MODULE 1: BASIC STRUCTURE OF COMPUTERS

BASIC CONCEPTS

• **Computer Architecture (CA)** is concerned with the structure and behaviour of the computer.
• CA includes the information formats, the instruction set and techniques for addressing memory.
• In general covers, CA covers 3 aspects of computer-design namely: 1) Computer Hardware, 2) Instruction set Architecture and 3) Computer Organization.

1. Computer Hardware

It consists of electronic circuits, displays, magnetic and optical storage media and communication facilities.

2. Instruction Set Architecture

It is programmer visible machine interface such as instruction set, registers, memory organization and exception handling.

Two main approaches are 1) CISC and 2) RISC.

(CISCComplex Instruction Set Computer, RISCReduced Instruction Set Computer)

3. Computer Organization

It includes the high level aspects of a design, such as

→ memory-system
→ bus-structure &
→ design of the internal CPU.

It refers to the operational units and their interconnections that realize the architectural specifications.

It describes the function of and design of the various units of digital computer that store and process information.

FUNCTIONAL UNITS

• A computer consists of 5 functionally independent main parts:

1) Input
2) Memory
3) ALU
4) Output &
5) Control units.

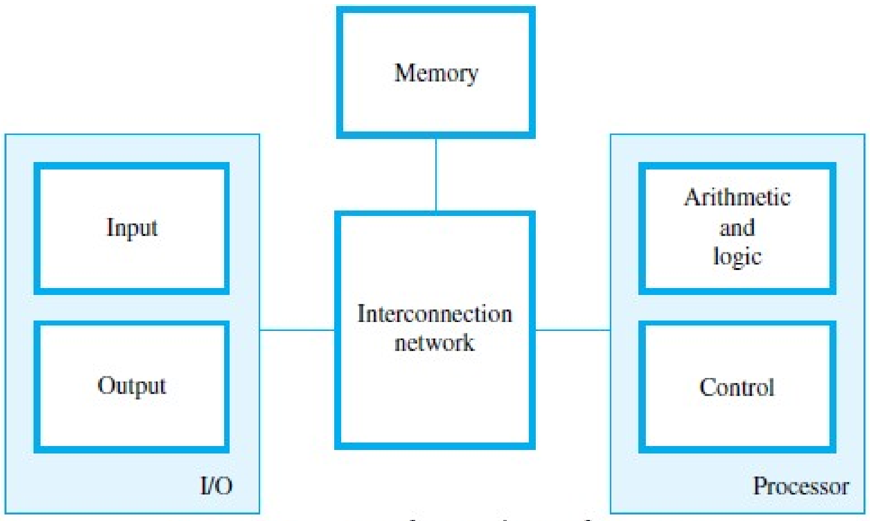


Figure 1.1 Basic functional units of a computer.

BASIC OPERATIONAL CONCEPTS

• An Instruction consists of 2 parts, 1) Operation code (Opcode) and 2) Operands.

OPCODE	*OPERANDS*

• The data/operands are stored in memory.
• The individual instruction are brought from the memory to the processor.
• Then, the processor performs the specified operation.
• Let us see a typical instruction

ADD LOCA, R0

• This instruction is an addition operation. The following are the steps to execute the instruction:

Step 1: Fetch the instruction from main-memory into the processor.
Step 2: Fetch the operand at location LOCA from main-memory into the processor.
Step 3: Add the memory operand (i.e. fetched contents of LOCA) to the contents of register R0.
Step 4: Store the result (sum) in R0.

• The same instruction can be realized using 2 instructions as:

Load LOCA, R1
Add R1, R0

• The following are the steps to execute the instruction:

Step 1: Fetch the instruction from main-memory into the processor.
Step 2: Fetch the operand at location LOCA from main-memory into the register R1.
Step 3: Add the content of Register R1 and the contents of register R0.
Step 4: Store the result (sum) in R0.

MAIN PARTS OF PROCESSOR

• The **processor** contains ALU, control-circuitry and many registers.
• The processor contains „n" general-purpose registers **R_0** through **R_{n-1}**.
• The **IR** holds the instruction that is currently being executed.
• The **control-unit** generates the timing-signals that determine when a given action is to take place.
• The **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 **MAR** holds the address of the memory-location to be accessed.
• The **MDR** contains the data to be written into or read out of the addressed location.
• MAR and MDR facilitates the communication with memory.

(IR Instruction-Register, PC Program Counter)
(MAR Memory Address Register, MDR Memory Data Register)

STEPS TO EXECUTE AN INSTRUCTION

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.
10) At some point during execution, contents of PC are incremented to point to next instruction in the program.

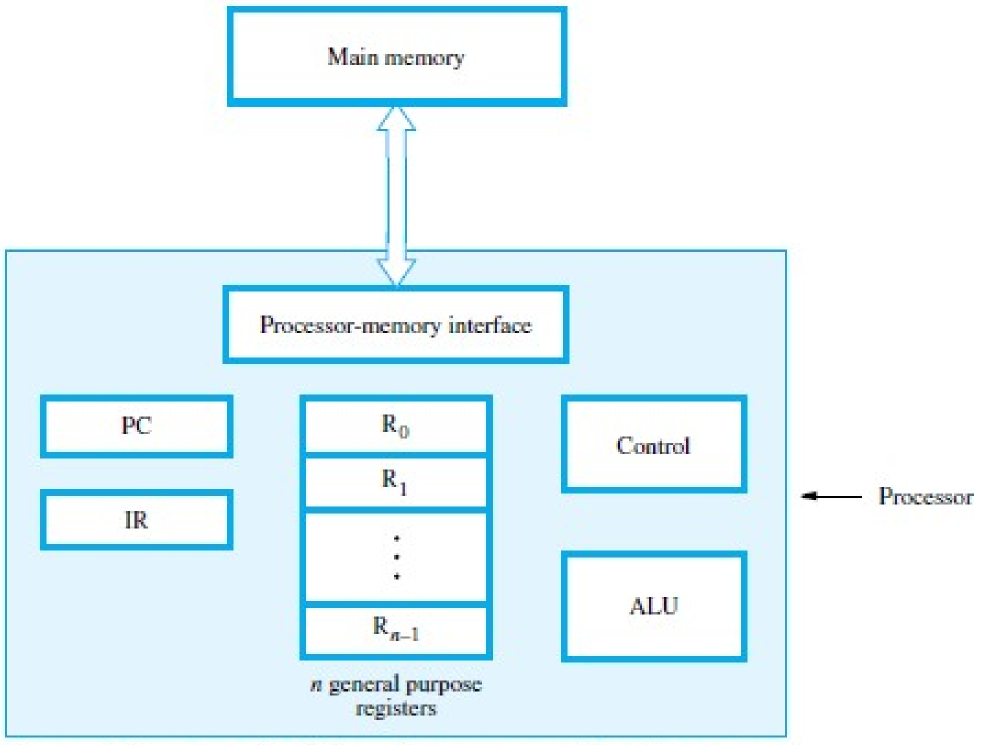


Figure 1.2 Connection between the processor and the main memory.

BUS STRUCTURE

- A bus is a group of lines that serves as a connecting path for several devices.
- A bus may be lines or wires.
- The lines carry data or address or control signal.
- There are 2 types of Bus structures: 1) Single Bus Structure and 2) Multiple Bus Structure.

1) Single Bus Structure

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.

Advantages:

1) Low cost &
2) Flexibility for attaching peripheral devices.

2) Multiple Bus Structure

Systems that contain multiple buses achieve more concurrency in operations.

Two or more transfers can be carried out at the same time.

Advantage: Better performance.

Disadvantage: Increased cost.

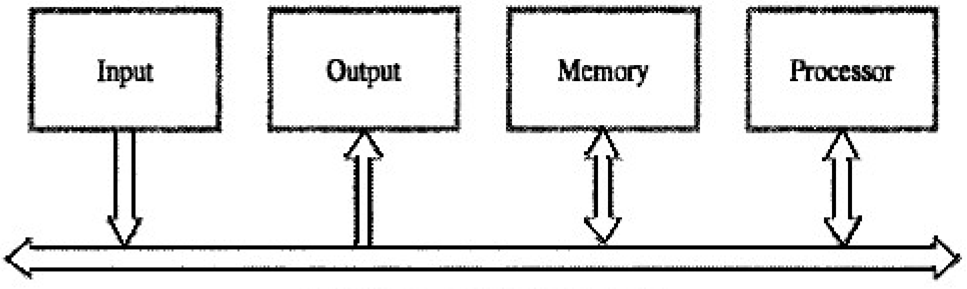


Figure 1.3 Single-bus structure.

- The devices connected to a bus vary widely in their speed of operation.
- To synchronize their operational-speed, buffer-registers can be used.
- **Buffer Registers**

  → are included with the devices to hold the information during transfers.

  → prevent a high-speed processor from being locked to a slow I/O device during data transfers.

PERFORMANCE

• The most important measure of performance of a computer is how quickly it can execute programs.
• The speed of a computer is affected by the design of

1) Instruction-set.
2) Hardware & the technology in which the hardware is implemented.
3) Software including the operating system.

• Because programs are usually written in a HLL, performance is also affected by the compiler that translates programs into machine language. (HLL High Level Language).
• For best performance, it is necessary to design the compiler, machine instruction set and hardware in a co-ordinated way.

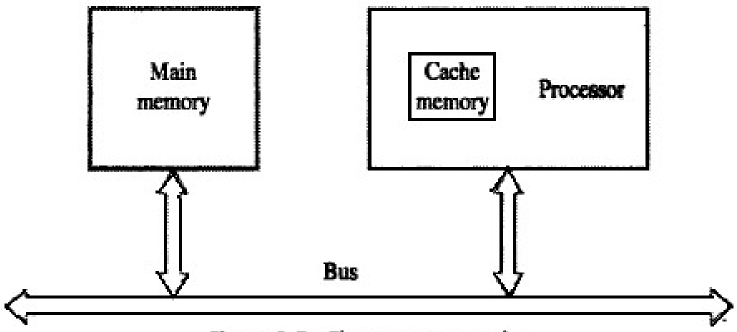


Figure 1.5 The processor cache.

• Let us examine the flow of program instructions and data between the memory & the processor.
• At the start of execution, all program instructions are stored in the main-memory.
• As execution proceeds, instructions are fetched into the processor, and a copy is placed in the cache.
• Later, if the same instruction is needed a second time, it is read directly from the cache.
• A program will be executed faster
  if movement of instruction/data between the main-memory and the processor is minimized
    which is achieved by using the cache.

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

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

• R 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

$$T = \frac{N \times S}{R} \qquad \text{------(1)}$$

• Equ1 is referred to as the basic performance equation.
• 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 values since changes in one parameter may affect the other.

CLOCK RATE

• There are 2 possibilities for increasing the clock rate R:

1) Improving the IC technology makes logic-circuits faster.

This reduces the time needed to compute a basic step. (IC integrated circuits).

This allows the clock period P to be reduced and the clock rate R to be increased.

2) Reducing the amount of processing done in one basic step also reduces the clock period P.

• In presence of a cache, the percentage of accesses to the main-memory is small.

Hence, much of performance-gain expected from the use of faster technology can be realized.

The value of T will be reduced by same factor as R is increased „.“ S & N are not affected.

PERFORMANCE MEASUREMENT

• Benchmark refers to standard task used to measure how well a processor operates.

• The Performance Measure is the time taken by a computer to execute a given benchmark.

• SPEC selects & publishes the standard programs along with their test results for different application domains. (SPEC System Performance Evaluation Corporation).

• SPEC Rating is given by

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

• SPEC rating = 50 The computer under test is 50 times as fast as reference-computer.

• The test is repeated for all the programs in the SPEC suite.

Then, the geometric mean of the results is computed.

• Let SPEC_i = Rating for program „i' in the suite.

Overall SPEC rating for the computer is given by

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

where n = no. of programs in the suite.

INSTRUCTION SET: CISC AND RISC

RISC	CISC
Simple instructions taking one cycle.	Complex instructions taking multiple cycle.
Few instructions.	Many instructions.
Fixed format instructions.	Variable format instructions.
	Many addressing modes.
Multiple register set.	Single register set.
Highly pipelined.	No pipelined or less pipelined.

Problem 1:

List the steps needed to execute the machine instruction:

Load R2, LOC

in terms of transfers between the components of processor and some simple control commands. Assume that the address of the memory-location containing this instruction is initially in register PC.

Solution:

1. Transfer the contents of register PC to register MAR.
2. Issue a Read command to memory.
   And, then wait until it has transferred the requested word into register MDR.
3. Transfer the instruction from MDR into IR and decode it.
4. Transfer the address LOCA from IR to MAR.
5. Issue a Read command and wait until MDR is loaded.
6. Transfer contents of MDR to the ALU.
7. Transfer contents of R0 to the ALU.
8. Perform addition of the two operands in the ALU and transfer result into R0.
9. Transfer contents of PC to ALU.
10. Add 1 to operand in ALU and transfer incremented address to PC.

List the steps needed to execute the machine instruction:

Add R4, R2, R3

in terms of transfers between the components of processor and some simple control commands. Assume that the address of the memory-location containing this instruction is initially in register PC.

Solution:

1. Transfer the contents of register PC to register MAR.
2. Issue a Read command to memory.
   And, then wait until it has transferred the requested word into register MDR.
3. Transfer the instruction from MDR into IR and decode it.
4. Transfer contents of R1 and R2 to the ALU.
5. Perform addition of two operands in the ALU and transfer answer into R3.
6. Transfer contents of PC to ALU.
7. Add 1 to operand in ALU and transfer incremented address to PC.

Problem 3:

(a) Give a short sequence of machine instructions for the task "Add the contents of memory-location A to those of location B, and place the answer in location C". Instructions:

Load R*i*, LOC
and
Store R*i*, LOC

are the only instructions available to transfer data between memory and the general purpose registers. Add instructions are described in Section 1.3. Do not change contents of either location A or B.

(b) Suppose that Move and Add instructions are available with the formats:

Move Location1, Location2
and
Add Location1, Location2

These instructions move or add a copy of the operand at the second location to the first location, overwriting the original operand at the first location. Either or both of the operands can be in the memory or the general-purpose registers. Is it possible to use fewer instructions of these types to accomplish the task in part (a)? If yes, give the sequence.

Solution:

(a)

```
Load A, R0
Load B, R1
Add R0, R1
Store R1, C
```

(b) Yes;

```
Move B, C
Add A, C
```

Problem 4:

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

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)/1X10_9 =(1000+2000+1050)/1*10_9= 4.05 μs.

Problem 5:

For the following processor, obtain the performance.

Clock rate = 800 MHz

No. of instructions executed = 1000

Average no of steps needed / machine instruction = 20

Solution:

$$T = \frac{N \times S}{R} = (1000*20)/800 * 10^6 = 25 \text{ micro sec or } 25*10^{-6} \text{ sec}$$

Problem 6:

(a) Program execution time T is to be examined for a certain high-level language program. The program can be run on a RISC or a CISC computer. Both computers use pipelined instruction execution, but pipelining in the RISC machine is more effective than in the CISC machine. Specifically, the effective value of S in the T expression for the RISC machine is 1.2, bit it is only 1.5 for the CISC machine. Both machines have the same clock rate R. What is the largest allowable value for N, the number of instructions executed on the CISC machine, expressed as a percentage of the N value for the RISC machine, if time for execution on the CISC machine is to be longer than on the RISC machine?

(b) Repeat Part (a) if the clock rate R for the RISC machine is 15 percent higher than that for the CISC machine.

Solution:

(a) Let $T_R = (N_R \times S_R)/R_R$ & $T_C = (N_C \times S_C)/R_C$ be execution times on RISC and CISC processors. Equating execution times and clock rates, we have

$1.2N_R = 1.5N_C$

Then $N_C/N_R = 1.2/1.5 = 0.8$

Therefore, the largest allowable value for N_C is 80% of N_R.

(b) In this case,

$1.2N_R/1.15 = 1.5N_C/1.00$

Then $N_C/N_R = 1.2/(1.15 \times 1.5) = 0.696$

Therefore, the largest allowable value for N_C is 69.6% of N_R.

Problem 7:

(a) Suppose that execution time for a program is proportional to instruction fetch time. Assume that fetching an instruction from the cache takes 1 time unit, but fetching it from the main-memory takes 10 time units. Also, assume that a requested instruction is found in the cache with probability 0.96. Finally, assume that if an instruction is not found in the cache it must first be fetched from the main-memory into the cache and then fetched from the cache to be executed. Compute the ratio of program execution time without the cache to program execution time with the cache. This ratio is called the speedup resulting from the presence of the cache.

(b) If the size of the cache is doubled, assume that the probability of not finding a requested instruction there is cut in half. Repeat part (a) for a doubled cache size.

Solution:

(a) Let cache access time be 1 and main-memory access time be 20. Every instruction that is executed must be fetched from the cache, and an additional fetch from the main-memory must be performed for 4% of these cache accesses.

Therefore,

$$\text{Speedup factor} = \frac{1.0 \times 20}{(1.0 \times 1) + (0.04 \times 20)} = 11.1$$

(b)

$$\text{Speedup factor} = \frac{1.0 \times 20}{(1.0 \times 1) + (0.02 \times 20)} = 16.7$$

MODULE 1 (CONT.): MACHINE INSTRUCTIONS & PROGRAMS

MEMORY-LOCATIONS & ADDRESSES

- **Memory** consists of many millions of storage cells (flip-flops).
- Each cell can store a bit of information i.e. 0 or 1 (Figure 2.1).
- 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 64 bits.
- A unit of 8 bits is called a **byte**.
- Accessing the memory to store or retrieve a single item of information (word/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 = no. 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 (16 MB).

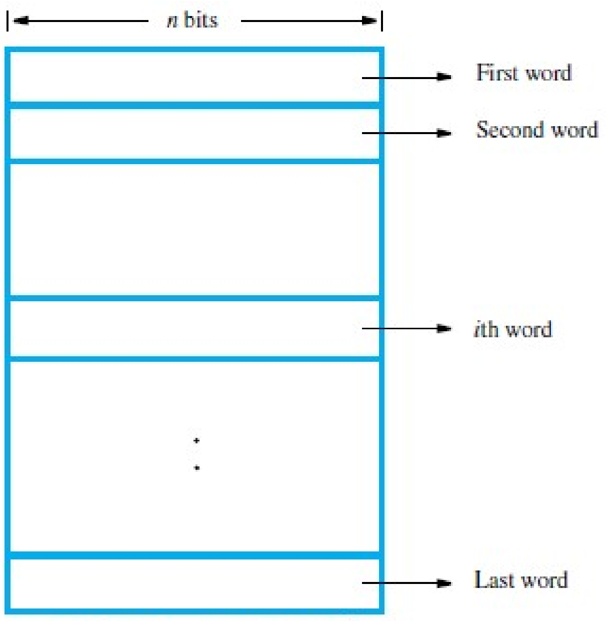


Figure 2.1 Memory words.

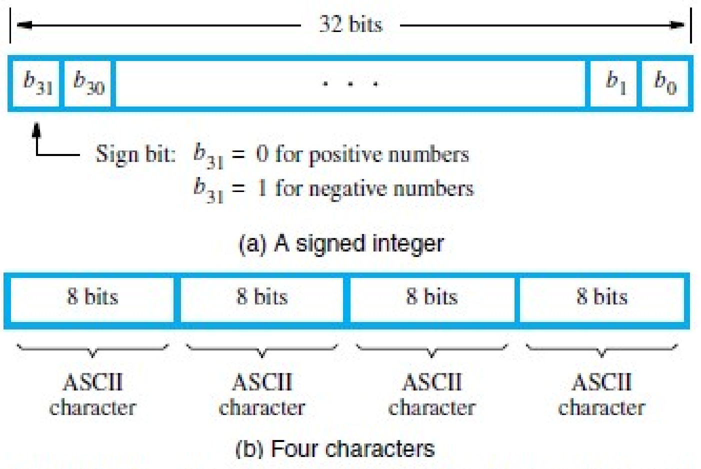


Figure 2.2 Examples of encoded information in a 32-bit word.

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 & LITTLE-ENDIAN ASSIGNMENTS

- There are two ways in which byte-addresses are arranged (Figure 2.3).
  1) **Big-Endian:** Lower byte-addresses are used for the more significant bytes of the word.
  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.

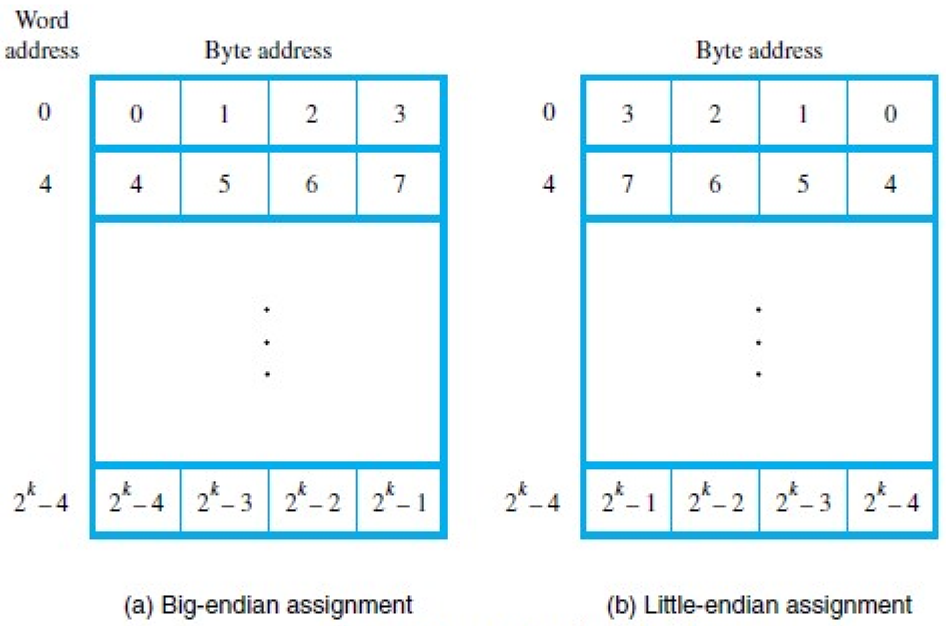

Word address
Byte address
0
0 1 2 3
4
4 5 6 7
2^k-4
2^k-4 2^k-3 2^k-2 2^k-1
(a) Big-endian assignment
Byte address
0
3 2 1 0
4
7 6 5 4
2^k-4
2^k-1 2^k-2 2^k-3 2^k-4
(b) Little-endian assignment

Figure 2.3 Byte and word addressing.

- Consider a 32-bit integer (in hex): 0**x**12345678 which consists of 4 bytes: 12, 34, 56, and 78.

  Hence this integer will occupy 4 bytes in memory.
  Assume, we store it at memory address starting 1000.
  On little-endian, memory will look like

Address	Value
1000	78
1001	56
1002	34
1003	12

  On big-endian, memory will look like

Address	Value
1000	12
1001	34
1002	56
1003	78

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 . . . . .
  If the word length is 64(2 bytes), 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:

1) A special control character with the meaning "end of string" can be used as the last character in the string.
2) A separate memory word location or register can contain a number indicating the length of the string in bytes.

MEMORY OPERATIONS

• Two memory operations are:

1) Load (Read/Fetch) &
2) Store (Write).

• The **Load** operation transfers a copy of the contents of a specific memory-location to the processor. The memory contents remain unchanged.

• 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 register to the specified memory-location. This will destroy the original contents of that memory-location.

• 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 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/0 transfers (IN, OUT).

REGISTER TRANSFER NOTATION (RTN)

• The possible locations in which transfer of information occurs are: 1) Memory-location 2) Processor register & 3) Registers in I/O device.

Location	Hardware Binary Address	Example	Description
Memory	LOC, PLACE, NUM	R1 [LOC]	Contents of memory-location LOC are transferred into register R1.
Processor	R0, R1 ,R2	[R3] [R1]+[R2]	Add the contents of register R1 &R2 and places their sum into R3.
I/O Registers	DATAIN, DATAOUT	R1 DATAIN	Contents of I/O register DATAIN are transferred into register R1.

ASSEMBLY LANGUAGE NOTATION

• To represent machine instructions and programs, assembly language format is used.

Assembly Language Format	Description
Move LOC, R1	Transfer data from memory-location LOC to 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	Add the contents of registers R1 and R2, and places their sum into register R3.

BASIC INSTRUCTION TYPES

Instruction Type	Syntax	Example	Description	Instructions for Operation C<-[A]+[B]
Three Address	Opcode Source1,Source2,Destination	Add A,B,C	Add the contents of memory-locations A & B. Then, place the result into location C.	
Two Address	Opcode Source, Destination	Add A,B	Add the contents of memory-locations A & B. Then, place the result into location B, replacing the original contents of this location. Operand B is both a source and a destination.	Move B, C Add A, C
One Address	Opcode Source/Destination	Load A	Copy contents of memory-location A into accumulator.	Load A Add B Store C
		Add B	Add contents of memory-location B to contents of accumulator register & place sum back into accumulator.	
		Store C	Copy the contents of the accumulator into location C.	
Zero Address	Opcode [no Source/Destination]	Push	Locations of all operands are defined implicitly. The operands are stored in a pushdown stack.	Not possible

• Access to data in the registers is much faster than to data stored in memory-locations.

• Let Ri represent a general-purpose register. The instructions: *Load A,Ri*
Store Ri,A
Add A,Ri

are generalizations of the Load, Store and Add Instructions for the single-accumulator case, in which register Ri performs the function of the accumulator.

• In processors, where arithmetic operations as allowed only on operands that are in registers, the task C<-[A]+[B] 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 (Figure 2.8).

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*.

3) During the execution of each instruction, PC is incremented by 4 to point to next instruction.

• There are 2 phases for Instruction Execution:

1) Fetch Phase: The instruction is fetched from the memory-location and placed in the IR.

2) 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.

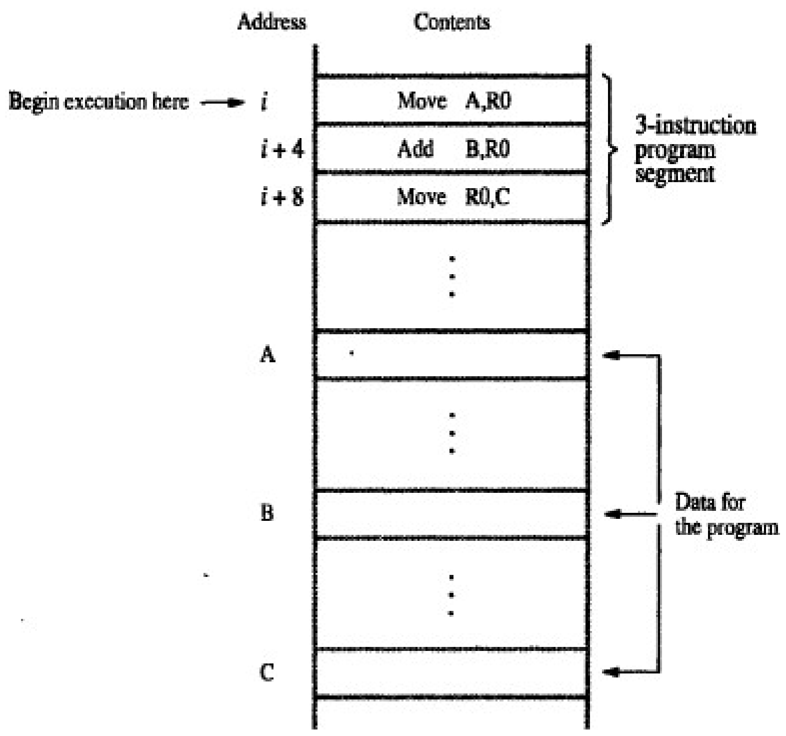


Figure 2.8 A program for C ← [A] + [B].

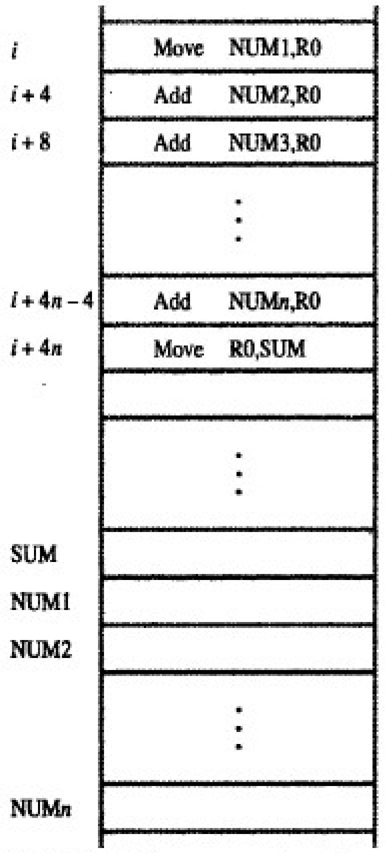


Figure 2.9 A straight-line program for adding *n* numbers.

Program Explanation

• Consider the program for adding a list of n numbers (Figure 2.9).

• The Address of the memory-locations containing the n numbers are symbolically given as NUM1, NUM2…..NUMn.

• Separate Add instruction is used to add each number to the contents of register R0.

• After all the numbers have been added, the result is placed in memory-location SUM.

BRANCHING
• Consider the task of adding a list of „n" numbers (Figure 2.10).
• Number of entries in the list „n" is stored in memory-location **N**.
• Register **R1** is used as a counter to determine the number of times the loop is executed.
• Content-location N is loaded into register R1 at the beginning of the program.
• The **Loop** is a straight line sequence of instructions executed as many times as needed.
    The loop starts at location LOOP and ends at the instruction Branch>0.
• During each pass,
    → address of the next list entry is determined and
    → that entry is fetched and added to R0.
• 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.

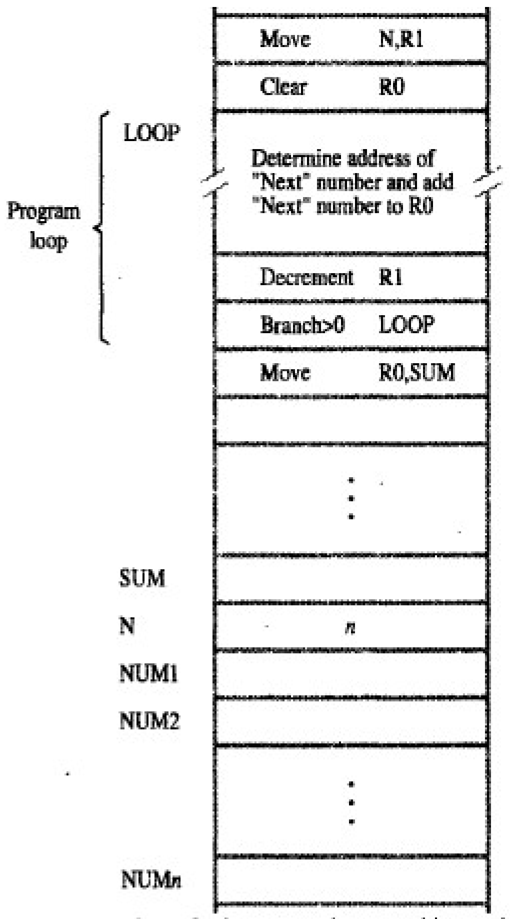


Figure 2.10 Using a loop to add *n* numbers.

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:
    1) N (negative) set to 1 if the result is negative, otherwise cleared to 0.
    2) Z (zero) set to 1 if the result is 0; otherwise, cleared to 0.
    3) V (overflow) set to 1 if arithmetic overflow occurs; otherwise, cleared to 0.
    4) 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** (Table 2.1).

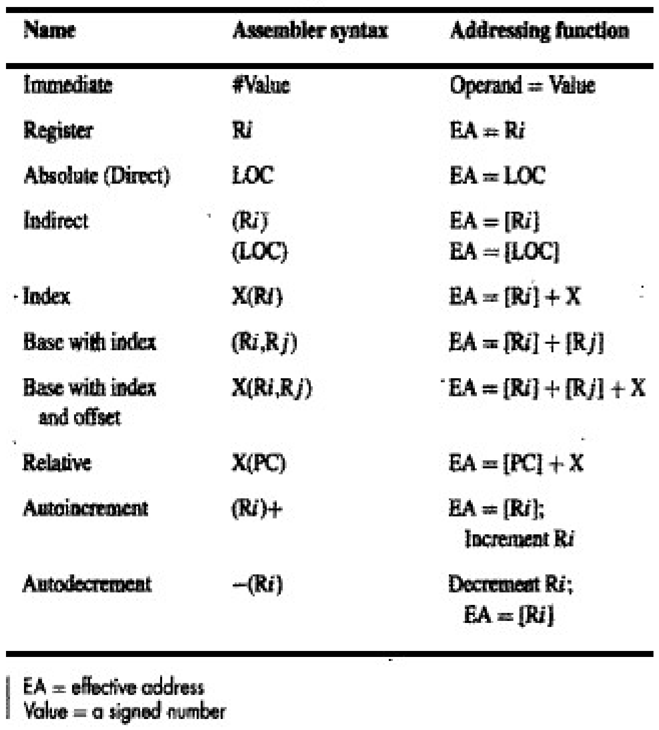


Table 2.1 Generic addressing modes

Name	Assembler syntax	Addressing function
Immediate	#Value	Operand = Value
Register	R*i*	EA = R*i*
Absolute (Direct)	LOC	EA = LOC

Indirect	(R*i*)
	(LOC)
		EA = [R*i*]
		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
• **Variable** is represented by allocating a memory-location to hold its value.
• Thus, the value can be changed as needed using appropriate instructions.
• There are 2 accessing modes to access the variables:
1) Register Mode
2) Absolute Mode

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.
• The absolute mode can represent global variables in the program.
• 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.
• For example, the instruction
Move #200, R0 ;Place the value 200 in register R0.
• Clearly, the immediate mode is only used to specify the value of a source-operand.

INDIRECTION AND POINTERS

• Instruction does not give the operand or its address explicitly.

• Instead, the instruction provides information from which the new address of the operand can be determined.

• This address is called **Effective Address (EA)** of the operand.

Indirect Mode

• The EA of the operand is the contents of a register(or memory-location).

• The register (or memory-location) that contains the address of an operand is called a **Pointer**.

• We denote the indirection by

→ name of the register or

→ new address given in the instruction.

E.g: *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.

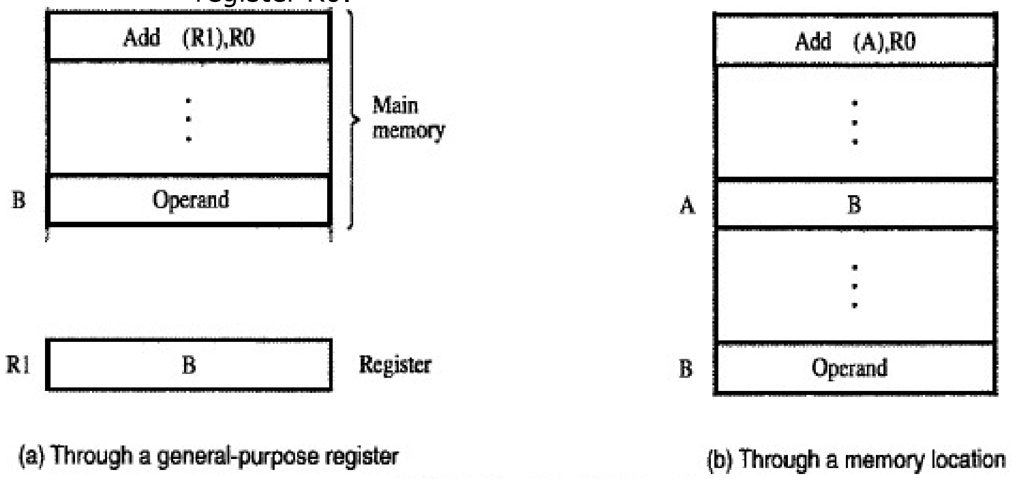

Add (R1),R0
Main memory
B
Operand
R1
B
Register
(a) Through a general-purpose register
Add (A),R0
A
B
B
Operand
(b) Through a memory location

Figure 2.11 Indirect addressing.

• To execute the Add instruction in fig 2.11 (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 2.11(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.

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 2.12 Use of indirect addressing in the program of Figure 2.10.

Program Explanation

• In above program, Register R2 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R2.

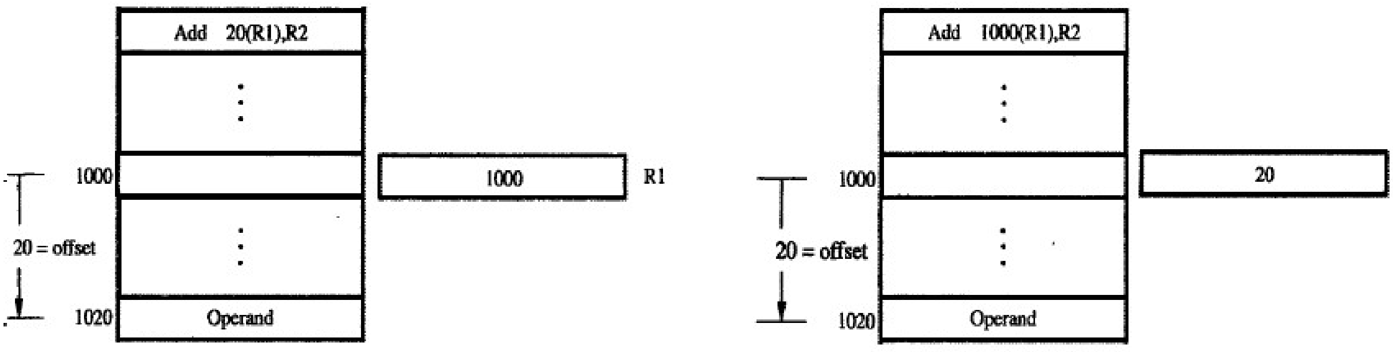
• The initialization-section of the program loads the counter-value n from memory-location N into R1 and uses the immediate addressing-mode to place the address value NUM1, which is the address of the first number in the list, into R2. Then it clears R0 to 0.

• 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 which defines an offset(also called a displacement).
Ri=the name of the index register which contains address of a new location.
• 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.
• The constant X may be given either
→ as an explicit number or
→ as a symbolic-name representing a numerical value.



(a) Offset is given as a constant (b) Offset is in the index register
Figure 2.13 Indexed addressing.

• Fig(a) illustrates two ways of using the Index mode. In fig(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.
• To find EA of operand:
Eg: Add 20(R1), R2
EA=>1000+20=1020
• An alternative use is illustrated in fig(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.

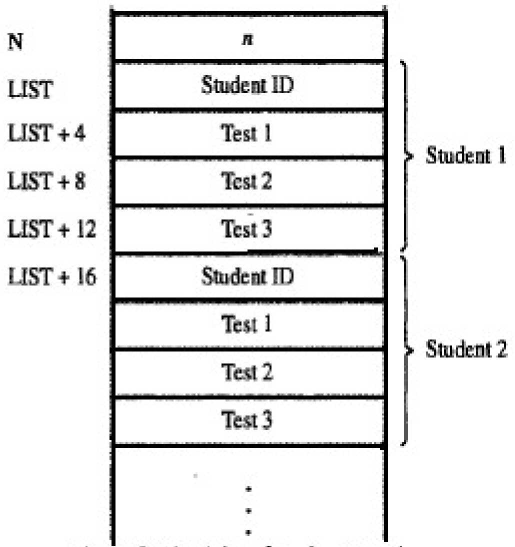
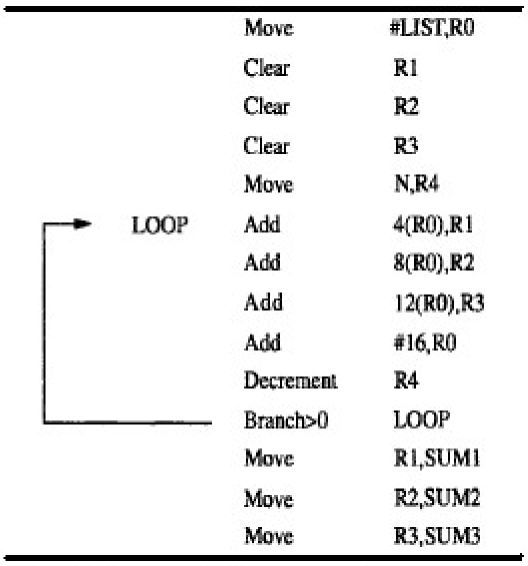


Figure 2.14 A list of students' marks.



```
                 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 one difference:
    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.

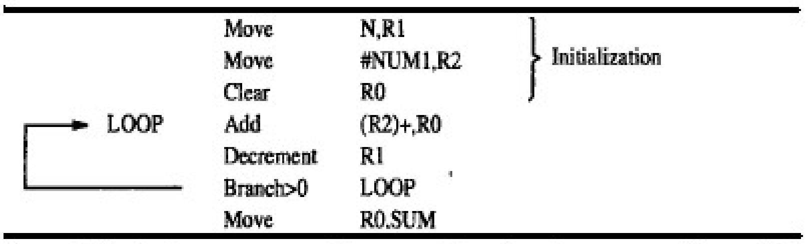
ADDITIONAL ADDRESSING MODES

1) Auto Increment Mode

Effective-address of operand is 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.

2) 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.



```
            Move        N,R1       }
            Move        #NUM1,R2   } Initialization
            Clear       R0         }
     LOOP   Add         (R2)+,R0
            Decrement   R1
            Branch>0    LOOP
            Move        R0,SUM
```

Figure 2.16 The Autoincrement addressing mode used in the program of Figure 2.12.

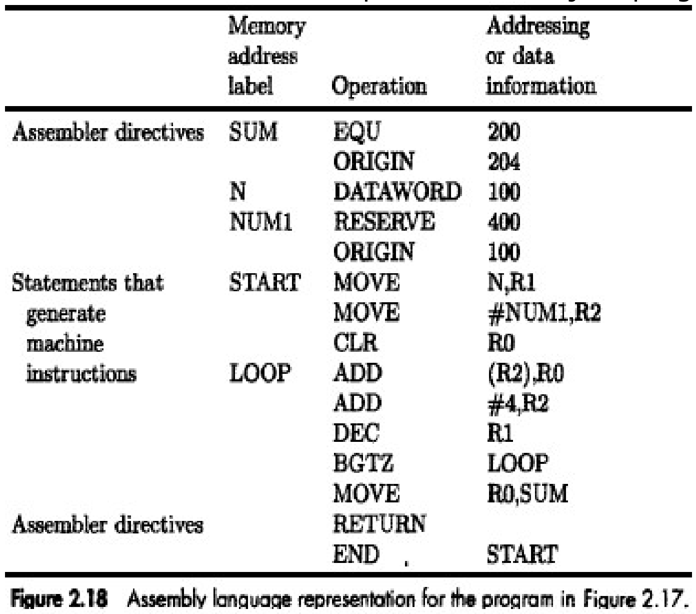
ASSEMBLY LANGUAGE

• We generally use symbolic-names to write a program.
• A complete set of symbolic-names 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**.
For example:

MOVE R0,SUM ;The term MOVE represents OP code for operation performed by instruction.
ADD #5,R3 ;Adds number 5 to contents of register R3 & puts the result back into registerR3.

ASSEMBLER DIRECTIVES

• **Directives** are the assembler commands to the assembler concerning the program being assembled.
• These commands are not translated into machine opcode in the object-program.



	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.

• **EQU** informs the assembler about the value of an identifier (Figure: 2.18).
Ex*: SUM EQU 200* ;Informs assembler that the name SUM should be replaced by the value 200.
• **ORIGIN** tells the assembler about the starting-address of memory-area to place the data block.
Ex*:* ORIGIN 204 ;Instructs assembler to initiate data-block at memory-locations starting from 204.
• **DATAWORD** directive tells the assembler to load a value into the location.
Ex: *N DATAWORD 100 ;*Informs the assembler to load data 100 into the memory-location N(204).
• **RESERVE** directive is used to reserve a block of memory.
Ex: *NUM1 RESERVE 400 ;*declares a memory-block of 400 bytes is to be reserved for data.
• **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*

1) Label is an optional name associated with the memory-address where the machine language instruction produced from the statement will be loaded.
2) Operation Field contains the OP-code mnemonic of the desired instruction or assembler.
3) Operand Field contains addressing information for accessing one or more operands, depending on the type of instruction.
4) Comment Field is used for documentation purposes to make 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.
→ replaces all names and labels with their actual values.
→ assigns addresses to instructions & data blocks, starting at address given in ORIGIN directive
→ inserts constants that may be given in DATAWORD directives.
→ reserves memory-space as requested by RESERVE directives.

• **Two Pass Assembler** has 2 passes:

1) First Pass: Work out all the addresses of labels.
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.
2) Second Pass: Generate machine code, substituting values for the labels.
When a name appears a second time in the source-program, it is replaced with its value from the table.

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

BASIC INPUT/OUTPUT OPERATIONS

• Consider the problem of moving a character-code from the keyboard to the processor (Figure: 2.19).
For this transfer, buffer-register DATAIN & a status control flags(SIN) are used.

• When a key is pressed, the corresponding ASCII code is stored in a **DATAIN** register associated with the keyboard.

SIN=1 When a character is typed in the keyboard. This informs the processor that a valid character is in DATAIN.

SIN=0 When the character is transferred to the processor.

• An analogous process takes place when characters are transferred from the processor to the display.
For this transfer, buffer-register DATAOUT & a status control flag SOUT are used.

SOUT=1 When the display is ready to receive a character.

SOUT=0 When the character is being transferred to DATAOUT.

• The buffer registers DATAIN and DATAOUT and the status flags SIN and SOUT are part of circuitry commonly known as a **device interface**.

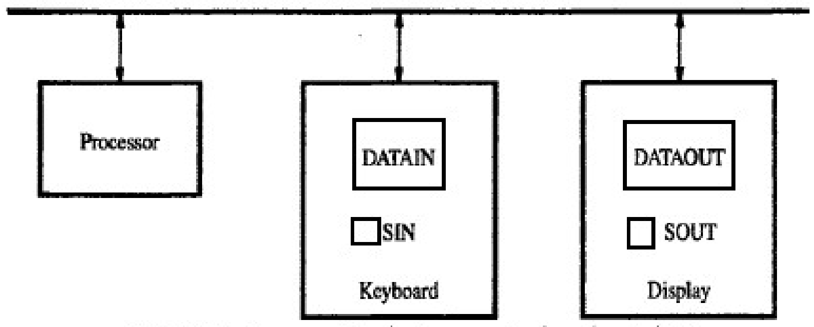
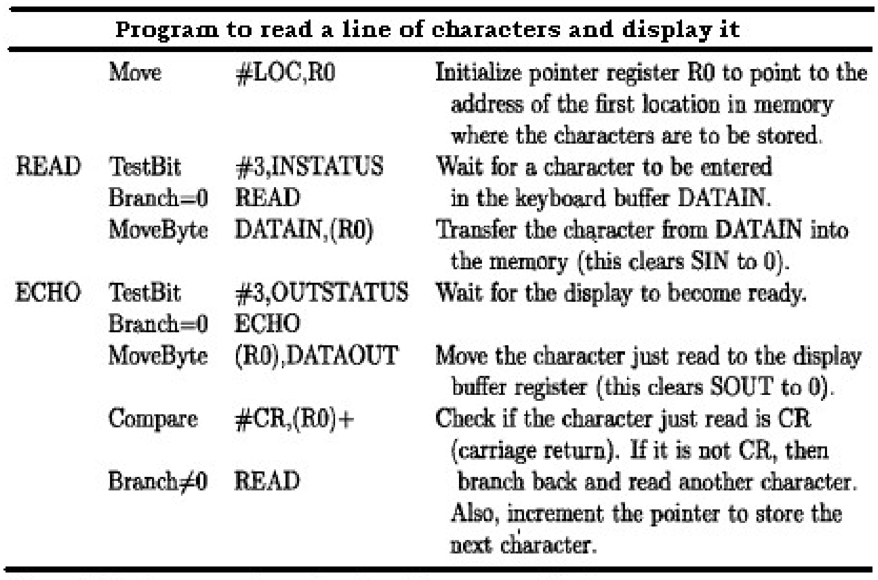


Figure 2.19 Bus connection for processor, keyboard, and display.



Program to read a line of characters and display 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 & 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 special type of data structure where elements are inserted from one end and elements are deleted from the same end. This end is called the **top** of the stack (Figure: 2.14).

• The various operations performed on stack:

1) Insert: An element is inserted from top end. Insertion operation is called **push** operation.
2) Delete: An element is deleted from top end. Deletion operation is called **pop** operation.

• 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)**.

• If we assume a byte-addressable memory with a 32-bit word length,

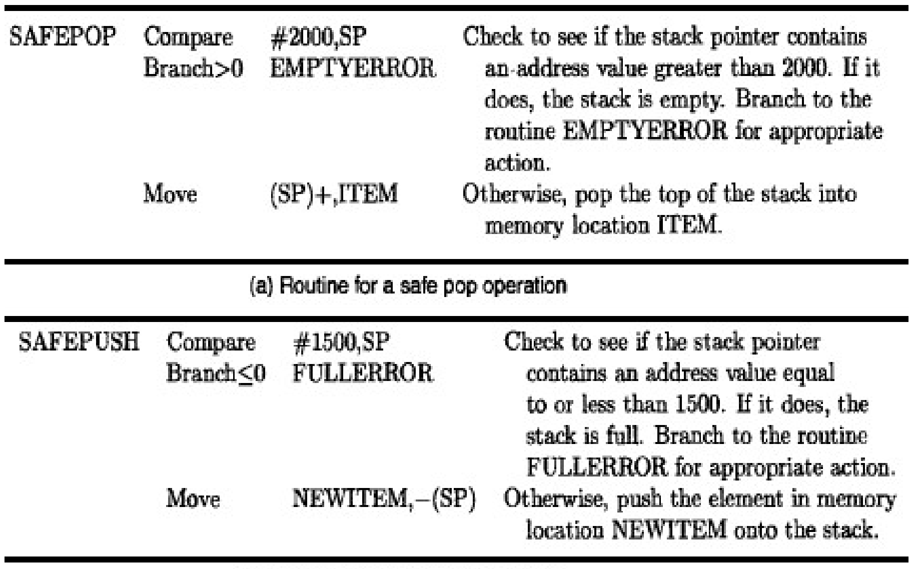
1) The push operation can be implemented as

Subtract #4, SP
Move NEWITEM, (SP)

2) 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.

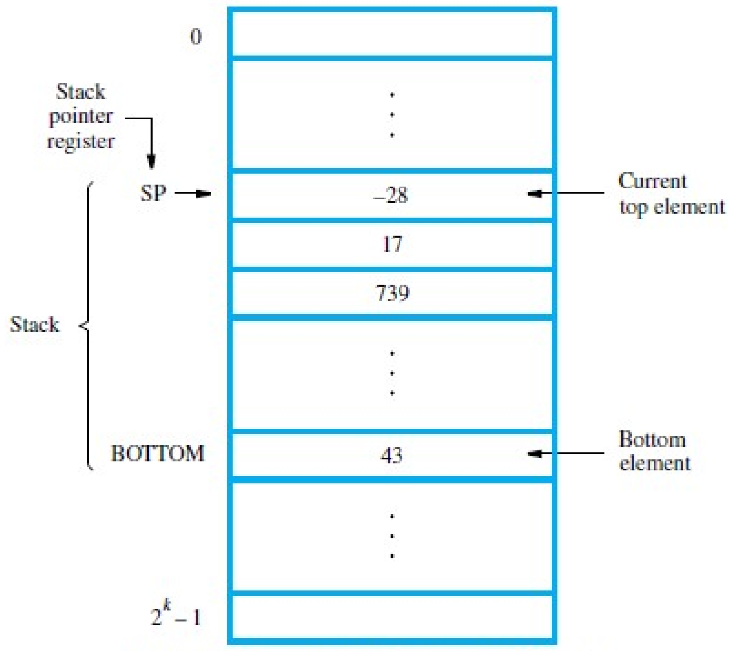


Figure 2.14 A stack of words in the memory.

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) In stack, a single pointer is needed to keep track of top of the stack at any given time.

In queue, two pointers are needed to keep track of both the front and end for removal and insertion respectively.

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**.

• A Call instruction causes a branch to the subroutine (Figure: 2.16).

• At the end of the subroutine, a return instruction is executed

• Program resumes execution at the instruction immediately following the subroutine call

• 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.

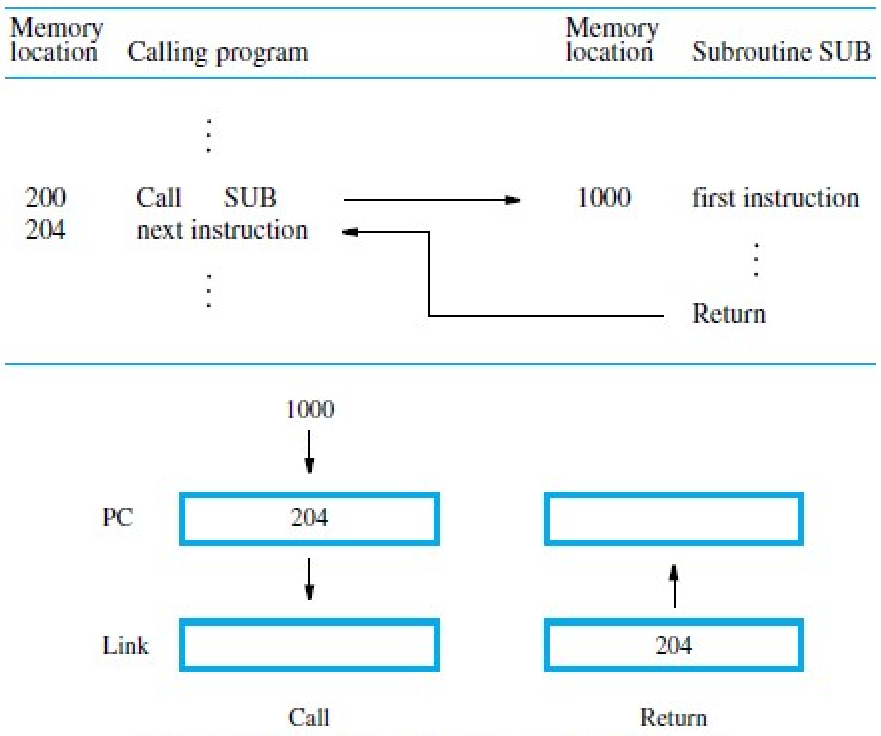


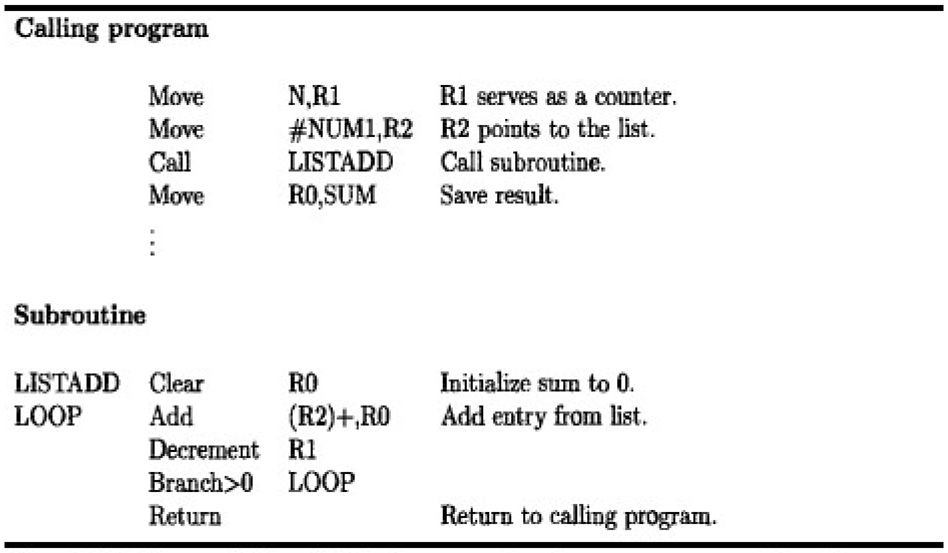
Figure 2.16 Subroutine linkage using a link register.

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.

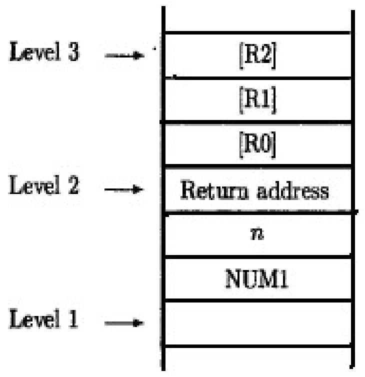
STACK FRAME

- **Stack Frame** refers to locations that constitute a private work-space for the subroutine.
- The work-space is
  → created at the time the subroutine is entered &
  → freed up when the subroutine returns control to the calling-program (Figure: 2.26).

Program for adding a list of numbers using subroutine with the parameters passed to stack.

```
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.

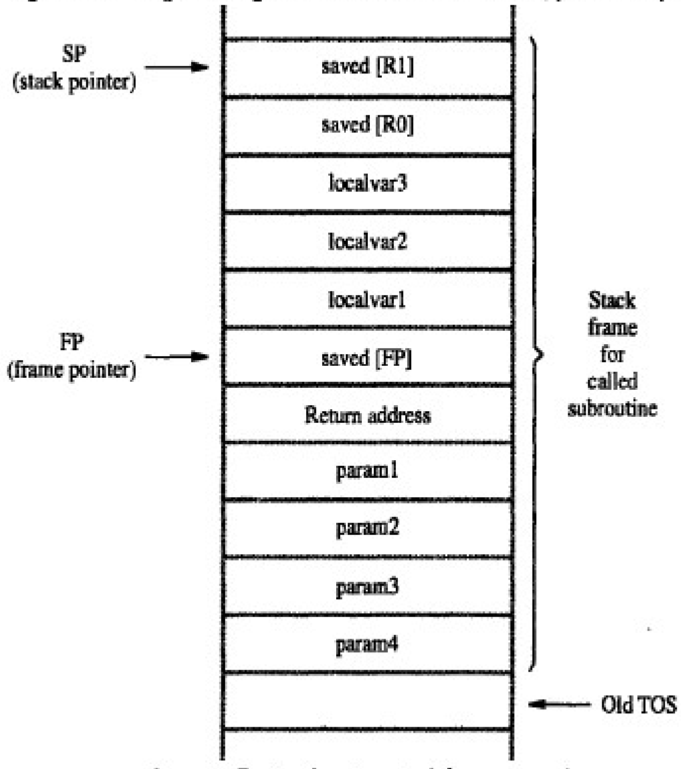


Figure 2.27 A subroutine stack frame example.

- 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 oldTOS.
• 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.

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.

Program to illustrate stack frames for nested subroutines

```
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.

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.

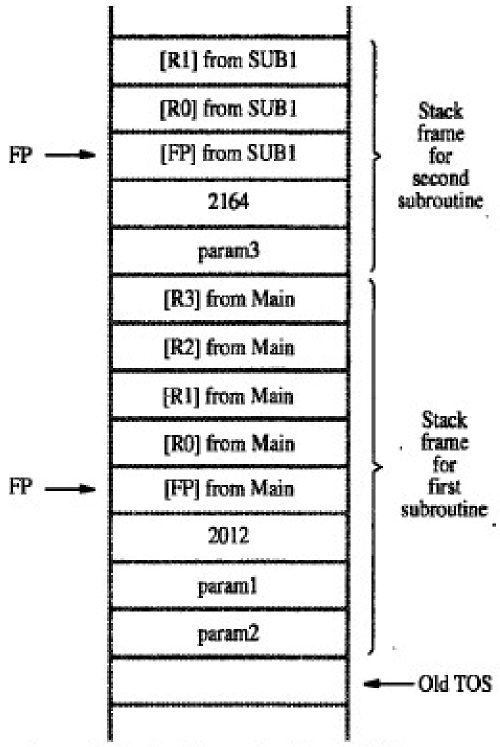


Figure 2.29 Stack frames for Figure 2.28.

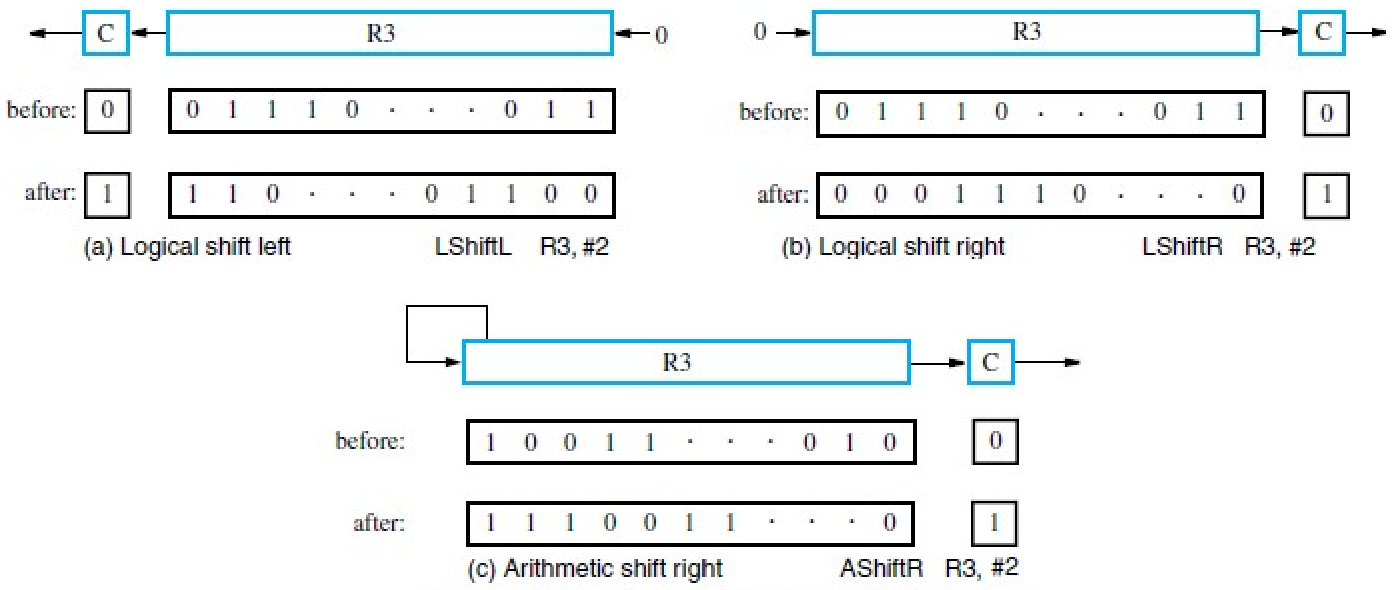
The Flow of Execution is as follows:

• 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.

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
1) Shifting left (LShiftL) &
2) Shifting right (LShiftR).
• These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction.

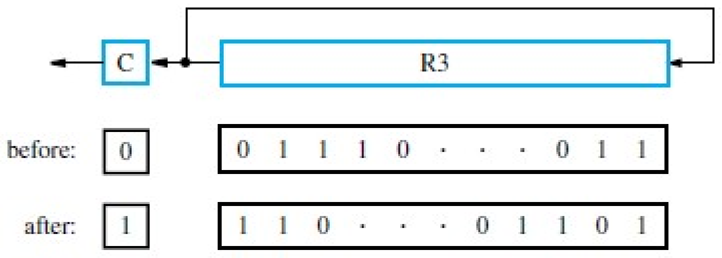

C
R3
0
before: 0 | 0 1 1 1 0 · · · 0 1 1
after: 1 | 1 1 0 · · · 0 1 1 0 0
(a) Logical shift left LShiftL R3, #2
0
R3
C
before: 0 1 1 1 0 · · · 0 1 1 | 0
after: 0 0 0 1 1 1 0 · · · 0 | 1
(b) Logical shift right LShiftR R3, #2
R3
C
before: 1 0 0 1 1 · · · 0 1 0 | 0
after: 1 1 1 0 0 1 1 · · · 0 | 1
(c) Arithmetic shift right AShiftR R3, #2
Figure 2.23 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.
MoveBvte    R2.PACKED     Store the result.
```

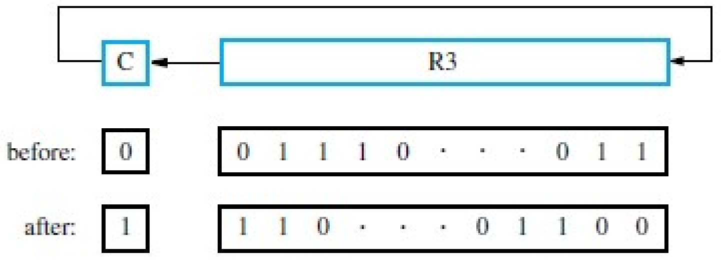
Figure 2.31 A routine that packs two BCD digits.

ROTATE OPERATIONS

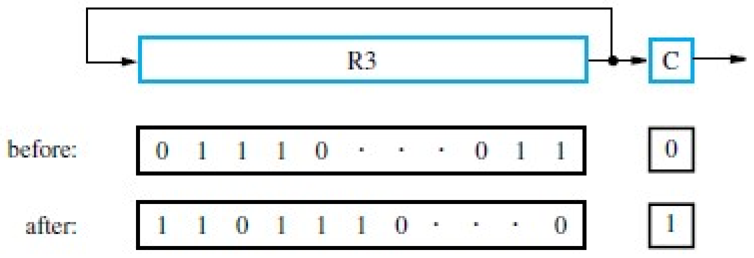
• In shift operations, the bits shifted out of the operand are lost, except for the last bit shifted out which is retained in the Carry-flag C.
• To preserve all bits, a set of rotate instructions can be used.
• They move the bits that are shifted out of one end of the operand back into the other end.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



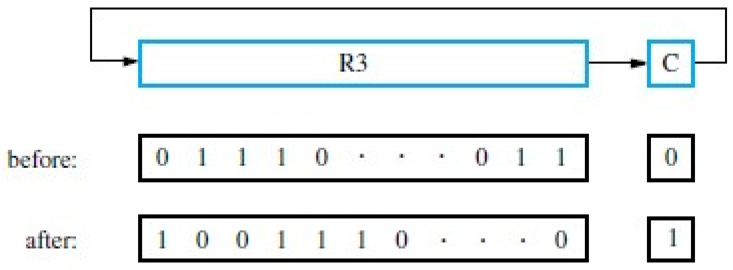
(a) Rotate left without carry RotateL R3, #2



(b) Rotate left with carry RotateLC R3, #2



(c) Rotate right without carry RotateR R3, #2



(d) Rotate right with carry RotateRC R3, #2

Figure 2.25 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 registers.

Ex: Add R1,R2,R3

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

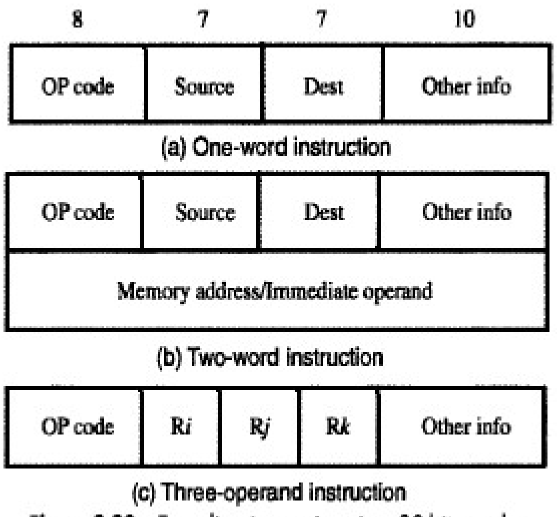


Figure 2.39 Encoding instructions into 32-bit words.

Problem 1:
Write a program that can evaluate the expression A*B+C*D In a single-accumulator processor. Assume that the processor has Load, Store, Multiply, and Add instructions and that all values fit in the accumulator
Solution:
A program for the expression is:

```
Load A
Multiply B
Store RESULT
Load C
Multiply D
Add RESULT
Store RESULT
```

Problem 2:
Registers R1 and R2 of a computer contains the decimal values 1200 and 4600. What is the effective-address of the memory operand in each of the following instructions?
(a) Load 20(R1), R5
(b) Move #3000,R5
(c) Store R5,30(R1,R2)
(d) Add -(R2),R5
(e) Subtract (R1)+,R5
Solution:
(a) EA = [R1]+Offset=1200+20 = 1220
(b) EA = 3000
(c) EA = [R1]+[R2]+Offset = 1200+4600+30=5830
(d) EA = [R2]-1 = 4599
(e) EA = [R1] = 1200

Problem 3:
Registers R1 and R2 of a computer contains the decimal values 2900 and 3300. What is the effective-address of the memory operand in each of the following instructions?
(a) Load R1,55(R2)
(b) Move #2000,R7
(c) Store 95(R1,R2),R5
(d) Add (R1)+,R5
(e) Subtract-(R2),R5
Solution:
a) Load R1,55(R2) This is indexed addressing mode. So EA = 55+R2=55+3300=3355.
b) Move #2000,R7 This is an immediate addressing mode. So, EA = 2000
c) Store 95(R1,R2),R5 This is a variation of indexed addressing mode, in which contents of 2 registers are added with the offset or index to generate EA. So, 95+R1+R2=95+2900+3300=6255.
d) Add (R1)+,R5 This is Autoincrement mode. Contents of R1 are the EA so, 2900 is the EA.
e) Subtract -(R2),R5 This is Auto decrement mode. Here, R2 is subtracted by 4 bytes (assuming 32-bt processor) to generate the EA, so, EA= 3300-4=3296.

Problem 4:
Given a binary pattern in some memory-location, is it possible to tell whether this pattern represents a machine instruction or a number?
Solution:
No; any binary pattern can be interpreted as a number or as an instruction.

Problem 5:

Both of the following statements cause the value 300 to be stored in location 1000, but at different times.

```
ORIGIN 1000
DATAWORD 300
```

And

```
Move #300,1000
```

Explain the difference.

Solution:

The assembler directives ORIGIN and DATAWORD cause the object program memory image constructed by the assembler to indicate that 300 is to be placed at memory word location 1000 at the time the program is loaded into memory prior to execution.

The Move instruction places 300 into memory word location 1000 when the instruction is executed as part of a program.

Problem 6:

Register R5 is used in a program to point to the top of a stack. Write a sequence of instructions using the Index, Autoincrement, and Autodecrement addressing modes to perform each of the following tasks:

(a) Pop the top two items off the stack, and them, and then push the result onto the stack.
(b) Copy the fifth item from the top into register R3.
(c) Remove the top ten items from the stack.

Solution:

```
(a)    Move (R5)+,R0
       Add (R5)+,R0
       Move R0,-(R5)
(b)    Move 16(R5),R3
(c)    Add #40,R5
```

Problem 7:

Consider the following possibilities for saving the return address of a subroutine:

(a) In the processor register.
(b) In a memory-location associated with the call, so that a different location is used when the subroutine is called from different places
(c) On a stack.

Which of these possibilities supports subroutine nesting and which supports subroutine recursion(that is, a subroutine that calls itself)?

Solution:

(a) Neither nesting nor recursion is supported.

(b) Nesting is supported, because different Call instructions will save the return address at different memory-locations. Recursion is not supported.

(c) Both nesting and recursion are supported.