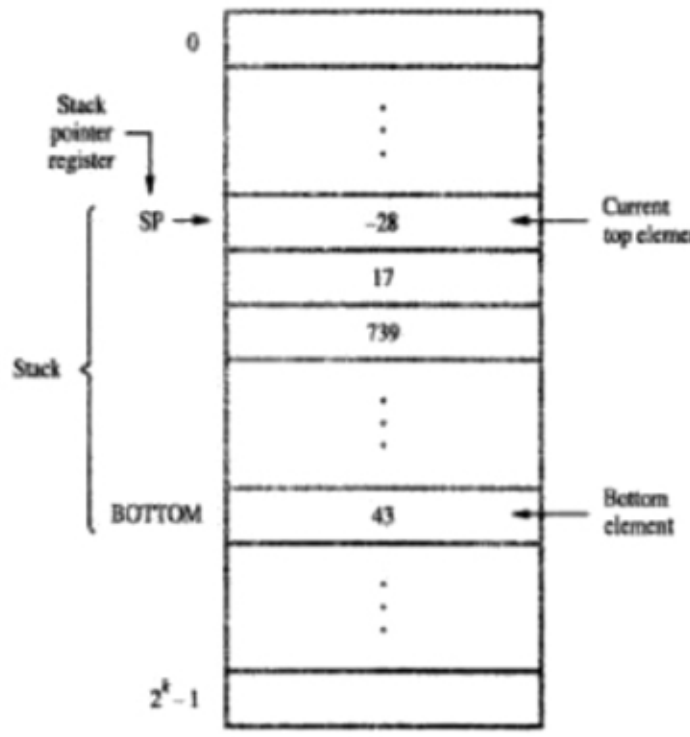


Figure 2.21 A stack of words in the memory.

QUEUE

- Data are stored in and retrieved from a queue on a FIFO basis.

Difference between stack and queue.

1. One end of the stack is fixed while the other end rises and falls as data are pushed and popped.
2. A single pointer is needed to point to the top of the stack at any given time. On the other hand, both does of a queue move to higher addresses as data are added at the back and removed from the front. So, two pointers are needed to keep track of the two does of the queue.

3. Without further control, a queue would continuously move through the memory of a computer in the direction of higher addresses. One way to limit the queue to a fixed region in memory is to use a circular buffer.

SUBROUTINES

- A subtask consisting of a set of instructions which is executed many times is called a *subroutine.*
- The program branches to a subroutine with a Call instruction (Figure: 2.24).
- Once the subroutine is executed, the calling-program must resume execution starting from the instruction immediately following the Call instructions i.e. control is to be transferred back to the calling-program. This is done by executing a Return instruction at the end of the subroutine.
- 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 a special branch instruction that performs the following operations:
  - → Store the contents of PC into 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.

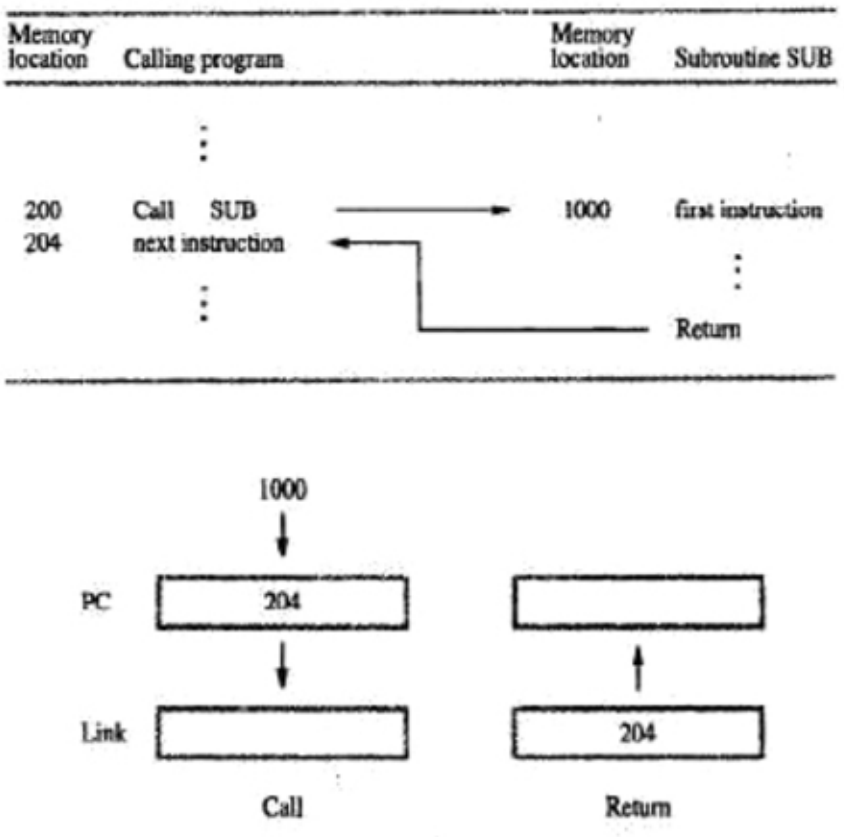


Figure 2.24 Subroutine linkage using a link register.

SUBROUTINE NESTING AND THE PROCESSOR STACK

- *Subroutine nesting* means one subroutine calls another subroutine.
- In this case, the return-address of the second call is also stored in the link-register, destroying its previous contents.
- Hence, it is essential to save the contents of the link-register in some other location before calling another subroutine. Otherwise, the return-address of the first subroutine will be lost.
- Subroutine nesting can be carried out to any depth. Eventually, the last subroutine called completes its computations and returns to the subroutine that called it.
- The return-address needed for this first return is the last one generated in the nested call sequence. That is, return-addresses are generated and used in a LIFO order.
- This suggests that the return-addresses associated with subroutine calls should be pushed onto a stack. A particular register is designated as the SP(Stack Pointer) to be used in this operation.
- SP is used to point to the processor-stack.
- Call instruction pushes the contents of the PC onto the processor-stack.
- Return instruction pops the return-address from the processor-stack into the PC.

PARAMETER PASSING

- The exchange of information between a calling-program and a subroutine is referred to as *parameter passing* (Figure: 2.25).
- The parameters may be placed in registers or in memory-location, where they can be accessed by the subroutine.
- Alternatively, parameters may be placed on the processor-stack used for saving the return-address
- Following is a program for adding a list of numbers using subroutine with the parameters passed through registers.

Calling program

```
        Move      N,R1          R1 serves as a counter.
        Move      #NUM1,R2      R2 points to the list.
        Call      LISTADD       Call subroutine.
        Move      R0,SUM        Save result.
        ⋮
```

Subroutine

```
LISTADD Clear     R0            Initialize sum to 0.
LOOP    Add       (R2)+,R0      Add entry from list.
        Decrement R1
        Branch>0  LOOP
        Return                  Return to calling program.
```

Figure 2.25 Program of Figure 2.16 written as a subroutine; parameters passed through registers.

STACK FRAME

- *The Locations constitute a private work space for the subroutine, created at the time the subroutine is entered and freed up when the subroutine returns the control to calling program such space is called Stack Frame* (Figure:2.26).
- The work-space is
  - → created at the time the subroutine is entered &
  - → Freed up when the subroutine returns control to the calling-program.
- Following is a program for adding a list of numbers using subroutine with the parameters passed to stack
- Fig: 2.27 show an example of a commonly used layout for information in a stack-frame.
- *Frame pointer(FP)* is used to access the parameters passed
  - → to the subroutine &
  - → to the local memory-variables
- The contents of FP remains fixed throughout the execution of the subroutine, unlike stack-pointer SP, which must always point to the current top element in the stack.

Operation on Stack Frame

- Initially SP is pointing to the address of old TOS.
- The calling-program saves 4 parameters on the stack (Figure 2.27).
- The Call instruction is now executed, pushing the return-address onto the stack.
- Now, SP points to this return-address, and the first instruction of the subroutine is executed.
- Now, FP is to be initialized and its old contents have to be stored. Hence, the first 2 instructions in the subroutine are:

Move FP,-(SP)
Move SP, FP

- The FP is initialized to the value of SP i.e. both FP and SP point to the saved FP address.
- The 3 local variables may now be pushed onto the stack. Space for local variables is allocated by executing the instruction

Subtract #12, SP

- Finally, the contents of processor-registers R0 and R1 are saved in the stack. At this point, the stack-frame has been set up as shown in the fig 2.27.
- The subroutine now executes its task. When the task is completed, the subroutine pops the saved values of R1 and R0 back into those registers, removes the local variables from the stack frame by executing the instruction.

Add #12, SP

- And subroutine pops saved old value of FP back into FP. At this point, SP points to return-address, so the Return instruction can be executed, transferring control back to the calling-program.

Memory location		Instructions		Comments
Main program				
		⋮		
2000		Move	PARAM2,−(SP)	Place parameters on stack.
2004		Move	PARAM1,−(SP)	
2008		Call	SUB1	
2012		Move	(SP),RESULT	Store result.
2016		Add	#8,SP	Restore stack level.
2020		next instruction		
		⋮		
First subroutine				
2100	SUB1	Move	FP,−(SP)	Save frame pointer register.
2104		Move	SP,FP	Load the frame pointer.
2108		MoveMultiple	R0−R3,−(SP)	Save registers.
2112		Move	8(FP),R0	Get first parameter.
		Move	12(FP),R1	Get second parameter.
		⋮		
		Move	PARAM3,−(SP)	Place a parameter on stack.
2160		Call	SUB2	
2164		Move	(SP)+,R2	Pop SUB2 result into R2.
		⋮		
		Move	R3,8(FP)	Place answer on stack.
		MoveMultiple	(SP)+,R0−R3	Restore registers.
		Move	(SP)+,FP	Restore frame pointer register.
		Return		Return to Main program.

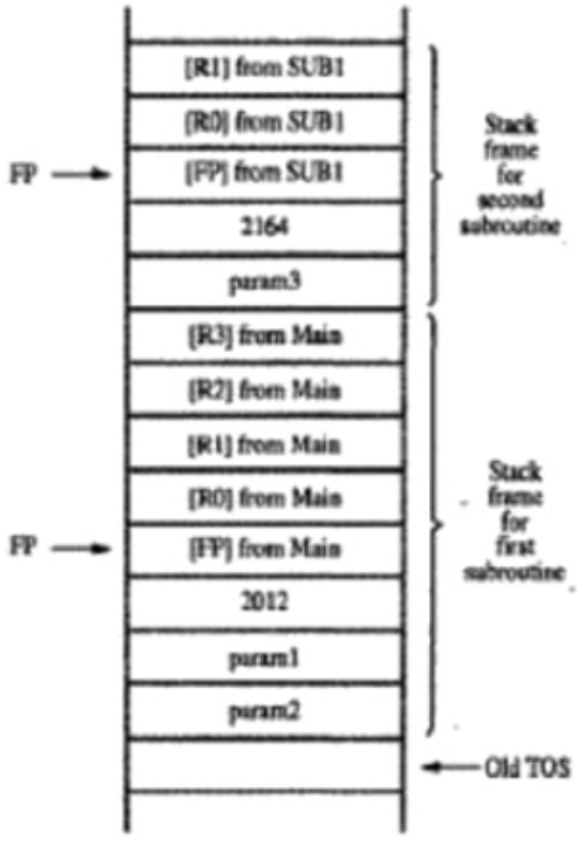


Figure 2.29 Stack frames for Figure 2.28.

Second subroutine

3000	SUB2	Move	FP,−(SP)	Save frame pointer register.
		Move	SP,FP	Load the frame pointer.
		MoveMultiple	R0−R1,−(SP)	Save registers R0 and R1.
		Move	8(FP),R0	Get the parameter.
		⋮		
		Move	R1,8(FP)	Place SUB2 result on stack.
		MoveMultiple	(SP)+,R0−R1	Restore registers R0 and R1.
		Move	(SP)+,FP	Restore frame pointer register.
		Return		Return to Subroutine 1.

Figure 2.28 Nested subroutines.

STACK FRAMES FOR NESTED SUBROUTINES

- Stack is very useful data structure for holding return-addresses when subroutines are nested.
- When nested subroutines are used; the stack-frames are built up in the processor-stack.
- Consider the above program to illustrate stack frames for nested subroutines

The Flow of Execution is as follows:

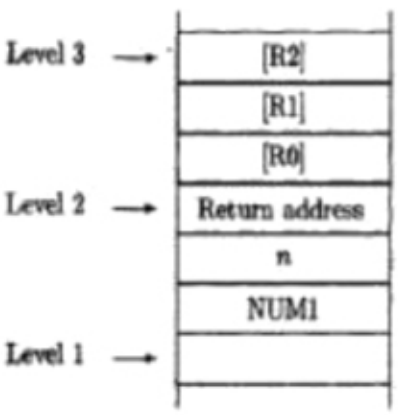
- Main program pushes the 2 parameters param2 and param1 onto the stack and then calls SUB1.
- SUB1 has to perform an operation & send result to the main-program on the stack (Fig:2.28& 29).

- During the process, SUB1 calls the second subroutine SUB2 (in order to perform some subtask).
- After SUB2 executes its Return instruction; the result is stored in register R2 by SUB1.
- SUB1 then continues its computations & eventually passes required answer back to main-program on the stack.
- When SUB1 executes return statement, the main-program stores this answers in memory-location RESULT and continues its execution.

Assume top of stack is at level 1 below.

```
                Move          #NUM1,-(SP)     Push parameters onto stack.
                Move          N,-(SP)
                Call          LISTADD         Call subroutine
                                                (top of stack at level 2).
                Move          4(SP),SUM       Save result.
                Add           #8,SP           Restore top of stack
                                                (top of stack at level 1).
                .
                .
                .
LISTADD         MoveMultiple  R0-R2,-(SP)     Save registers
                                                (top of stack at level 3).
                Move          16(SP),R1       Initialize counter to n.
                Move          20(SP),R2       Initialize pointer to the list.
                Clear         R0              Initialize sum to 0.
LOOP            Add           (R2)+,R0        Add entry from list.
                Decrement     R1
                Branch>0      LOOP
                Move          R0,20(SP)       Put result on the stack.
                MoveMultiple  (SP)+,R0-R2     Restore registers.
                Return                        Return to calling program.
```

(a) Calling program and subroutine



(b) Top of stack at various times

Figure 2.26 Program of Figure 2.16 written as a subroutine; parameters passed on the stack.

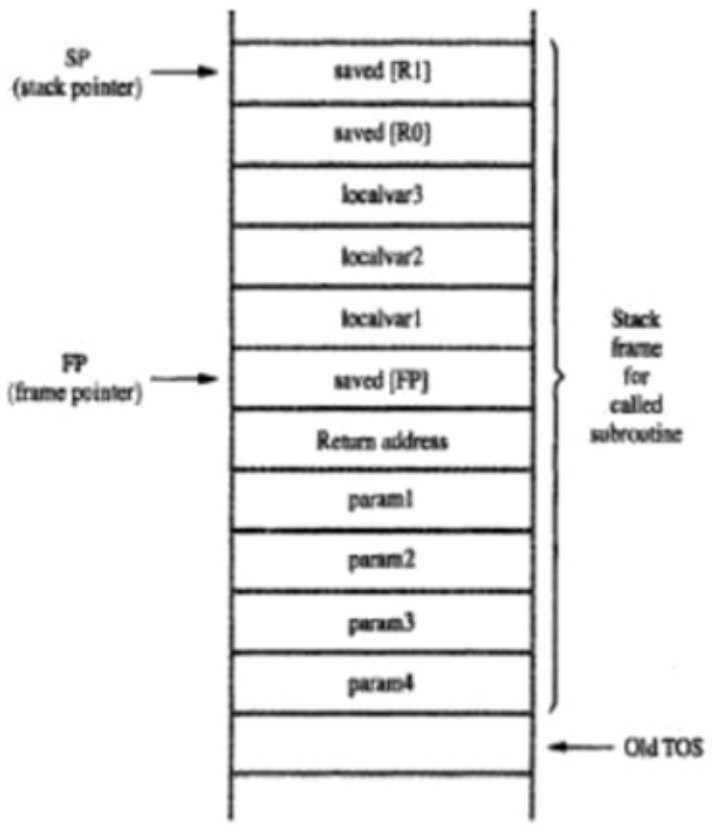


Figure 2.27 A subroutine stack frame example.

LOGIC INSTRUCTIONS

- Logic operations such as AND, OR, and NOT applied to individual bits.
- These are the basic building blocks of digital-circuits.
- This is also useful to be able to perform logic operations is software, which is done using instructions that apply these operations to all bits of a word or byte independently and in parallel.

For example, the instruction

Not dst

SHIFT AND ROTATE INSTRUCTIONS

- There are many applications that require the bits of an operand to be shifted right or left some specified number of bit positions.
- The details of how the shifts are performed depend on whether the operand is a signed number or some more general binary-coded information.
- For general operands, we use a logical shift.
  For a number, we use an arithmetic shift, which preserves the sign of the number.

LOGICAL SHIFTS

- Two logical shift instructions are needed, one for shifting left(LShiftL) and another for shifting right(LShiftR).
- These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction.

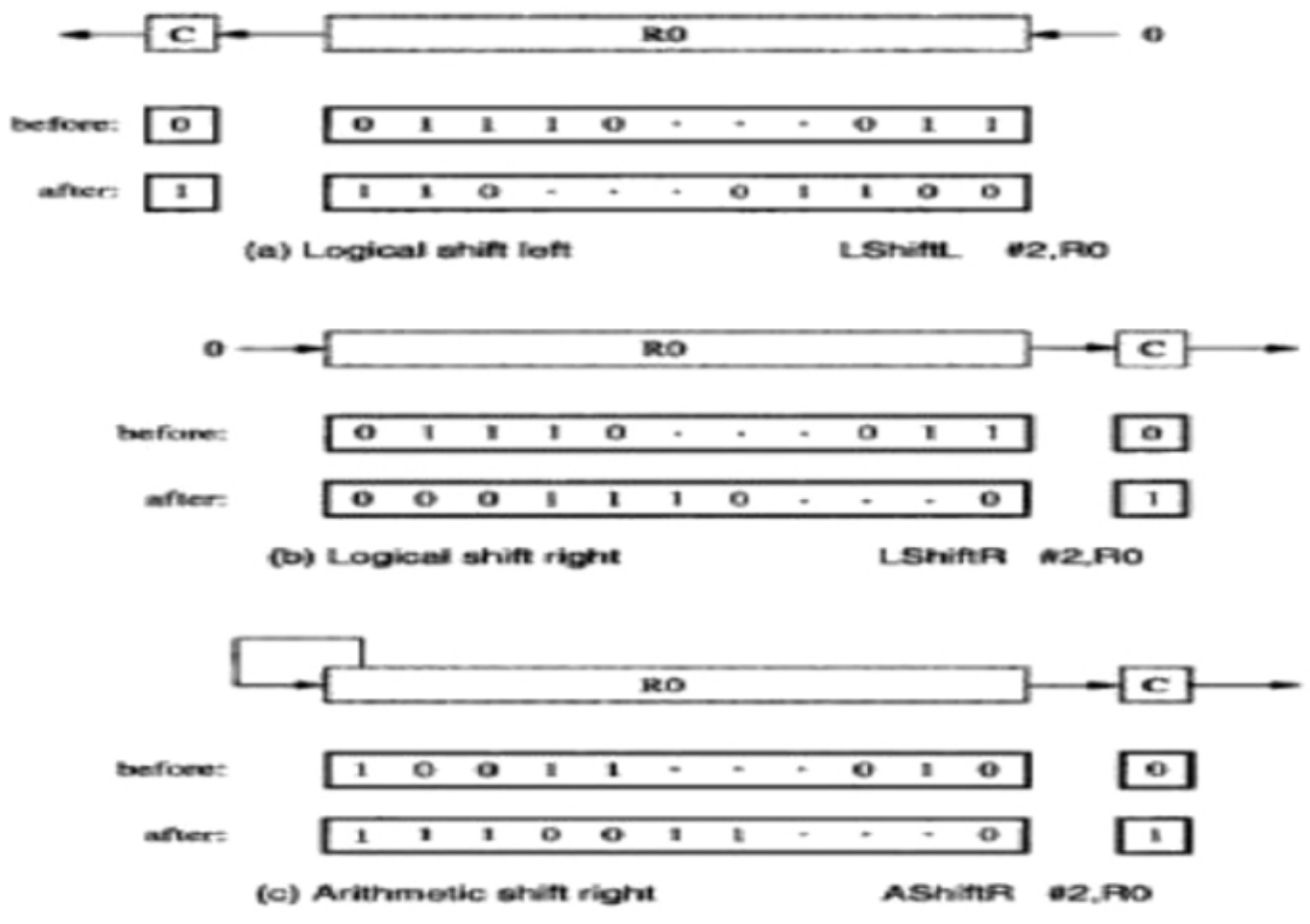
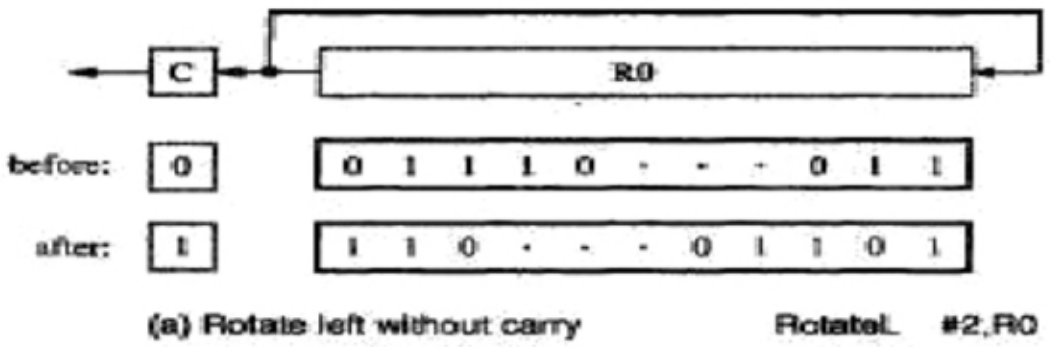


Figure 2.30 Logical and arithmetic shift instructions.

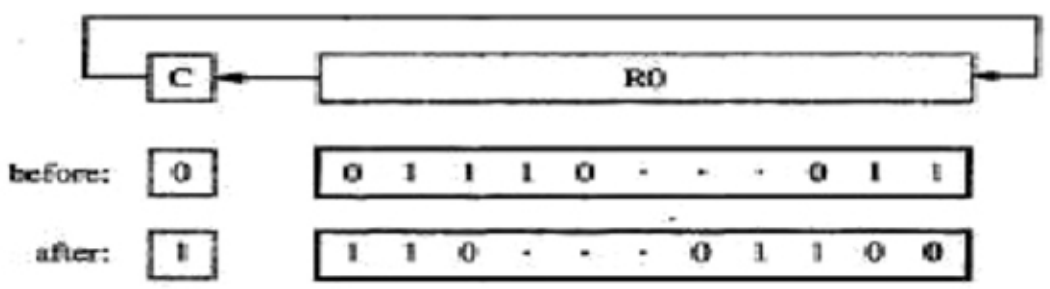
```
Move        #LOC,R0       R0 points to data.
MoveByte    (R0)+,R1      Load first byte into R1.
LShiftL     #4,R1         Shift left by 4 bit positions.
MoveByte    (R0),R2       Load second byte into R2.
And         #$F,R2        Eliminate high-order bits.
Or          R1,R2         Concatenate the BCD digits.
MoveByte    R2,PACKED     Store the result.
```

ROTATE OPERATIONS

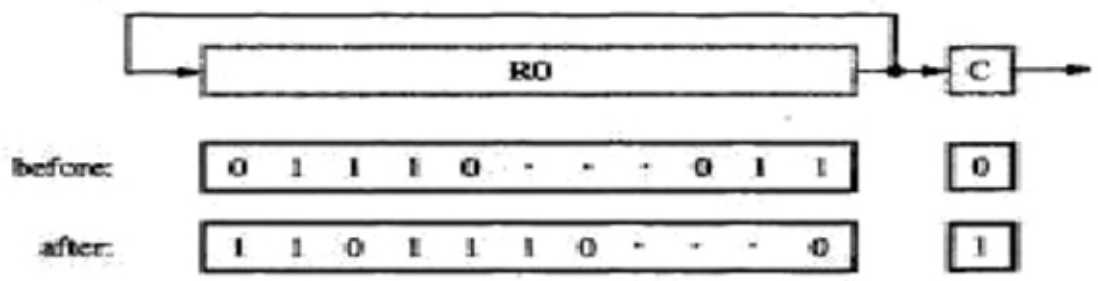
- In shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the Carry-flag C.
- To preserve all bits, a set of rotate instructions can be used.
- They move the bits that are shifted out of one end of the operand back into the other end.
- Two versions of both the left and right rotate instructions are usually provided.
  In one version, the bits of the operand are simply rotated. In the other version, the rotation includes the C flag.



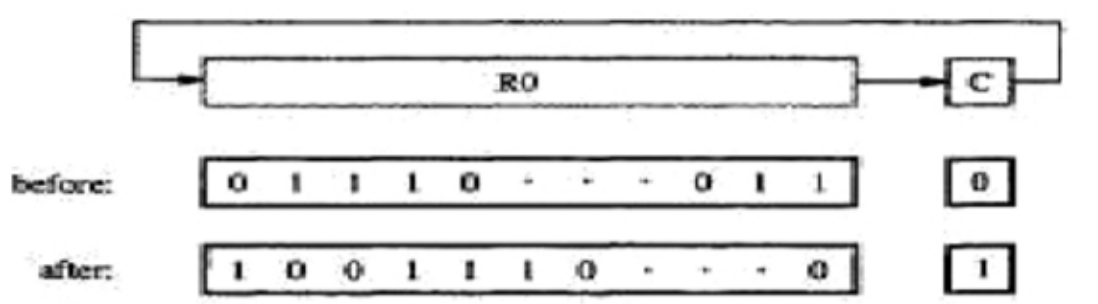
(a) Rotate left without carry RotateL #2,R0



(b) Rotate left with carry RotateLC #2,R0



(c) Rotate right without carry RotateR #2,R0



(d) Rotate right with carry RotateRC #2,R0

Figure 2.32 Rotate instructions.

ENCODING OF MACHINE INSTRUCTIONS

- To be executed in a processor, an instruction must be encoded in a binary-pattern. Such encoded instructions are referred to as *machine instructions*.

- The instructions that use symbolic names and acronyms are called *assembly language instructions*.
- We have seen instructions that perform operations such as add, subtract, move, shift, rotate, and branch. These instructions may use operands of different sizes, such as 32-bit and 8-bit numbers.

Let us examine some typical cases.

The instruction

Add R1, R2

Has to specify the registers R1 and R2, in addition to the OP code. If the processor has 16 registers, then four bits are needed to identify each register. Additional bits are needed to indicate that the Register addressing-mode is used for each operand.

The instruction

Move 24(R0), R5

Requires 16 bits to denote the OP code and the two registers, and some bits to express that the source operand uses the Index addressing mode and that the index value is 24.

- In all these examples, the instructions can be encoded in a 32-bit word (Fig 2.39).
- The OP code for given instruction refers to type of operation that is to be performed.
- Source and destination field refers to source and destination operand respectively.
- The "Other info" field allows us to specify the additional information that may be needed such as an index value or an immediate operand.
- Using multiple words, we can implement complex instructions, closely resembling operations in high-level programming languages. The term complex instruction set computers (CISC) refers to processors that use
- CISC approach results in instructions of variable length, dependent on the number of operands and the type of addressing modes used.
- In RISC (reduced instruction set computers), any instruction occupies only one word.
- The RISC approach introduced other restrictions such as that all manipulation of data must be done on operands that are already in processor registers.

Example

Add R1,R2,R3

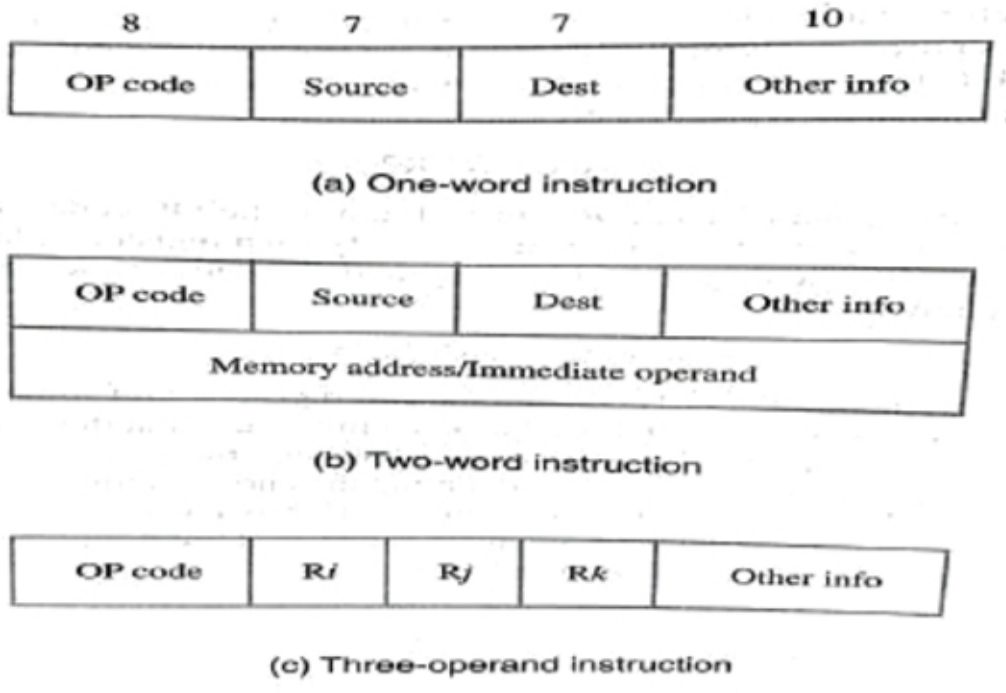


Figure 2.39 Encoding instructions into 32-bit words.

- In RISC type machine, the memory references are limited to only Load/Store operations.