COMPUTER ORGANIZATION AND ARCHITECTURE

Textbooks

- "Computer Organization," by Carl Hamacher, Zvonko Vranesic and Safwat Zaky. Fifth Edition McGraw-Hill, 2002.

BASIC STRUCTURE OF COMPUTERS

TOPICS COVERED

- INTRODUCTION
- FUNCTIONAL UNITS
- BASIC OPERATIONAL CONCEPTS
- BUS STRUCTURES
- SOFTWARE
- MEMORY OPERATIONS
- INSTRUCTIONS
- ADDRESSING MODES
- ARM

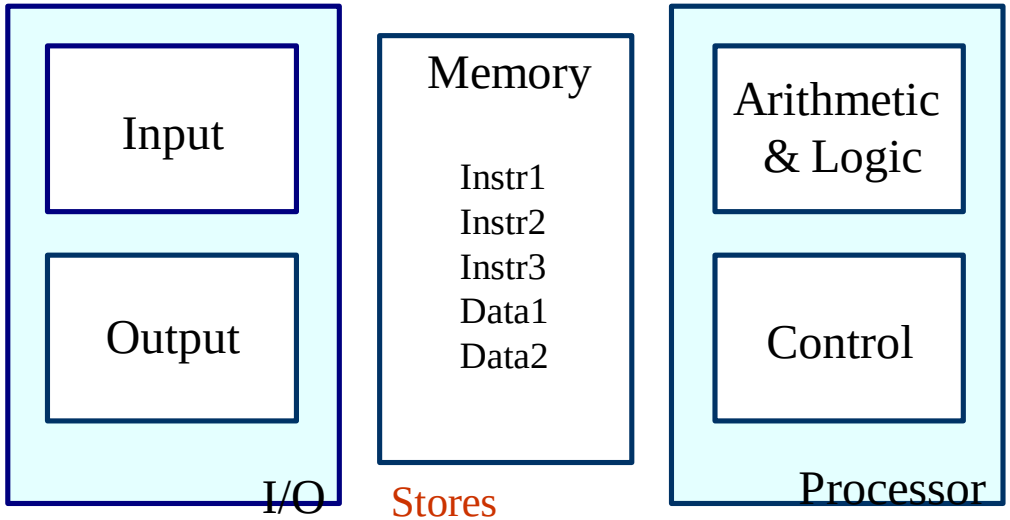
What is a computer?

- a computer is a sophisticated electronic calculating machine that:
  - Accepts input information,
  - Processes the information according to a list of internally stored instructions and
  - Produces the resulting output information.
- Functions performed by a computer are:
  - Accepting information to be processed as input.
  - Storing a list of instructions to process the information.
  - Processing the information according to the list of instructions.
  - Providing the results of the processing as output.
- What are the functional units of a computer?

Functional units of a computer

Input unit accepts information:
- Human operators,
- Electromechanical devices (keyboard)
- Other computers

Arithmetic and logic unit(ALU):
- Performs the desired operations on the input information as determined by instructions in the memory



Output unit sends results of processing:
- To a monitor display,
- To a printer

Stores information:
- Instructions,
- Data

Control unit coordinates various actions
- Input,
- Output
- Processing

Information in a computer -- ***Instructions***

- Instructions specify commands to:
  - Transfer information within a computer (e.g., from memory to ALU)
  - Transfer of information between the computer and I/O devices (e.g., from keyboard to computer, or computer to printer)
  - Perform arithmetic and logic operations (e.g., Add two numbers, Perform a logical AND).
- A sequence of instructions to perform a task is called a program, which is stored in the memory.
- Processor fetches instructions that make up a program from the memory and performs the operations stated in those instructions.

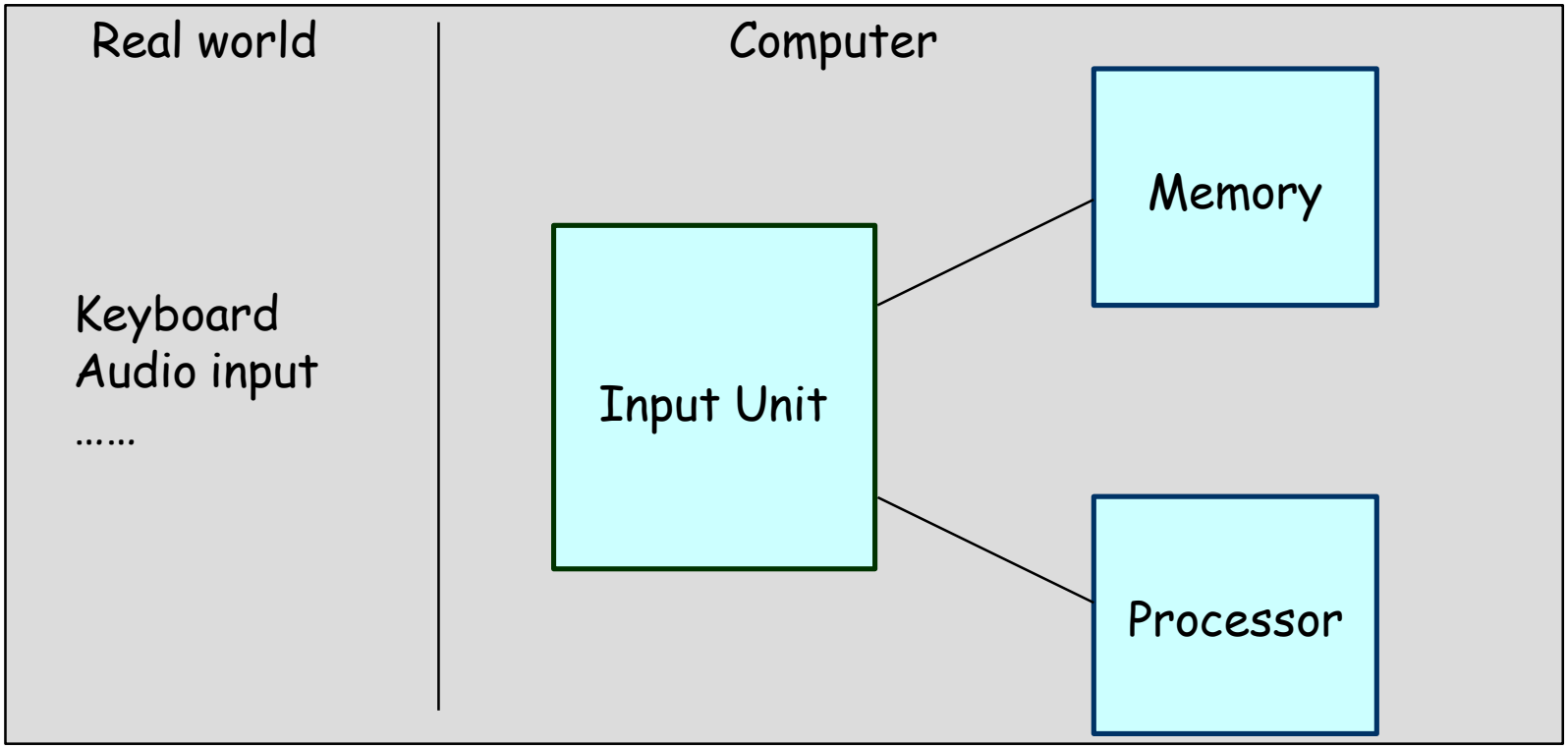
Information in a computer -- **Data**

- Data are the "operands" upon which instructions operate.
- Data could be:
  - Numbers,
  - Encoded characters.
- Data, in a broad sense means any digital information.
- Computers use data that is encoded as a string of binary digits called bits.

Input unit

Binary information must be presented to a computer in a specific format. This task is performed by the input unit:

- Interfaces with input devices.
- Accepts binary information from the input devices.
- Presents this binary information in a format expected by the computer.
- Transfers this information to the memory or processor.



Memory unit

- Memory unit stores instructions and data.
  - Recall, data is represented as a series of bits.
  - To store data, memory unit thus stores bits.
- Processor reads instructions and reads/writes data from/to the memory during the execution of a program.
  - In theory, instructions and data could be fetched one bit at a time.
  - In practice, a group of bits is fetched at a time.
  - Group of bits stored or retrieved at a time is termed as "word"
  - Number of bits in a word is termed as the "word length" of a computer.
- In order to read/write to and from memory, a processor should know where to look:
  - "Address" is associated with each word location.

Memory unit (contd..)

- Processor reads/writes to/from memory based on the memory address:
  - Access any word location in a short and fixed amount of time based on the address.
  - Random Access Memory (RAM) provides fixed access time independent of the location of the word.
  - Access time is known as "Memory Access Time".
- Memory and processor have to "communicate" with each other in order to read/write information.
  - In order to reduce "communication time", a small amount of RAM (known as Cache) is tightly coupled with the processor.
- Modern computers have three to four levels of RAM units with different speeds and sizes:
  - Fastest, smallest known as Cache
  - Slowest, largest known as Main memory.

Memory unit (contd..)

- Primary storage of the computer consists of RAM units.
  - Fastest, smallest unit is Cache.
  - Slowest, largest unit is Main Memory.
- Primary storage is insufficient to store large amounts of data and programs.
  - Primary storage can be added, but it is expensive.
- Store large amounts of data on secondary storage devices:
  - Magnetic disks and tapes,
  - Optical disks (CD-ROMS).
  - Access to the data stored in secondary storage in slower, but take advantage of the fact that some information may be accessed infrequently.
- Cost of a memory unit depends on its access time, lesser access time implies higher cost.

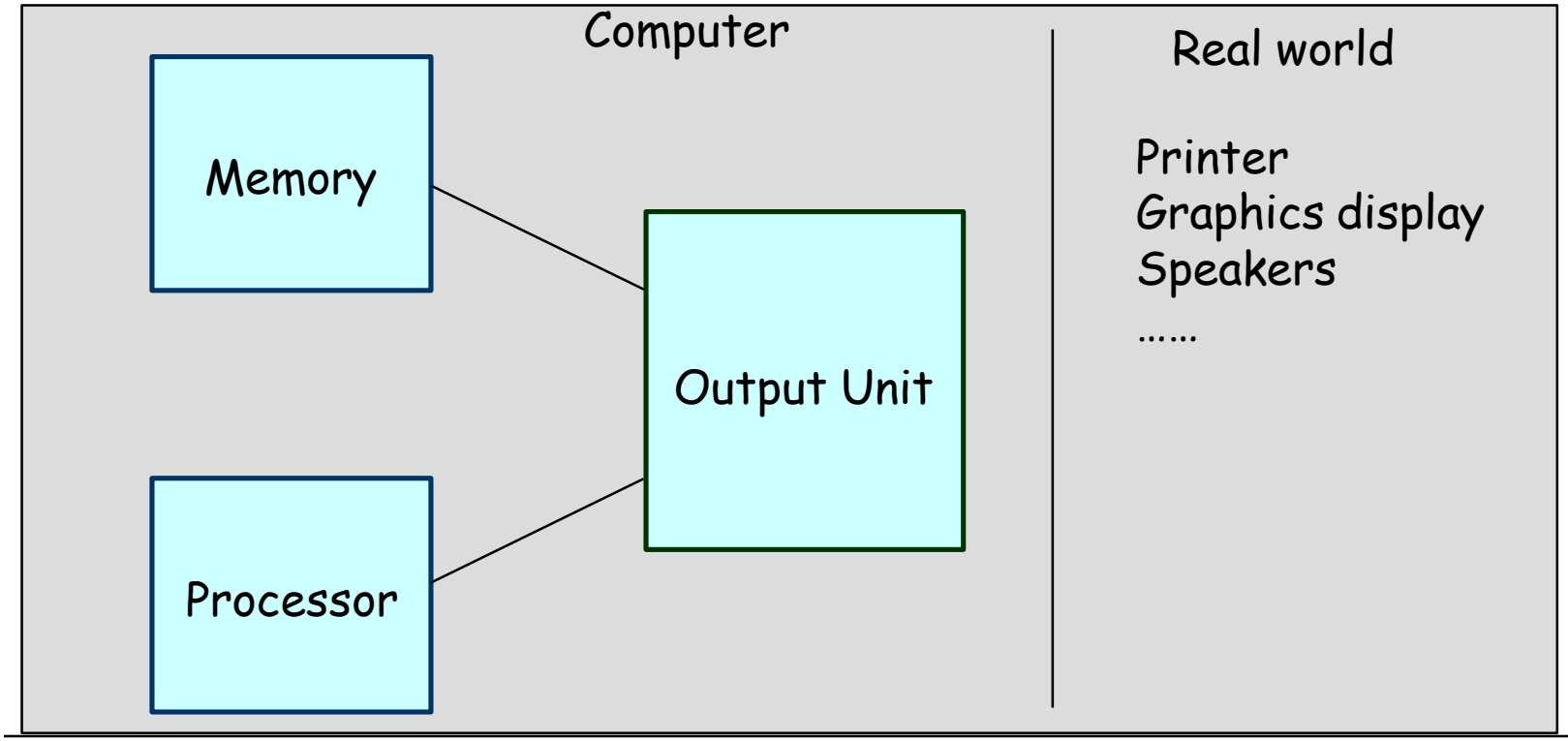
Arithmetic and logic unit (ALU)

- Operations are executed in the Arithmetic and Logic Unit (ALU).
  - Arithmetic operations such as addition, subtraction.
  - Logic operations such as comparison of numbers.
- In order to execute an instruction, operands need to be brought into the ALU from the memory.
  - Operands are stored in general purpose registers available in the ALU.
  - Access times of general purpose registers are faster than the cache.
- Results of the operations are stored back in the memory or retained in the processor for immediate use.

Output unit

•Computers represent information in a specific binary form. Output units:

- Interface with output devices.
- Accept processed results provided by the computer in specific binary form.
- Convert the information in binary form to a form understood by an output device.

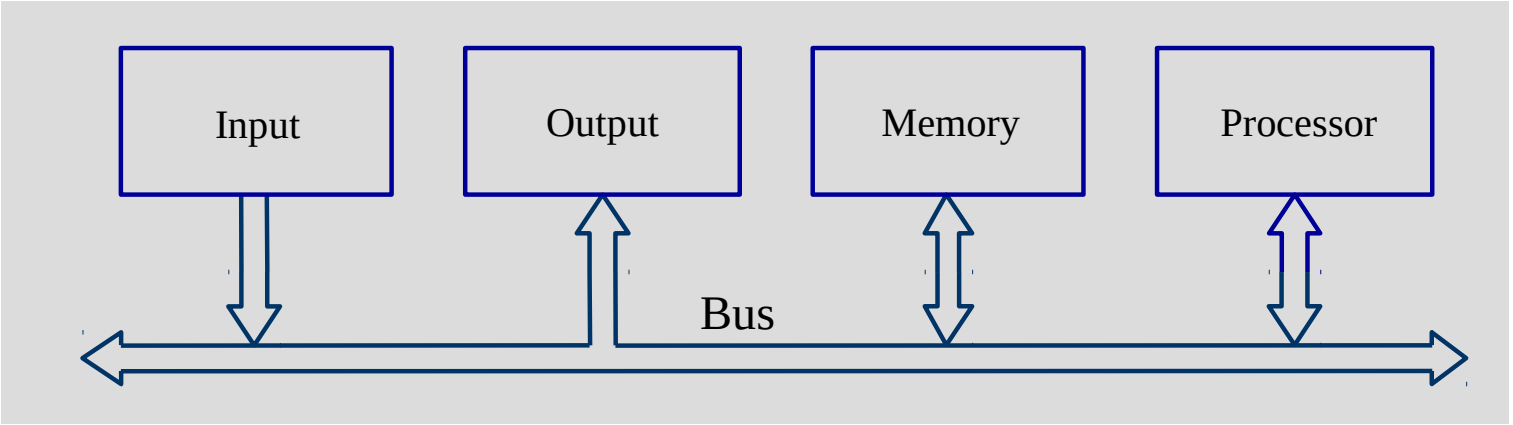


Control unit

- Operation of a computer can be summarized as:
  - Accepts information from the input units (Input unit).
  - Stores the information (Memory).
  - Processes the information (ALU).
  - Provides processed results through the output units (Output unit).
- Operations of Input unit, Memory, ALU and Output unit are coordinated by Control unit.
- Instructions control "what" operations take place (e.g. data transfer, processing).
- Control unit generates timing signals which determines "when" a particular operation takes place.

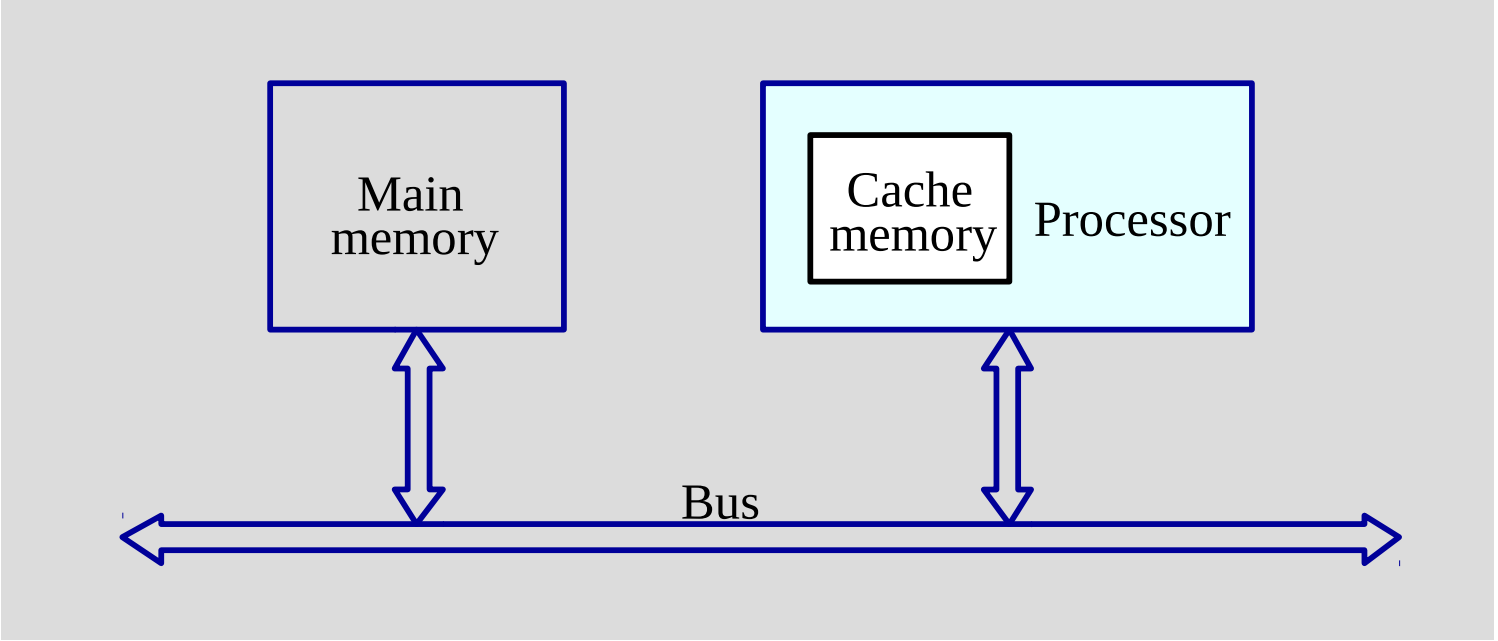
How are the functional units connected?

•For a computer to achieve its operation, the functional units need to communicate with each other.
•In order to communicate, they need to be connected.



•Functional units may be connected by a group of parallel wires.
•The group of parallel wires is called a ***bus***.
•Each wire in a bus can transfer one bit of information.
•The number of parallel wires in a bus is equal to the word length of a computer

Organization of cache and main memory



Why is the access time of the cache memory lesser than the access time of the main memory?

BASIC OPERATIONAL CONCEPTS OF COMPUTER

•To perform a given task an appropriate program consisting of a list of instructions is stored in the memory

•Examples: - Add LOCA, R0
•This instruction adds the operand at memory location LOCA, to operand in register R0 & places the sum into register.
• This instruction requires the performance of several steps,
• 1. First the instruction is fetched from the memory into the processor.
•2. The operand at LOCA is fetched and added to the contents of R0
•3. Finally the resulting sum is stored in the register R0

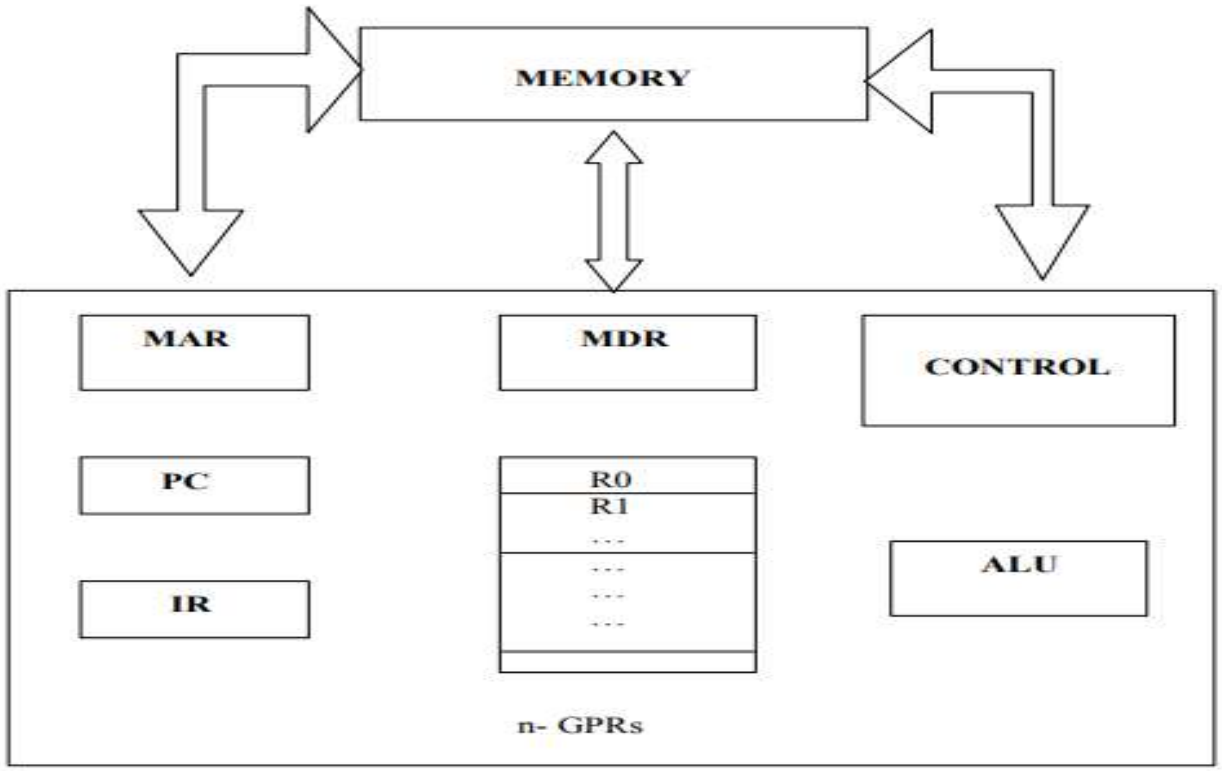


Fig b : Connections between the processor and the memory

The fig shows how memory & the processor can be connected. In addition to the ALU & the control circuitry, the processor contains a number of registers used for several different purposes.

The instruction register (IR):- Holds the instructions that is currently being executed. Its output is available for the control circuits which generates the timing signals that control the various processing elements in one execution of instruction. The program counter

PC:- This is another specialized register that keeps track of execution of a program. It contains the memory address of the next instruction to be fetched and executed.

Besides IR and PC, there are n-general purpose registers R0 through Rn-1.

registers

The other two registers which facilitate communication with memory are:
- 1. MAR – (Memory Address Register):- It holds the address of the location to be accessed.
2 MDR – (Memory Data Register):- It contains the data to be written into or read out of the address location.

Operating steps are

1. Programs reside in the memory & usually get these through the I/P unit.
2. Execution of the program starts when the PC is set to point at the first instruction of the program.
3. Contents of PC are transferred to MAR and a Read Control Signal is sent to the memory.
4. After the time required to access the memory elapses, the address word is read out of the memory and loaded into the MDR.
5. Now contents of MDR are transferred to the IR & now the instruction is ready to be decoded and executed.
6. If the instruction involves an operation by the ALU, it is necessary to obtain the required operands.
7. An operand in the memory is fetched by sending its address to MAR & Initiating a read cycle.

. 8. When the operand has been read from the memory to the MDR, it is transferred from MDR to the ALU.
9. After one or two such repeated cycles, the ALU can perform the desired operation.
10. If the result of this operation is to be stored in the memory, the result is sent to MDR.
11. Address of location where the result is stored is sent to MAR & a write cycle is initiated.
12. The contents of PC are incremented so that PC points to the next instruction that is to be executed.

Interrupt

- Normal execution of programs may be interrupted if some device requires urgent servicing
  - To deal with the situation immediately, the normal execution of the current program must be interrupted

- Procedure of interrupt operation
  - The device raises an interrupt signal
  - The processor provides the requested service by executing an appropriate interrupt-service routine
  - The state of the processor is first saved before servicing the interrupt
    - Normally, the contents of the PC, the general registers, and some control information are stored in memory
  - When the interrupt-service routine is completed, the state of the processor is restored so that the interrupted program may continue

Classes of Interrupts

- Program
  - Generated by some condition that occurs as a result of an instruction execution such as arithmetic overflow, division by zero, attempt to execute an illegal machine instruction, or reference outside a user's allowed memory space
- Timer
  - Generated by a timer within the processor. This allows the operating system to perform certain functions on a regular basis
- I/O
  - Generated by an I/O controller, to signal normal completion of an operation or to signal a variety of error conditions
- Hardware failure
  - Generated by a failure such as power failure or memory parity error

Bus Structures

- A group of lines that serves a connecting path for several devices is called a bus
  - In addition to the lines that carry the data, the bus must have lines for address and control purposes

The connections can be made in several ways using a Bus.

- **Data Bus: it is used for transmission of data. The number of data lines corresponds** to the number of bits in a word.
- **Address Bus: it carries the address of the main memory location from where** the data can be accessed

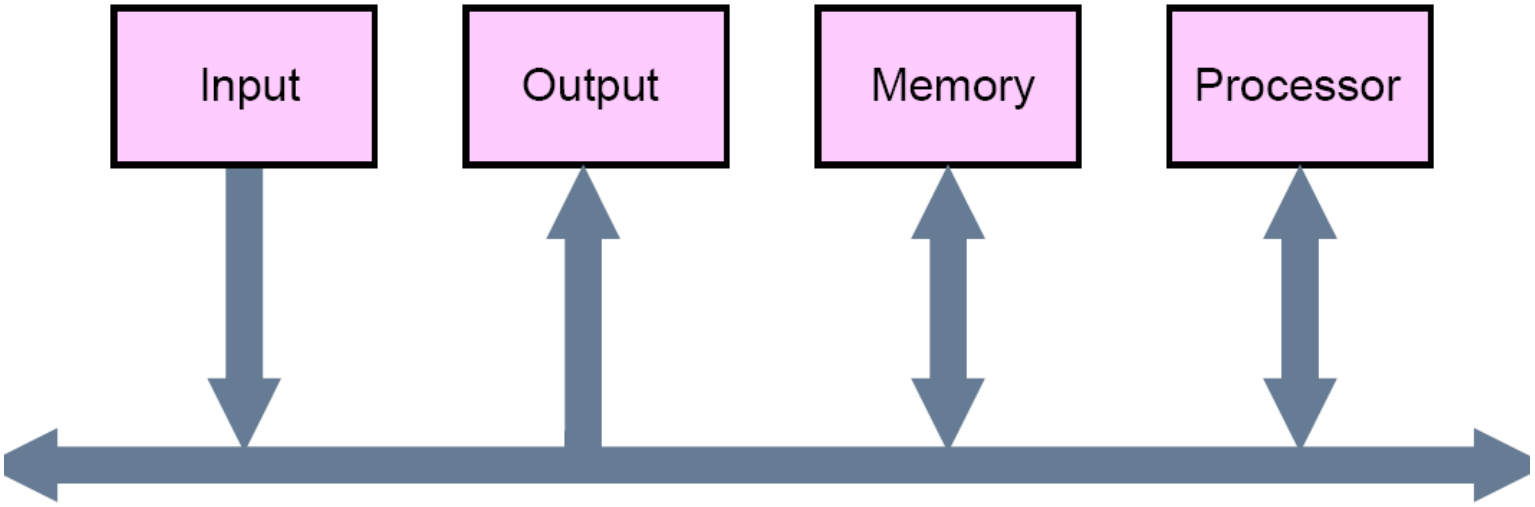
- **Control Bus:** it is used to indicate the direction of data transfer and to coordinate the timing of events during the transfer.

The different functional units of a computer can be connected through a bus structure such as:

- A Single-bus structure
- A Two-bus structure

SINGLE-BUS STRUCTURE:

- The simplest way to interconnect functional units is to use a single bus, as shown below



Drawbacks of the Single Bus Structure

- The devices connected to a bus vary widely in their speed of operation
  - Some devices are relatively slow, such as printer and keyboard
  - Some devices are considerably fast, such as optical disks
  - Memory and processor units operate are the fastest parts of a computer
- Efficient transfer mechanism thus is needed to cope with this problem
  - A common approach is to include buffer registers with the devices to hold the information during transfers
  - An another approach is to use two-bus structure and an additional transfer mechanism
    - A high-performance bus, a low-performance, and a bridge for transferring the data between the two buses.

TWO-BUS STRUCTURE:

- The processor interacts with the memory through a memory bus and handles input/output functions over I/O bus.
- The I/O transfers are always under the direct control of the processor, which initiates transfer and monitors their progress until completion.
- The main advantage of this structure is good operating speed but on account of more cost.

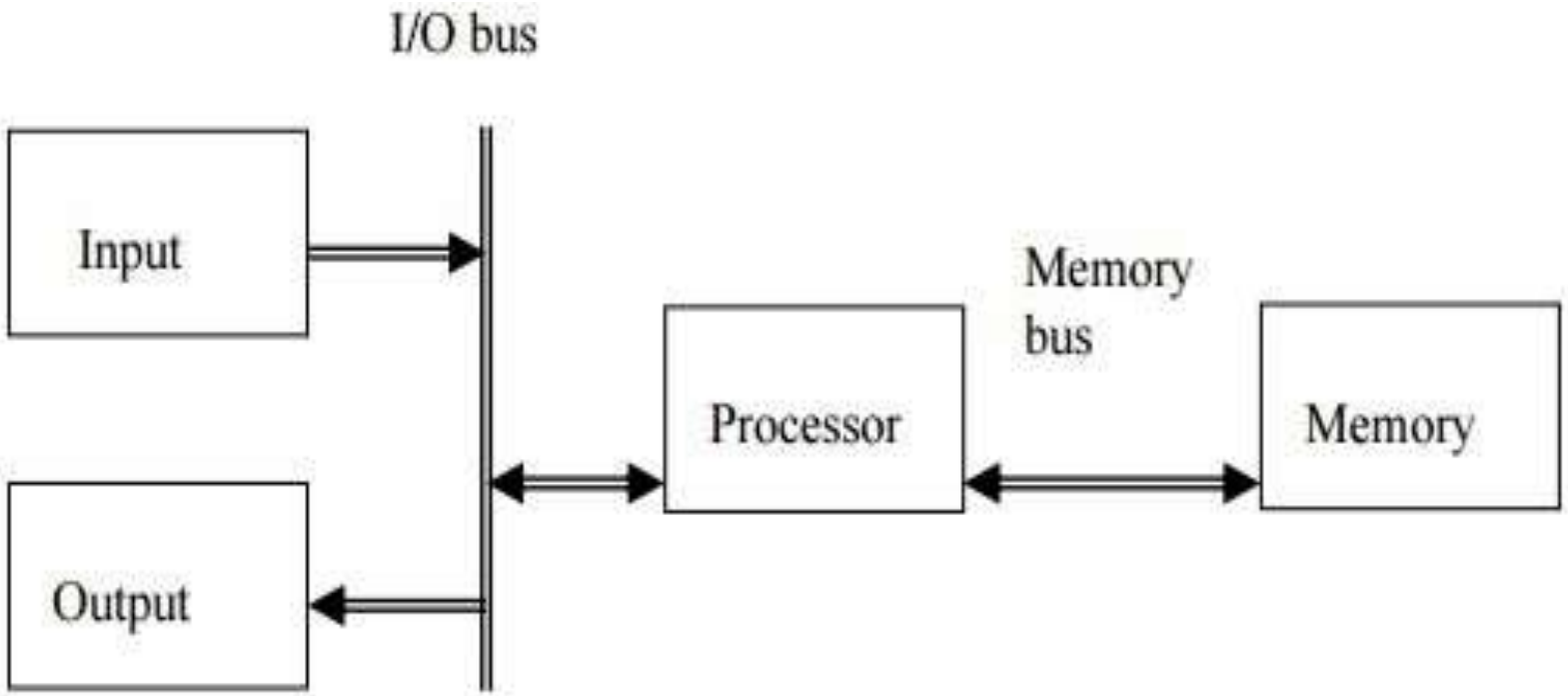


Figure 2.2 Two-bus Structure

Software

- In order for a user to enter and run an application program, the computer must already contain some system software in its memory

- System software is a collection of programs that are executed as needed to perform functions such as
  - Receiving and interpreting user commands
  - Running standard application programs such as word processors, etc, or games
  - Managing the storage and retrieval of files in secondary storage devices
  - Controlling I/O units to receive input information and produce output results

Software

- Translating programs from source form prepared by the user into object form consisting of machine instructions
- Linking and running user-written application programs with existing standard library routines, such as numerical computation packages
- System software is thus responsible for the coordination of all activities in a computing system

Operating System

Operating system (OS)

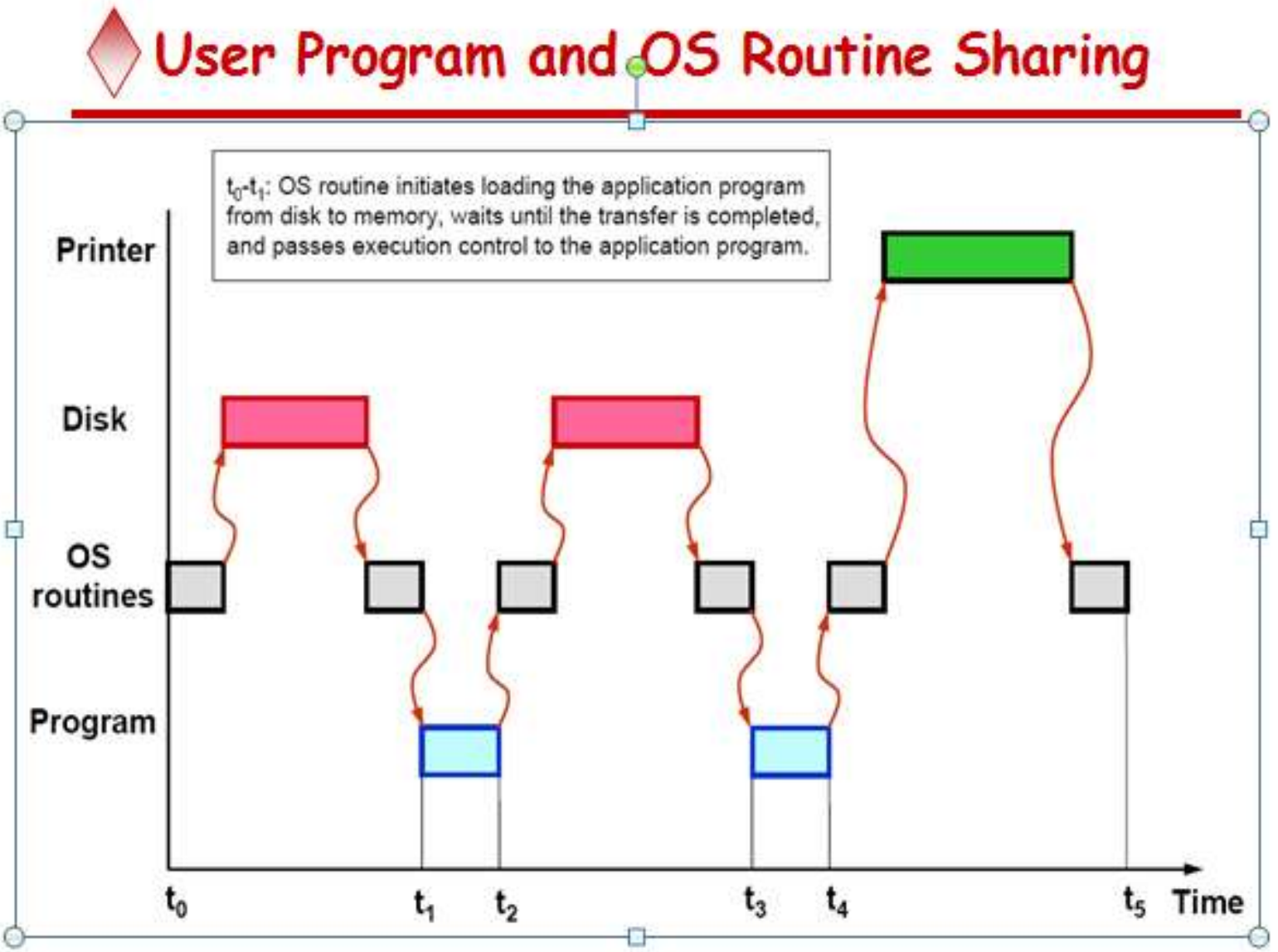
This is a large program, or actually a collection of routines, that is used to control the sharing of and interaction among various computer units as they perform application programs

The OS routines perform the tasks required to assign computer resource to individual application programs

These tasks include assigning memory and magnetic disk space to program and data files, moving data between memory and disk units, and handling I/O operations

In the following, a system with one processor, one disk, and one printer is given to explain the basics of OS

Assume that part of the program's task involves reading a data file from the disk into the memory, performing some computation on the data, and printing the results


User Program and OS Routine Sharing
t_0-t_1: OS routine initiates loading the application program from disk to memory, waits until the transfer is completed, and passes execution control to the application program.
Printer
Disk
OS routines
Program
t_0
t_1
t_2
t_3
t_4
t_5
Time

MEMORY LOCATIONS AND ADDRESSES

- The memory consists of many millions of storage cells, each of which can store a bit of information having the value 0 or 1
- the bits are normally not handled individually.
- deal with them in groups of fixed size.
- the memory is organized so that a group of m bits can be stored or retrieved in a single, basic operation.
- Each group of m bits is referred d to as a word of information, and m is called the word length

The memory of a computer can be schematically represented as a collection of words

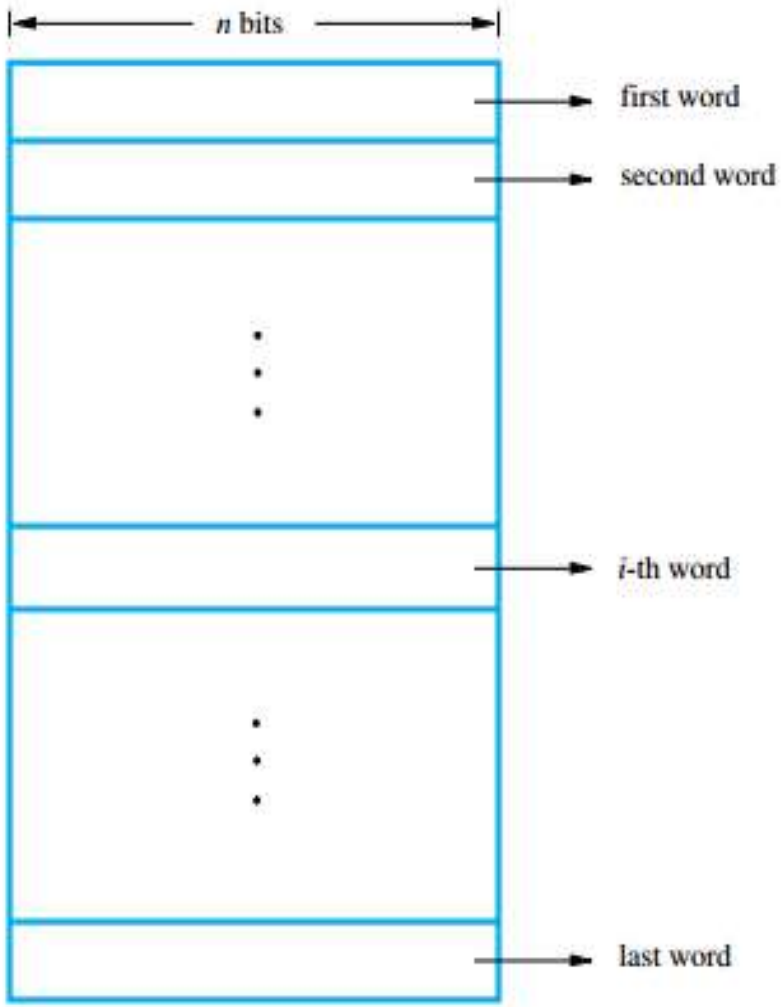
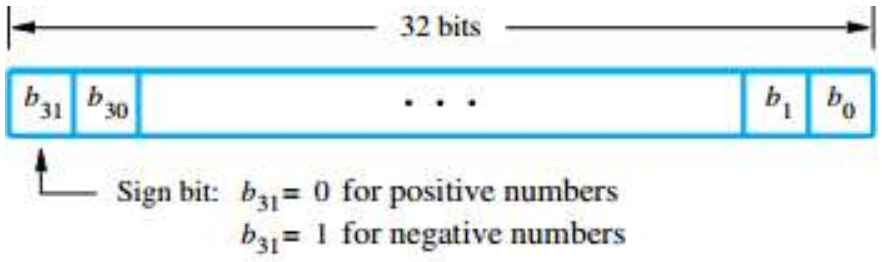


Figure 2.5 Memory words.

- word lengths that typically range from 16 to 64 bits.
- If the word length of a computer is 32 bits,
  - -can store a 32-bit 2's-complement number
  - -four ASCII characters, each occupying 8 bits



(a) A signed integer

(b) Four characters

- Accessing the memory to store or retrieve a single item of information, either a word or a byte, requires distinct names or addresses for each item location.
- to use numbers from 0 through 2^k-1, for some suitable value of k.
- The 2^k addresses constitute the address space of the computer.
- Eg: a 24-bit address generates an address space of 2^{24} (16,777,216) locations.

This is usually written as 16M (16 mega)

BYTE ADDRESSABILITY:

- three basic information quantities to deal with: the bit, byte and word.
- A byte is always 8 bits, but the word length typically ranges from 16 to 64 bits.
- The most practical assignment is to have successive addresses refer to successive byte.
- Contents of the memory locations can represent either instructions or operands.
- Operands can be either numbers or characters.

Representation of Numbers in main memory:

- a 32 bit pattern to represent a signed integer.

b31 b30 … b1 b0

- Sign bit:

b31 = 0 for positive numbers

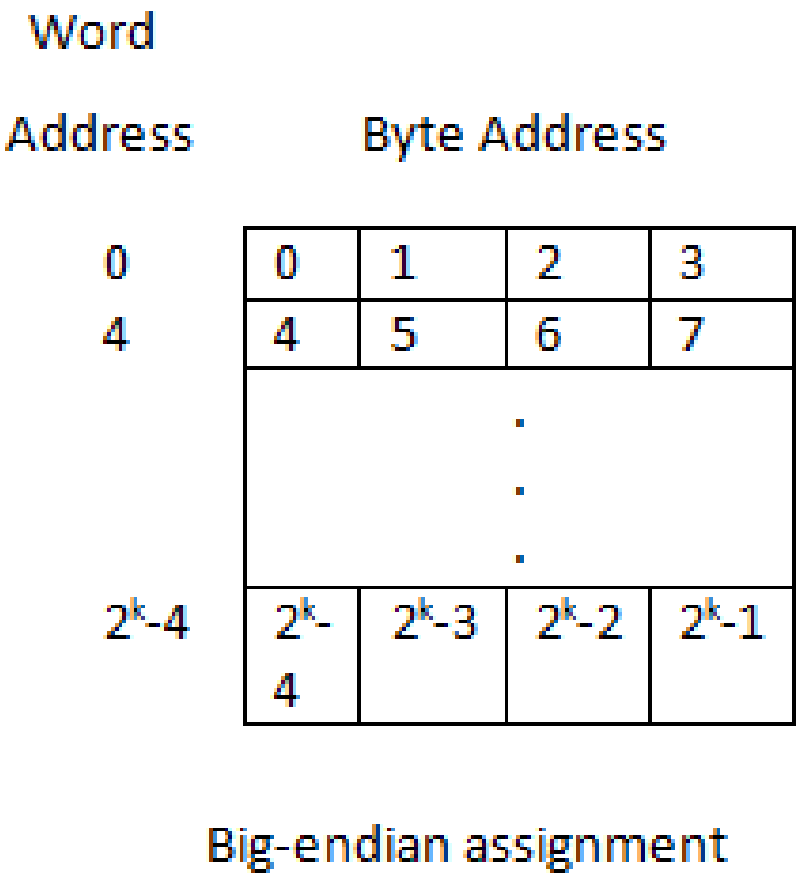
b31 = 1 for negative numbers

Magnitude:= $b_{30} \cdot 2^{30} + b_{29} \cdot 2^{29} + \ldots\ldots + b_1 \cdot 2^1 + b_0 \cdot 2^0$

- Magnitude can range from 0 to 2^{31}-1 and the numbers are said to be in binary positional notation.
- The above encoding format is called Signed Magnitude representation.
- The other two binary representations are 1‘s compliment and 2‘s compliment representations.
- Representation of positive numbers is the same in the three cases.
- The difference is only in the negative number.
- In all the three methods the left most bit is the sign bit (i.e.0 represents positive number and 1 represents negative number).
- 2‘s compliment method is the most suitable one and is used in all modern computers.

BIG- ENDIAN AND LITTLE ENDIAN ASSIGNMENTS

- Little and big-endian are two ways of storing multibyte data-types (int, float, etc).
- In little-endian machines, last byte of binary representation of the multibyte data-type is stored first.
- In big-endian machines, first byte of binary representation of the multibyte data-type is stored last.


Word
Address
Byte Address
0
4
2k-4
0
1
2
3
4
5
6
7
2k-4
2k-3
2k-2
2k-1
Big-endian assignment

Word Address	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

Little-endian assignment

MEMORY OPERATIONS

- To execute an instruction the instructions must be transferred from the main memory to the CPU.
- This is done by the CPU control circuits. Operands and results must also be moved between the main memory and the CPU.
- Thus two basic operations involving the memory are needed namely, **Load (or Fetch or Read) and Store (or Write).**

- **Load operation:** Transfers a copy of the contents of a specific memory location to the CPU.
- The Word in the main memory remains unchanged.
- To start a Load or Fetch operation, CPU sends the address of the desired location to the main memory and requests to read its contents.
- The main memory reads the data stored at that address and sends them to the CPU

- **Store operation:** Transfers a word of information from the CPU to a specific main memory location, destroying the former contents of that location.
- The CPU sends the address of the desired location to the main memory, together with the data to be written to that location

INSTRUCTIONS AND INSTRUCTION SEQUENCING

- A computer must have instructions capable of performing four types of operations:

  1. Data transfers between the main memory and the CPU registers
  2. Arithmetic and logic operations on data
  3. Program sequencing and control
  4. I/O transfers

Notations used:-

a) Register Transfer Notation (RTN):- Possible locations involved in transfer of information are memory location, CPU registers or registers in the I/O subsystem.

- We identify the names for the addresses of memory location as LOC, PLACE, A, VAR2 etc and the names for CPU registers as R0, R5 etc.
- The contents of a location or a register are denoted by placing the corresponding name between square brackets.

- E.g. i) R1 ← [LOC] means that the contents of memory location LOC are transferred into register R1.

  ii) R3 ← [R1] + [R2] adds the contents of registers R1 and R2 and then places their sum into register R3.

b) Assembly Language Notation:- The same operations can be represented in assembly language format as shown below.

- E.g. i) Move LOC, R1

  ii) Add R1,R2,R3

BASIC INSTRUCTION TYPES

- There are five types of instruction formats in a computer that are commonly used, namely:

    1. Three-address instruction format
    2. Two-address instruction format
    3. One-address instruction format
    4. Zero-address instruction format
    5. One-and-half address instruction format

Three-address instruction

- C = A + B is a high level instruction to add the values of the two variables A and B and to assign the sum to a third variable C.
- When this statement is compiled, each of these variables is assigned to a location in the memory.
- The contents of these locations represent the values of the three variables. Hence the above instruction requires the
- Action: C □ [A] + [B]

- To carry out this instruction, the contents of the memory locations A and B are fetched from the main memory and transferred into the processor – sum is computed – result is sent back to memory and stored in location C.
- The same action is performed by a single machine instruction (three address instruction)

Add A,B,C

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

- The general format is

  Operation Source1,Source2, Destination

- If *k bits are needed to specify the memory address of each operand, the encoded form of* the above instruction must contain 3*k bits for addressing purposes + the bits needed to* denote the Add operation

Two-address instruction

- An alternative method is to use two address instruction of the form

  Operation Source, Destination

- E.g. Add A,B which perform the operation

  B □[A] + [B]

- Here the sum is calculated and the result is stored in location B replacing the original contents of this location. i.e. operand B acts as source as well as destination.

- In the former case (three address instruction) the contents of A and B were not destroyed.
- But here the contents of B are destroyed. This problem is solved by using another two-address instruction to copy the contents of one memory location into another location.

C [A] + [B] is equivalent to

Move B,C

Add A,C

One-address instruction

- Instead of mentioning the second operand, it is understood to be in a unique location.
- A processor register usually called the Accumulator is used for this purpose.

E.g.

i)Add A means that the contents of the memory location A is added to the contents of accumulator and places the sum in the accumulator.

ii) Load A copies the contents of memory location A into accumulator

iii) Store A copies the contents of accumulator to the location A

- Depending on the instruction, the operand may be source or destination.
- Now the operation C [A] + [B] can be performed by executing the following instructions

```
Load A
Add B
Store C
```

- The above mentioned instructions can also be handled by using general purpose registers.

  Let R*i represent a general purpose register.*

  Move A,R*i*

  Move R*i*,*A*

  Add A,R*i*

- They are generalizations of Load, Store and Add instructions of the single accumulator case in which register R*i performs the functions of accumulator.*

- When a processor has several general-purpose registers, then many instructions involve only operands that are in registers

  E.g., Add R*i*, *Rj*

  Add R*i*, *Rj*, *Rk*

- In the first instruction, R*j* *acts as both source and destination. In the second instruction*, R*i* And R*j* *are source and Rk is the destination*

- **Advantages of using CPU registers:**

  i)Data access from these registers is faster than that of main memory locations; because these registers are inside the processor.

  ii)Only few bits are needed to specify the register; because the number of registers is very less.

- For example, only 5 bits are needed to specify 32 registers.

  iii) Instructions, where only register names are contained, will normally fit into one word of memory

Zero-address instruction

- Here locations of all operands are defined implicitly. Such instructions are found in machines that store operands in a structure called a *pushdown stack.*
- *A stack is a list of* data elements, usually words or bytes, in which these elements can be added or removed through the top of the stack by following the LIFO (last-in-first-out) storage mechanism.
- A processor register called **stack pointer. (SP) is used to keep track of the address of the** element at the top of the stack at any given time.

- 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.

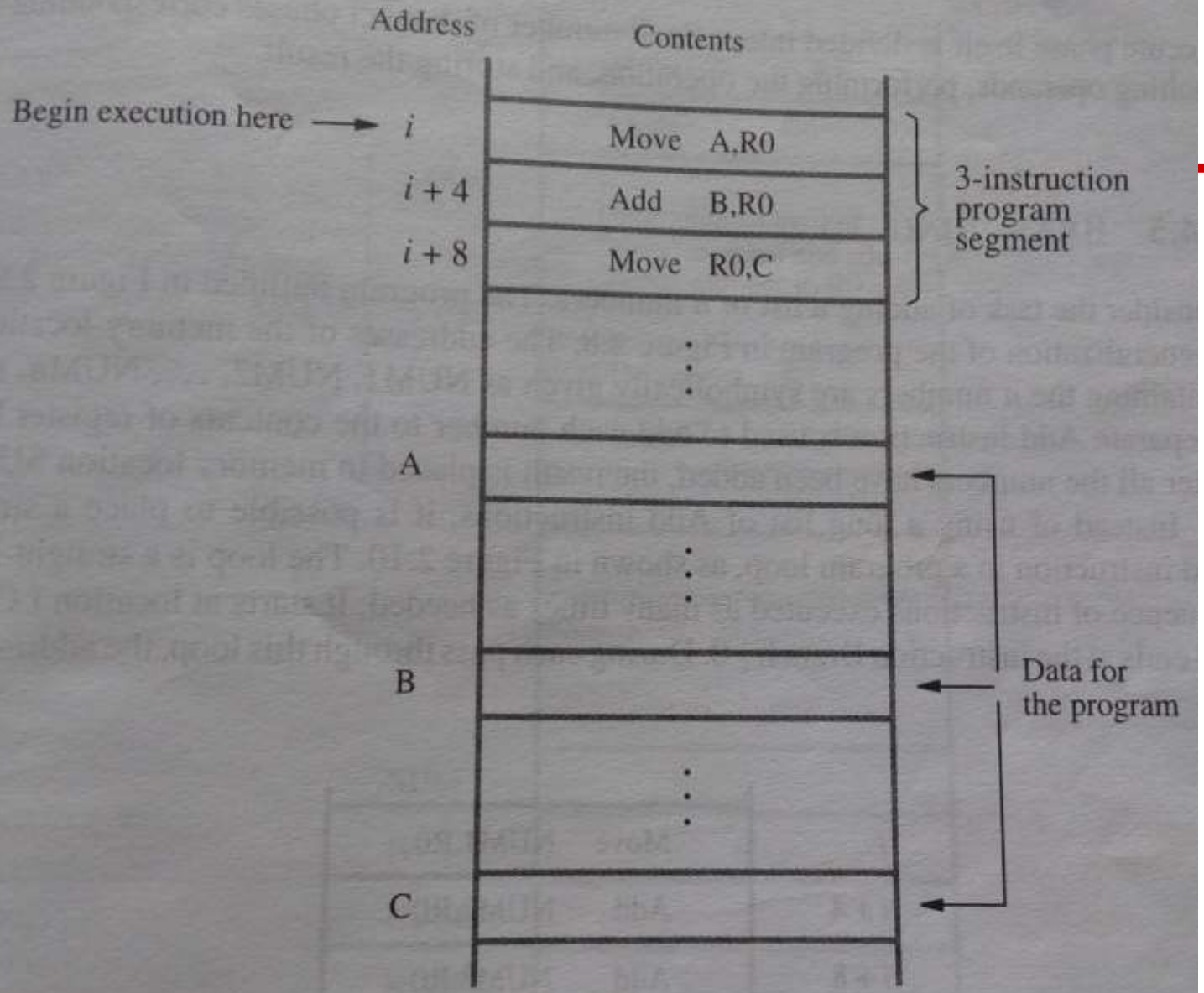
One and half-address instruction

- An instruction that specifies one operand in memory and one operand in a CPU register is referred to as one-and-half address instruction.
- Using registers it is possible to increase the speed of processing

INSTRUCTIONS EXECUTION AND STRAIGHT LINE SEQUENCING

- Let us take the operation C ←[A] + [B]. The Example shows the program segment as it appears in the main memory of a computer that has a two-address instruction format and a number of general purpose CPU registers

Example : A program for C← [A] + [B]



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

- Steps for executing this program,:

1.CPU contains the register called PC which holds the address of the instruction to be executed next.

To begin execution, the address of the first instruction 'i' must be placed in PC.

2.CPU control circuits use the information in the PC to fetch and execute the instructions one at a time in the increasing order of addresses. This is called straight line sequencing.

- *As each instruction is executed, the PC is incremented by 4 to point to the next instruction.*
- Executing a given instruction is a two-phase procedure
- First Phase - Instruction Fetch: Instruction is fetched from the main memory location whose address is in the PC and is placed in the Instruction Register (IR)

- Second Phase – Instruction Execute: **Instruction in the IR is examined to determine** which operation is to be performed.
- The specified operation is performed by the processor.
- This may involve fetching operands from main memory (or processor registers), performing an arithmetic or logic operation and storing the result in the destination location.
- At some point during this two-phase procedure, the contents of the PC are advanced to point to the next instruction.
- After the execution phase is over, new instruction fetch can begin.

BRANCHING

- Consider the task of adding ‘n‘ numbers.
- Let the address of memory locations containing n numbers are NUM1, NUM2,…NUM*n*.
- *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 the memory location SUM.

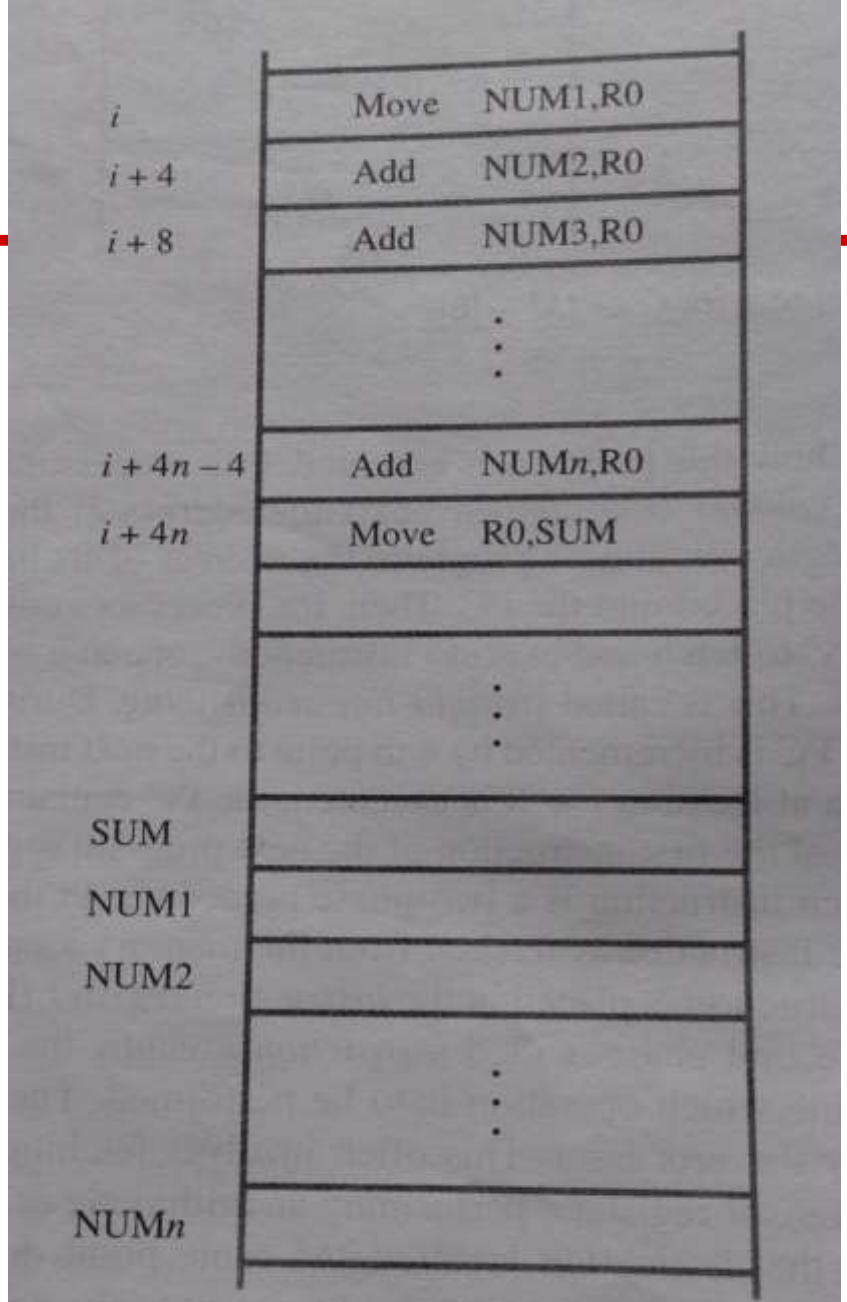


Figure 2.9 A straight-line program for adding *n* numbers.

- Instead of using long list of Add instruction, it is possible to place a single Add instruction in a loop.
- This loop causes a straight line sequence of instructions to be executed repeatedly.
- The loop starts at location LOOP and ends at the instruction Branch>0.
- During each pass through this loop, the address of the next entry is determined and that entry is fetched and added to R0.
- Assume that the number of entries in the list '*n*' *is stored in location N*

- Register R1 is used as a counter to determine the number of time the loop is to be executed.
- Hence the contents of the location N are loaded in register R1 at the beginning of the program.
- Then within the body of the loop the instruction Decrement R1 reduces the contents of R1 by 1 each time through the loop.
- This means that execution of the loop must be repeated as long as the result of the decrement operation is greater than 0.

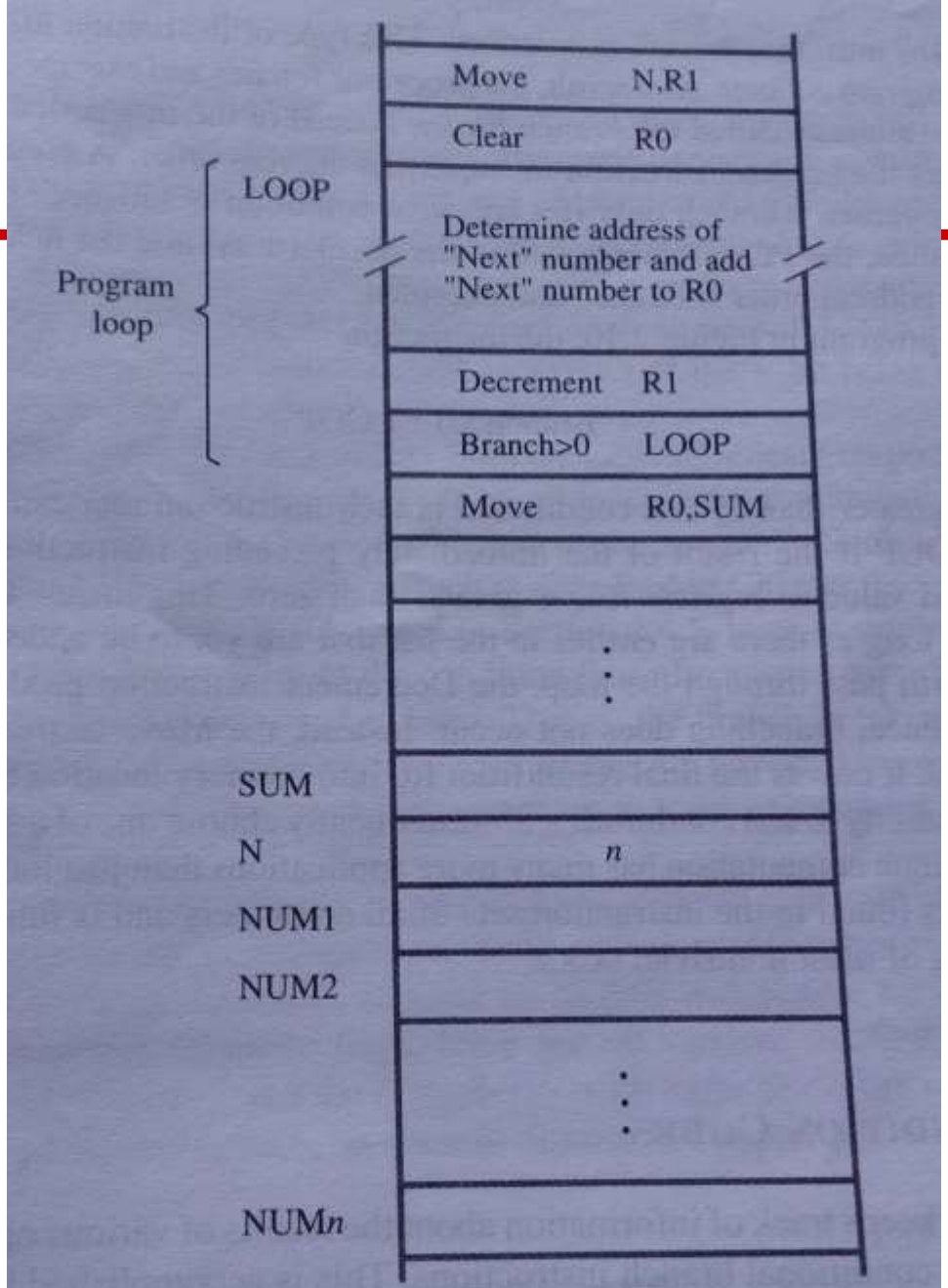


Figure 2.10 Using a loop to add *n* numbers.

- This type of instruction loads a new value into the program counter.
- As a result, the processor fetches and executes the instruction at this new address.
- 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

CONDITION CODES

- The processor keeps track of some information about the results of various operations for use by subsequent conditional branch instructions.
- This is done by recording the required information into individual bits called as ***condition code flags.***
- ***In some processors, these***flags are grouped together in a special register called the ***condition code register or status register.***

- Four commonly used flags are:-

  N (negative) Sets to 1 if the result is negative; otherwise, cleared to 0

  Z (zero) Sets to 1 if the result is 0; otherwise, cleared to 0

  V (overflow) Sets to 1 if arithmetic overflow occurs; otherwise, cleared to 0

  C (carry) Sets to 1 if carry-out results from the operation; otherwise, cleared to 0

ADDRESSING MODES AND ASSEMBLY LANGUAGE

- The term addressing mode refers to the way in which the operand of an instruction is specified.

1. *Register mode:* *The operand is the contents of a CPU register; the name of register is* given in the instruction

E.g. Move R1,R2 The contents of R1 is transferred to R2

2. Absolute mode (Direct mode): The operand is in a memory location.

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

E.g. Add A,B

- The contents of the memory location A is added to the contents of the memory location B.
- The addresses of A and B are given in the instruction itself.

3. Immediate mode: *The operand is given explicitly in the instruction.*

- *This mode is used* in specifying address and data constants in programs

E. g. Move #200, R0

- This instruction places the value 200 in register R0.

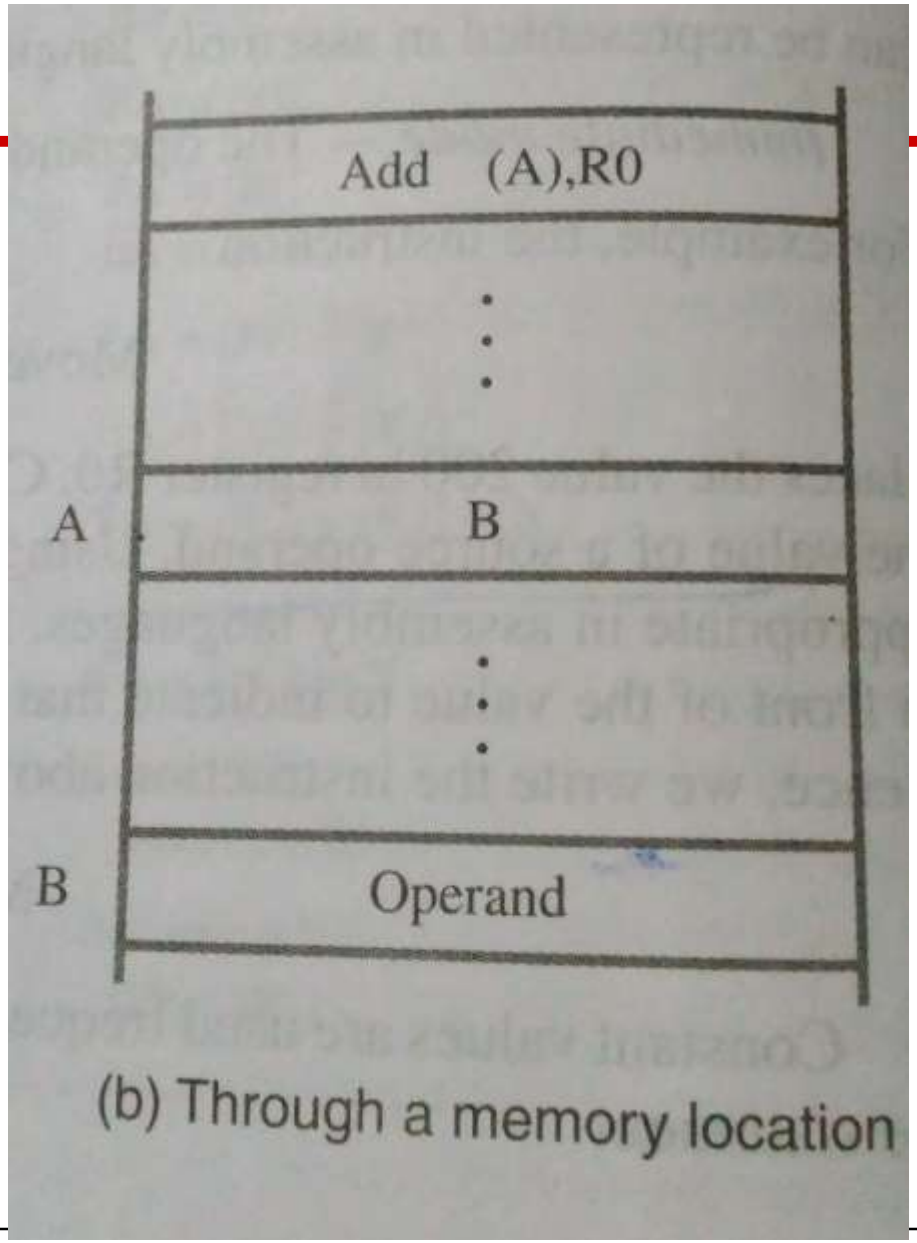
4. Indirect mode: *Here the instruction does not give the operand or its address explicitly.*

- Instead it provides the effective address of the operand. Effective address of the operand is the contents of a register or main memory location, whose address appears in the instruction.
- We denote indirection by placing the name of the register or the memory address given in the instruction in parenthesis
- Let us consider two cases:

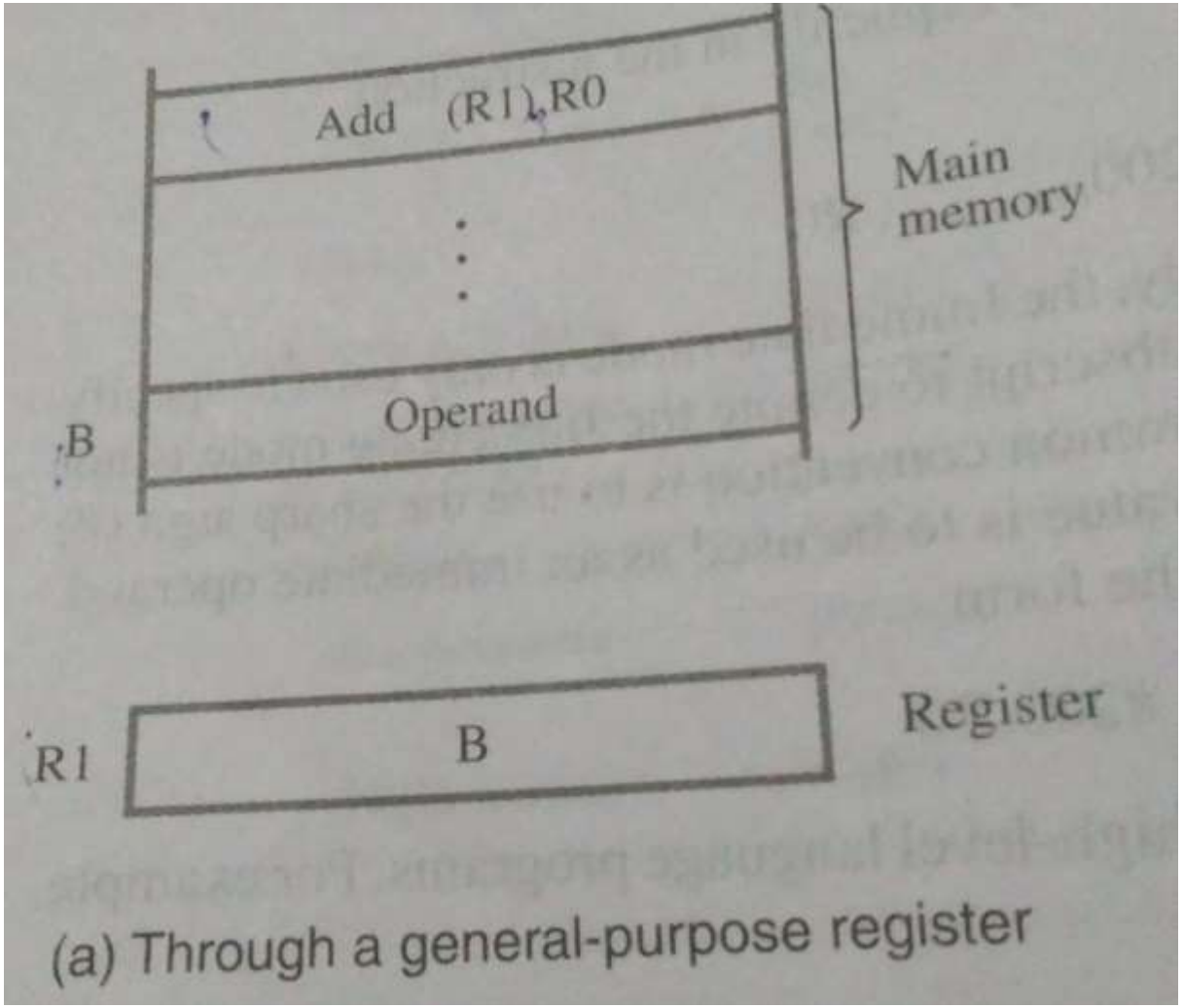
  i) Add (A), R0

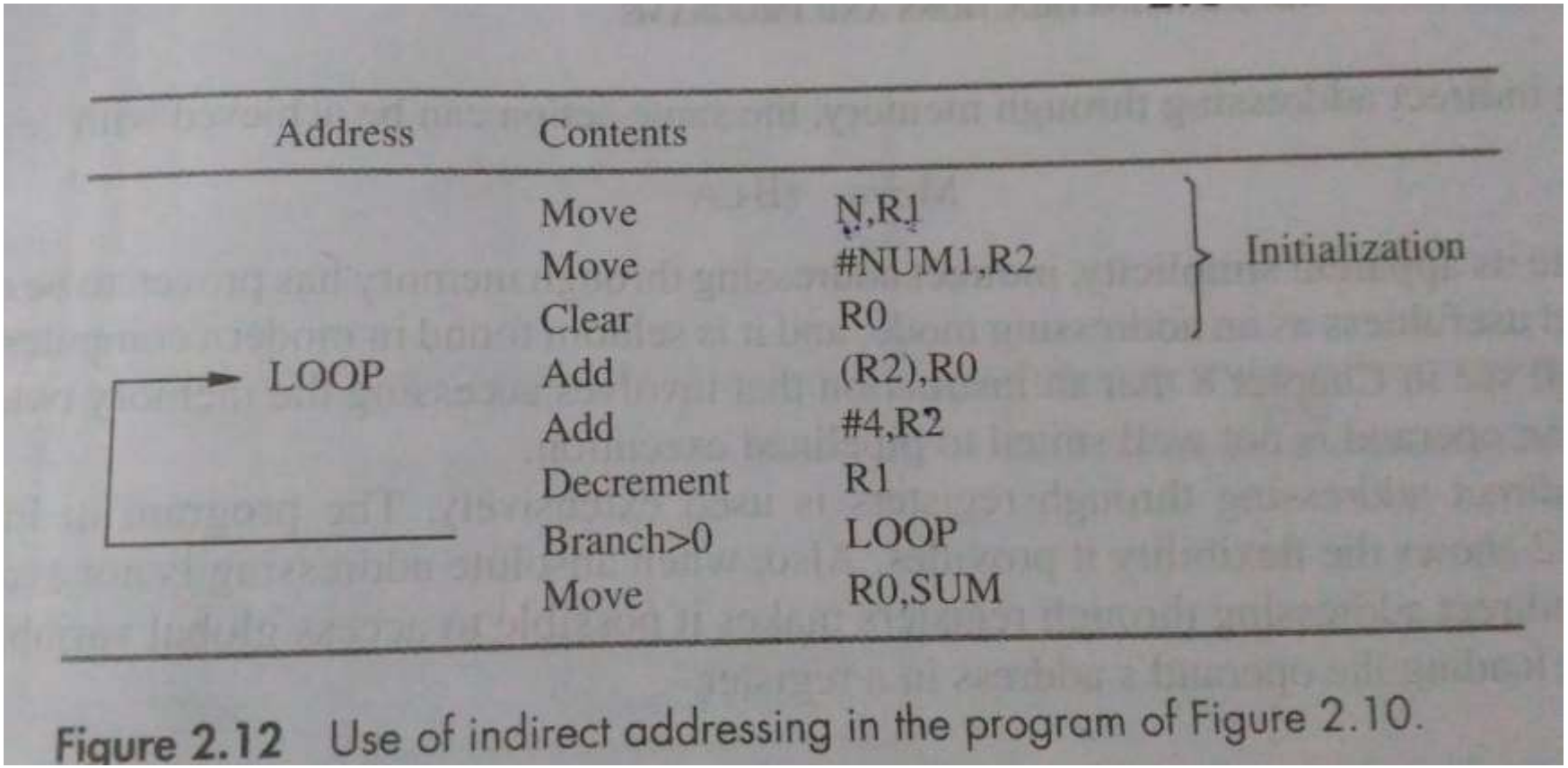
  ii) Add (R1), R0
- In the first case, when the instruction is executed, CPU starts by fetching the contents of location A in the main memory.

- Since indirect addressing is used, the value B stored in A is not the operand, but the address of the operand.
- Hence CPU requests another read operation from the main memory and this is to read the operands (contents of location B).
- The CPU then adds the operand to the contents of R0
- In the second case, the operand is accessed indirectly through register R1 which contains the value B.



(b) Through a memory location


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



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.

- ***5. Index mode:*** ***In this mode, the effective address of the operand is generated by adding*** a constant value to the contents of a register.

- There are two ways of using the index mode.

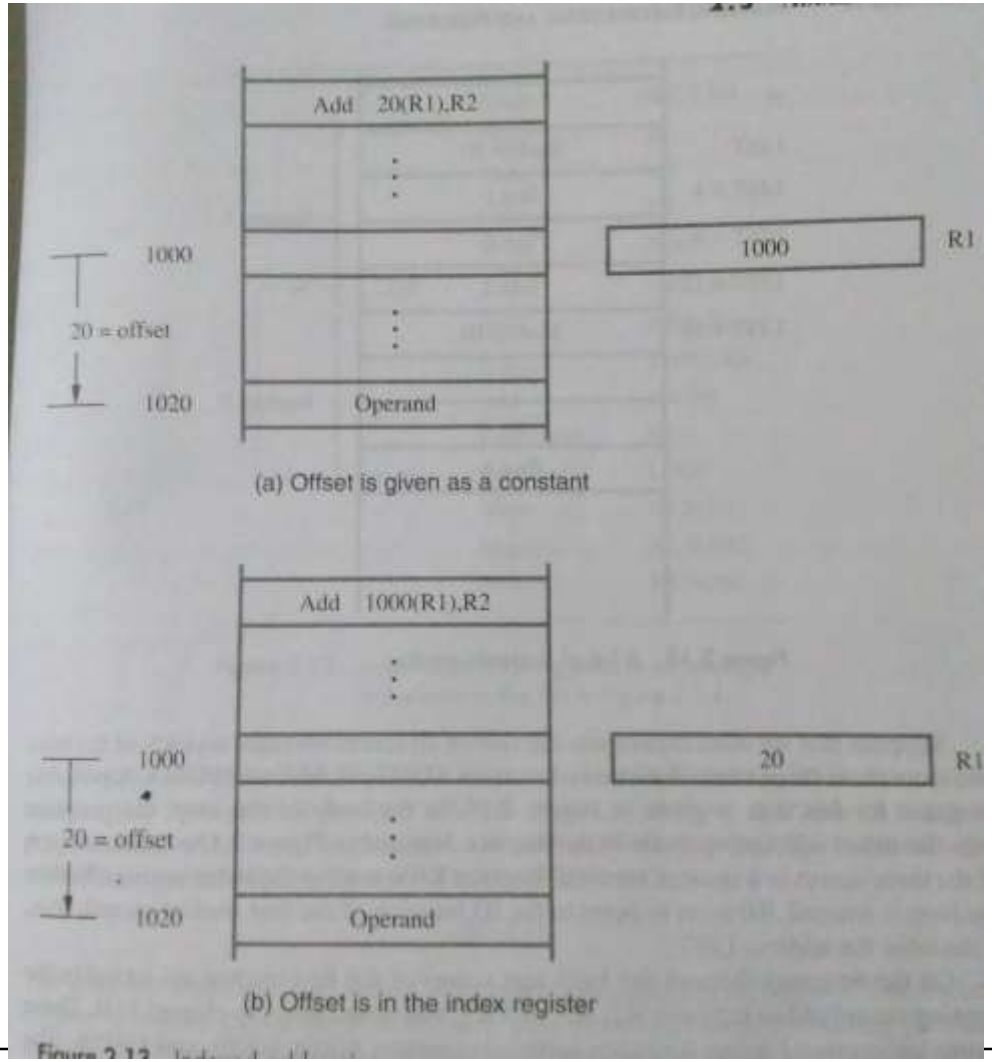
  1.Offset is given as a constant: Here the index register R1 contains the address of a memory location and the value X defines an offset called displacement

  Add 20(R1),R2

  2.Offset is in the index register: Here the constant X corresponds to a memory address and the contents of the index register define the offset to the operand.

  Add 1000(R1),R2

- The register used may be a special register provided for this purpose or may be any one of the general purpose register – referred to as an ***Index Register.***
- Index mode is indicated symbolically as X(R*i*), *where X denotes a constant value* contained in the instruction and R*i is the name of the register involved.*
- *The effective* address of the operand is given by EA or Aeff = X + [R*i]*
- In assembly language program, the constant X may be given either as an explicit number or as a name representing a numerical value.


Add 20(R1),R2
1000
20 = offset
1020
Operand
1000
R1
(a) Offset is given as a constant
Add 1000(R1),R2
1000
20 = offset
1020
Operand
20
R1
(b) Offset is in the index register

- In either case, the effective address is the sum of two values; one is given explicitly in the instruction and the other is in a register.
- ***5. Relative mode:*** *The effective address is determined by the index mode.*
- *But here the* program counter (PC) is used in place of the general purpose register R*i*.
- *i.e. X(PC) can* be used to address a memory location that is X bytes away from the location presently pointed by the program counter.
- Since the addressed location is identified ―relativell to the program counter, which always identifies the current execution point in a program, this mode is called as Relative mode.

- This mode is used to access data operands. It is commonly used to specify the target address in branch instruction.
- ***6. Autoincrement mode:*** *The effective address of the operand is the contents of a* register specified in the instruction.
- After accessing the operand, the contents of the register is automatically incremented to point to the next item in a list.
- We denote the auto increment mode by putting the specified register in parenthesis to show that the contents of register is used as the effective address, followed by a plus (+) sign to indicate that these contents are to be incremented after the operand is accessed.

- Thus the auto increment mode is written as (R*i)* +. *If we use auto increment mode, it is possible to eliminate the increment* instruction
- ***7. Auto decrement mode:*** *The contents of a register specified in the instruction are decremented*.
- These contents are then used as the effective address of the operand. We denote the auto decrement mode by putting the specified register in the parenthesis, preceded by a minus (-) sign to indicate that the contents of the register are to be decremented before being used as the effective address.
- Thus we write − (R4).
- This mode allows accessing of operands in the direction of descending address

BASIC INPUT/OUTPUT OPERATIONS

- Data are transferred between the memory of a computer and the outside world.
- The way they are performed can have significant effect on the performance of a computer.
- Program controlled I/O
  - Simple method used to perform I/O task
- The rate of data transfer from the keyboard to the computer and from the computer to the display is still much slower than the speed of the processor.
- The mechanism used to synchronize the transfer of data between them is:
- On Input
  - Moving a character code from the keyboard to the processor

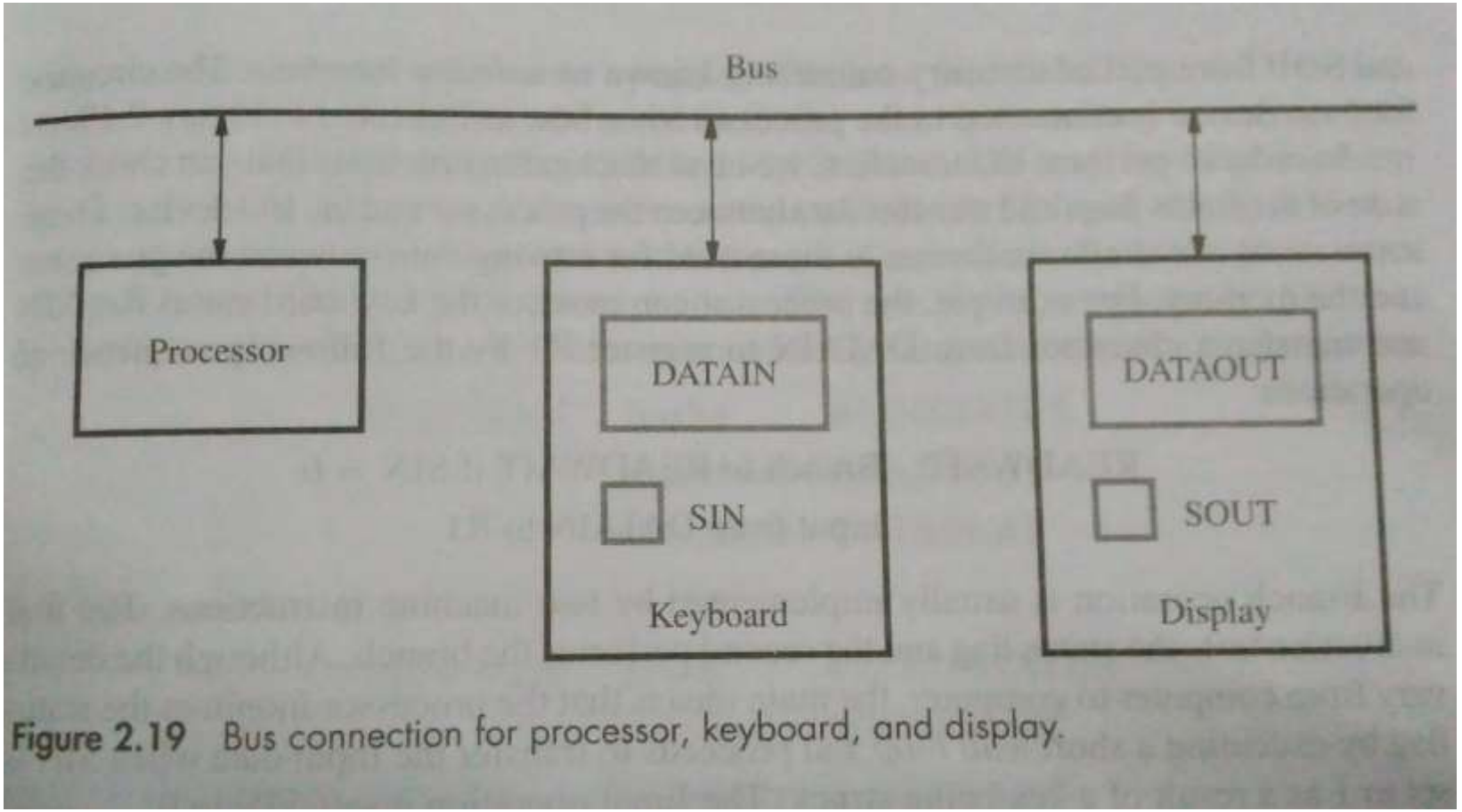


Figure 2.19 Bus connection for processor, keyboard, and display.

- Striking a key stores the corresponding code in an 8-bit buffer register(DATAIN) associated with the keyboard.
- To inform the processor that a valid character is in DATAIN, a status control flag, SIN, Is set to 1.
- A program monitors SIN, and when SIN is set to 1,processor reads the contents of DATAIN.
- When the character is transferred to the processor ,SIN is automatically cleared to 0.
- Second character is entered, SIN is again set to 1. and the process repeats.

- When the characters are transferred from processor to the display, a buffer Register, DATAOUT, and a status control flag, SOUT, are used.
- When SOUT equals 1, the display is ready to receive a character.
- The processor monitors SOUT, and when SOUT is set to 1,the processor transfers a character code to DATAOUT.
- The transfer of a character to DATAOUT clears SOUT to 0.
- When the display device is ready to receive a second character, SOUT is again set to 1

- The buffer registers and status flags are part of device interfaces.
- To perform I/O operations Machine Instructions are needed.
- That check the state of status flags and transfer data between the processor and I/O devices.
- Input to Processor
- Eg:
  - READWAIT Branch to READWAIT if SIN=0

    Input from DATAIN to R1

- Output to the Display
- Eg:
  - WRITEWAIT Branch to WRITEWAIT if SOUT=0

    Output from R1 to DATAOUT

- When I/O device and memory share the same address space, the arrangement is called-memory –mapped I/O
- Some memory 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 these registers.
  - Eg:
    - MoveByte DATAIN,R1

STACKS

- Data structure-Used to control and information linkage between the main program and subroutine
- Is a list of data elements, usually words or bytes.
- The elements can be added or removed at one end of the list only.
- This end is called the top of the stack.
- Other end is called bottom of the stack.
- Structure is also known as pushdown stack.

- Last-in-First-out(LIFO) stack.
  - The last data item placed on the stack is the first one removed when retrieval begins.

- Push and Pop are the terms used for placing a new item on the stack and removing the top item from the stack.

A stack of words in the memory

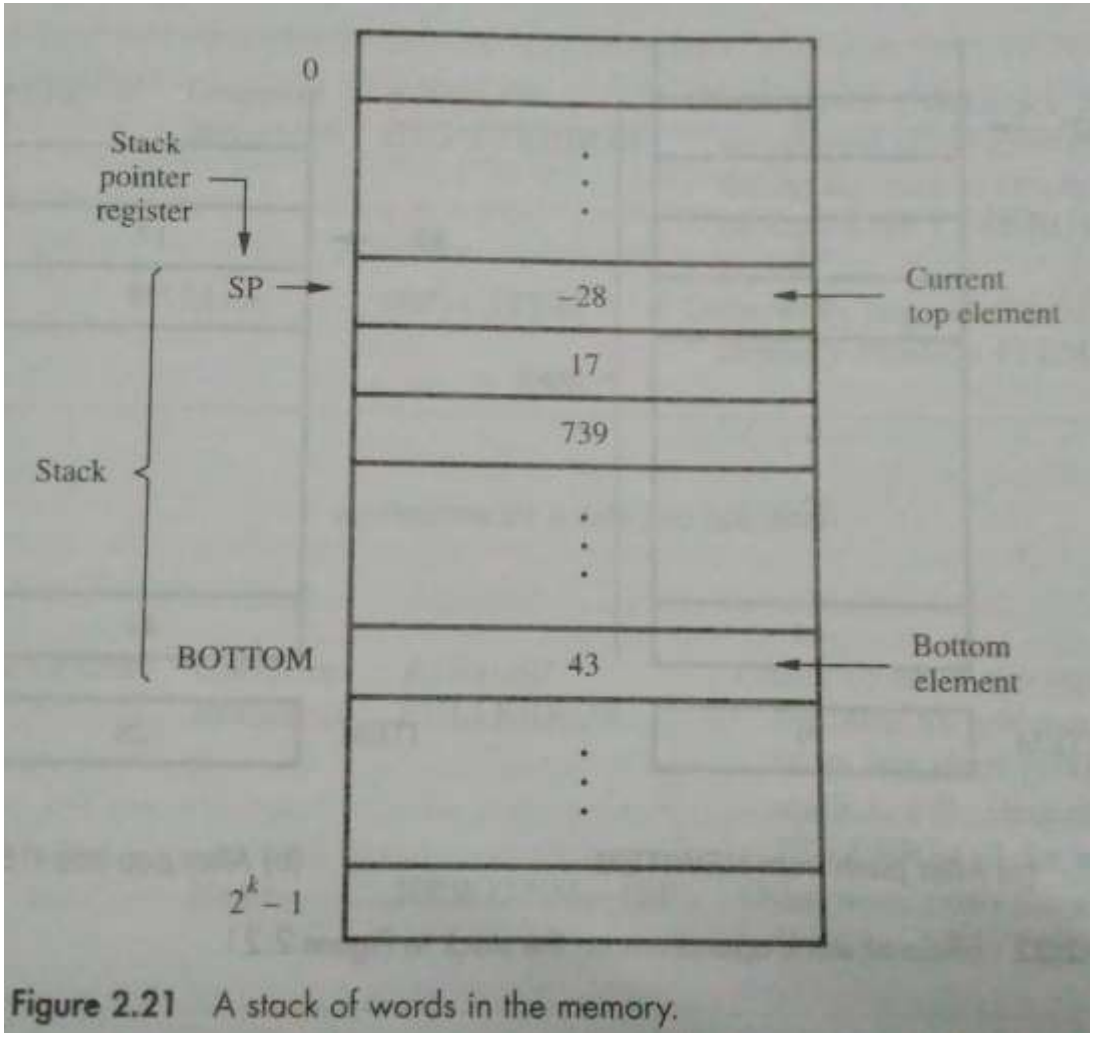


Figure 2.21 A stack of words in the memory.

- A stack of word data item in the memory of a computer
- It contains numerical values with 43 at the bottom and -28 at the top.
- Stack pointer is a processor register used to keep track of the address of the element of the stack that is at the top at any given time.
- Assume a byte addressable memory with 32- bit word length
- Push operation can be implemented as:

  Subtract #4,SP

  Move NEWITEM,(SP)

- Where, the subtract instruction subtract the source operand 4 from the destination operand contained in SP and places the result in SP.

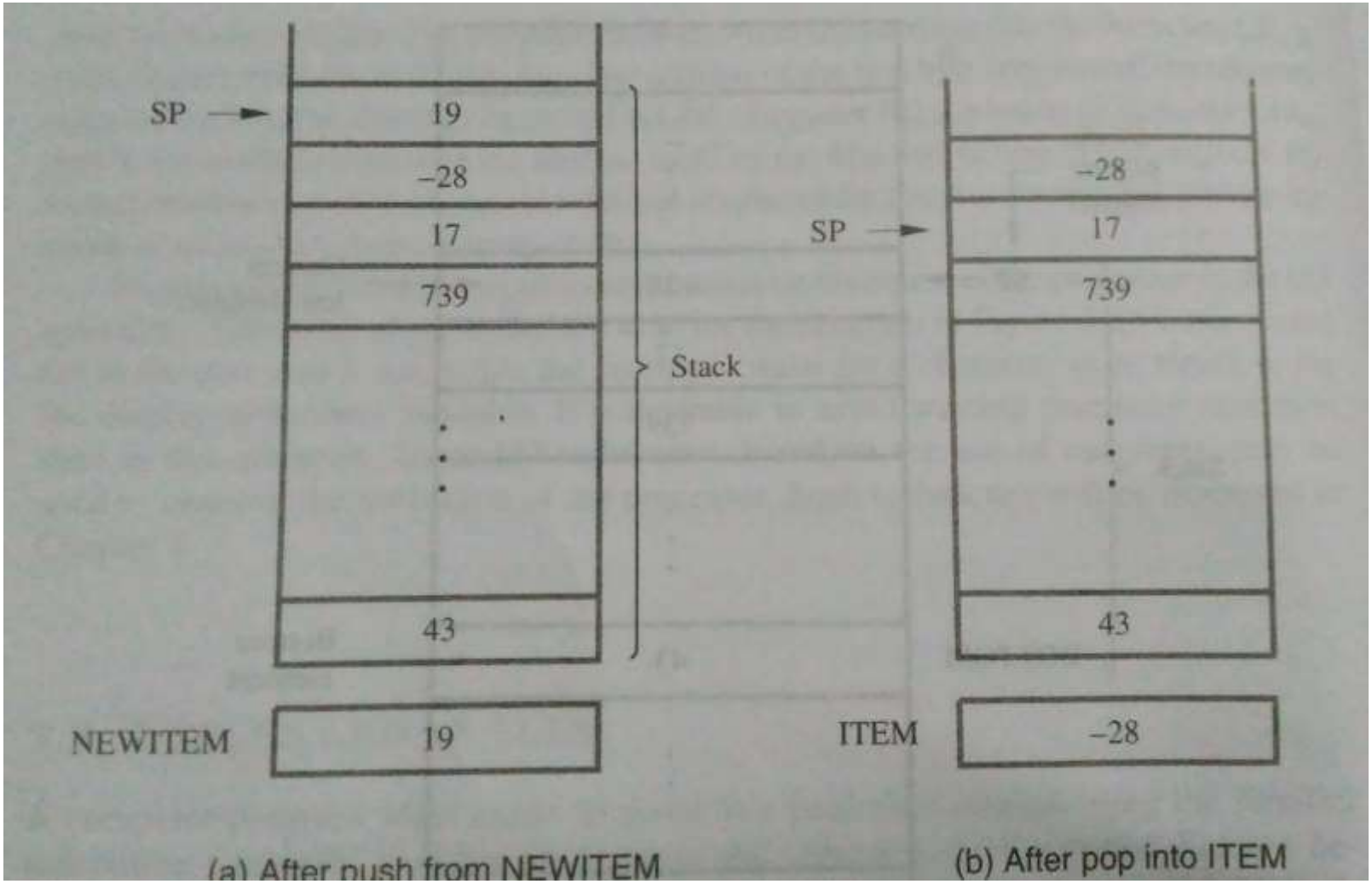
- Move the word from location NEWITEM into the top of the stack, decrementing the stack pointer by 4 before the move
- The Pop operation:

  Move(SP),ITEM

  Add #4,SP

  - Move the top value from the stack into location ITEM and then increment the stack pointer by 4 so that it points to the new top element

Effect of stack operation



- If the processor has the Auto increment and Auto decrement addressing modes, the push operation can be performed by a single instruction

  Move NEWITEM,-(SP)

- Pop operation can be performed by

  Move (SP)+,ITEM

SUBROUTINE

- In a given program, it is often necessary to perform a particular subtask many times on different data-values. Such a subtask is usually called a subroutine.
- For example, a subroutine may evaluate the sine function or sort a list of values into increasing or decreasing order.
- It is possible to include the block of instructions that constitute a subroutine at every place where it is needed in the program

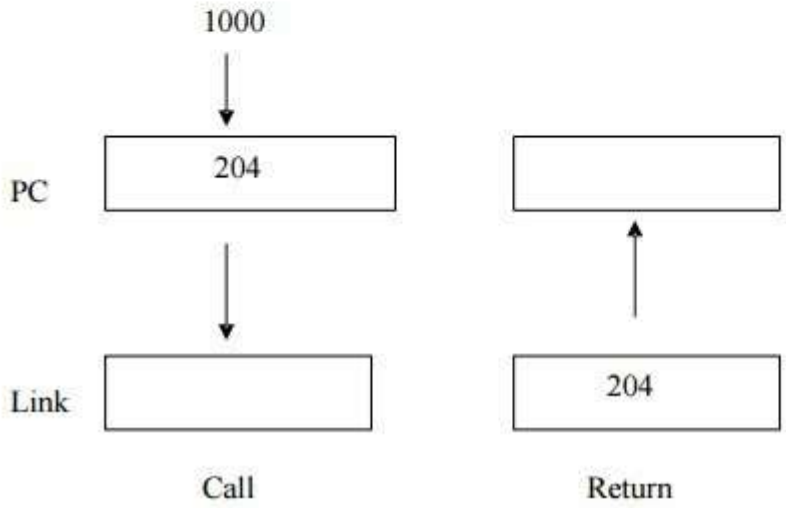
- The way in which a computer makes it possible to call and return from subroutines is referred to as its subroutine linkage method. The simplest subroutine linkage method is to save the return address in a specific location, which may be a register dedicated to this function. Such a register is called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

- The Call instruction is just a special branch instruction that performs the following operations • Store the contents of the PC in the link register • Branch to the target address specified by the instruction The Return instruction is a special branch instruction that performs the operation • Branch to the address contained in the link register .

Subroutine linkage using a link register

Fig a illustrates this procedure

Memory Location	Calling program	Memory location	Subroutine SUB
	....		
	....		
200	Call SUB	1000	first instruction
204	next instruction		....
	....		....
	....		Return
	....		



SUBROUTINE NESTING AND THE PROCESSOR STACK

- A common programming practice, called subroutine nesting, is to have one subroutine call another. 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 Memory location Subroutine SUB 1000 first instruction …. …. Return 204 Return sequence. That is, return addresses are generated and used in a last-in-first-out order. This suggests that the return addresses associated with subroutine calls should be pushed onto a stack. A particular register is designated as the stack pointer, SP, to be used in this operation. The stack pointer points to a stack called the processor stack

- The Call instruction pushes the contents of the PC onto the processor stack and loads the subroutine address into the PC. The Return instruction pops the return address from the processor stack into the PC.

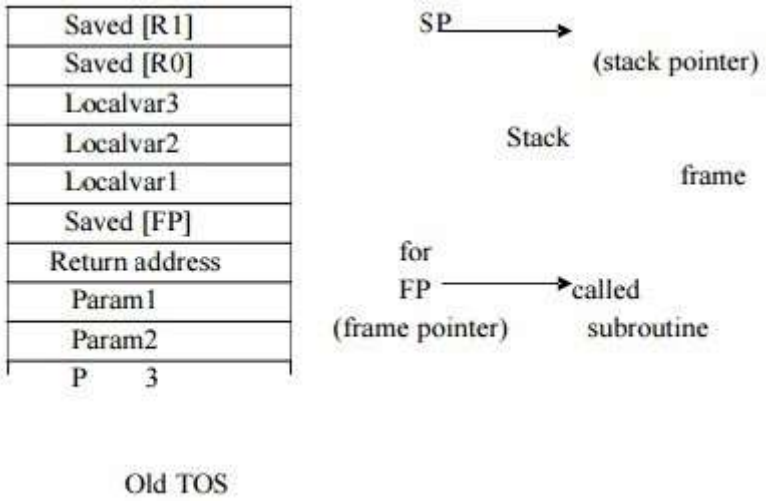
PARAMETER PASSING

- When calling a subroutine, a program must provide to the subroutine the parameters, that is, the operands or their addresses, to be used in the computation. Later, the subroutine returns other parameters, in this case, the results of the computation. This exchange of information between a calling program and a subroutine is referred to as parameter passing. Parameter passing may be accomplished in several ways. The parameters may be placed in registers or in memory locations, where they can be accessed by the subroutine. Alternatively, the parameters may be placed on the processor stack used for saving the return address

- The purpose of the subroutines is to add a list of numbers. Instead of passing the actual list entries, the calling program passes the address of the first number in the list. This technique is called passing by reference. The second parameter is passed by value, that is, the actual number of entries, n, is passed to the subroutine

THE STACK FRAME:-

- Now, observe how space is used in the stack in the example. During execution of the subroutine, six locations at the top of the stack contain entries that are needed by the subroutine. These locations constitute a private workspace for the subroutine, created at the time the subroutine is entered and freed up when the subroutine returns control to the calling program. Such space is called a stack frame. Fig a A subroutine stack frame example


Saved [R1]
Saved [R0]
Localvar3
Localvar2
Localvar1
Saved [FP]
Return address
Param1
Param2
P 3
SP
(stack pointer)
Stack
frame
for
FP
(frame pointer)
called
subroutine
Old TOS

- fig b shows an example of a commonly used layout for information in a stack frame. In addition to the stack pointer SP, it is useful to have another pointer register, called the frame pointer (FP), for convenient access to the parameters passed to the subroutine and to the local memory variables used by the subroutine. These local variables are only used within the subroutine, so it is appropriate to allocate space for them in the stack frame associated with the subroutine. We assume that four parameters are passed to the subroutine, three local variables are used within the subroutine, and registers R0 and R1 need to be saved because they will also be used within the subroutin

- The pointers SP and FP are manipulated as the stack frame is built, used, and dismantled for a particular of the subroutine. We begin by assuming that SP point to the old top-of-stack (TOS) element in fig b. Before the subroutine is called, the calling program pushes the four parameters onto the stack. The call instruction is then executed, resulting in the return address being pushed onto the stack. Now, SP points to this return address, and the first instruction of the subroutine is about to be executed. This is the point at which the frame pointer FP is set to contain the proper memory address. Since FP is usually a general-purpose register, it may contain information of use to the Calling program. Therefore, its contents are saved by pushing them onto the stack. Since the SP now points to this position, its contents are copied into FP

- Thus, the first two instructions executed in the subroutine are Move FP, -(SP) Move SP, FP After these instructions are executed, both SP and FP point to the saved FP contents. Subtract #12, SP Finally, the contents of processor registers R0 and R1 are saved by pushing them onto the stack. At this point, the stack frame has been set up as shown in the fig.

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

Module 1 related Questions

1.**What are fundamental phases of the instruction cycle?**

2. The register R1 = 12, and R2= 13. The instruction ADD R1, R2 is in memory location 2000H. After the execution of the instruction, what will be the value of PC, MAR, IR and R1?.

3.a) Discuss the sequencing of control signals for the following instructions. i) Load R1,10(R2) ii) Add R1, R2

b) Compare and contrast memory mapped IO over programmed IO.

4.a) Illustrate with example, explain the different types of addressing modes in a RISC processor.

b) Discuss how stack used for subroutine call.

5.What is meant by zero- address instruction?

6. Autoincrement mode is useful for accessing data items in successive memory locations. Justify the statement.

7.Explain the execution of a complete instruction.

a) Specify the actions needed to execute the instruction Move (R1), R2

b) What is the role of processor stack in subroutine call and return?

8.List various addressing modes explain any four with an example for each.

9. Draw the diagram of a multi-bus organization with 3 buses. Write the control sequence for the instruction Add R4, R5, R6 for the above mentioned multi-bus organization.

10. Give the sequence of control steps required to perform the operation Add [R3], R1 in a single-bus organization.

11.Illustrate the basic operational concepts in transferring data between main memory and processor with neat diagram.

12.Differentiate between big endian and little endian byte ordering.. Describe the basic instruction types.. Give the control sequence for execution of instruction *Add[R3],R1*

13.Describe the different addressing modes.

14Discuss the data path inside the processor with single bus organization with neat diagram

b) Write down the control sequence for the execution of the instruction
Add (R1), R2 in single bus organization

15.Show the effect of stack operations on the stack with diagram.

16.Give the relevance of MAR, PC and IR in a typical computer system with neat diagram.
17.Differentiate between Big-endian and Little-endian assignment for word addressing.

18. Illustrate the advantages of using multiple bus organization over single bus organization with the help of a sample instruction execution.

19.Assuming that stack grows towards lower address range write the following (Without using PUSH and POP) :

(i) Pushing elements stored at ITEM1, ITEM2 onto stack

(ii) Popping an element onto address ITEM

(iii) Copying value of top of stack to address TOP

Write the three-address, two-address and one-address representations of the operation below with relevant assumptions:

$C \leftarrow [A] + [B]$

20.With a neat diagram, explain the internal architecture of the CPU.

21.What are condition codes? List the different condition codes

22.Enumerate the sequence of actions involved in executing an unconditional branch instruction.

23.Register R6 is used in a program to point to the top of a stack containing 32-bit numbers. 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, add them, then push the result onto the stack.

(*b*) Copy the fifth item from the top into register R3. For each case, assume that the stack contains ten or more elements.

24. With the help of a diagram, describe the datapath inside the processor.

25.Write down the sequence of actions needed to fetch and execute the instruction: Store R6, X(R8)

26.Write the sequence of actions involved in executing an unconditional branch instruction

27.What are different types of bus structures?

28.What is addressing mode? Explain the following addressing modes with example (i)Immediate (ii) Indexed (iv)Auto increment

29.Illustrate stack operations with example?

30.Write and explain the control sequence to execute the instruction Add (R1), R2 for a single bus organization. Draw a neat diagram for single bus organization of CPU.

3`1.What is subroutine? Explain subroutine linkage method using link register.

1.1 Basic Structure of Computers

A Computer is a fast electronic machine that accepts input information in digital form, process the input according to a set of stored instructions (called programs) and outputs the resulting information.

Computer Types

Based on size, cost, computational power and intended use computers can be classified as below:

- **Personal computer**
  The most common form of desktop computers is personal computers. It has processing and storage units, visual display, audio output units and a keyboard that can be placed easily on office/home desk. Storage media include hard disk, CD ROM and diskettes.
- **Notebook Computers**
  These are compact version of personal computer. All components are packaged into a single unit, with the size of thin briefcase.
- **Workstations**
  Dimensions of workstations are same as that of desktop computers. They have high resolution graphics input/output capability and have more computational power than personal computers.
- **Mainframes(Enterprise systems)**
  These types of systems are used for business data processing. These have more computing power and storage capacity than workstations.
- **Servers**
  Contain sizable database storage units and are capable of handling large volumes of requests to access the data. These are widely accessible to the education, business and personal user communities. Requests and responses are usually transported over internet communication facilities.
- **Supercomputers**
  Super computers are high end powerful computer systems. Used for large scale numerical calculations required in applications such as weather forecasting. India's first Supercomputer is PARAM 8000 developed by CDAC (Centre for Development of Advanced Computing).

1.1.1 Functional Units

A computer consist of mainly five independent functional parts

- **Input Unit**
  This unit accepts information from human operators with the help of electomechanical devices such as keyboard. Whenever a key is pressed the corresponding letter or digit is automatically translated into its corresponding binary code and transmitted over a cable to either the memory or processor. The received information can be stored in computers memory for later references otherwise it can be immediately used by the ALU circuitry to perform the desired operations.
- **Memory Unit**
  It is the storage unit of the computer system. The input information is processed based on the stored instructions called programs; these programs are stored in memory unit. The

memory contains a large number of semiconductor storage cells where each cell can store one bit of information. The cells are processed in groups of fixed sizes called words, with a distinct address for each word. Addresses are simply numbers that can identify successive locations. The number of bits in each word determines word length of the computer (16 to 64 bits). Memory units are mainly classified into two types:

✓ **Primary Memory**

It is a fast memory operating at electronic speeds. Programs are stored in this memory during their execution period. Primary memory is divided into two types:

- **RAM(Random Access Memory)**

  It is a volatile memory that means the contents will lost when the power is switched off. RAM can again be divided into two types.

  - **Static Memory(SRAM)**

    Memories that consist of circuits that are capable of retaining their states as long as the power is applied are known as static memories. These memories are designed by using transistors and inverters.

  - **Dynamic Memory(DRAM)**

    These are less expensive RAMs. The cost of static RAMs are high because of the usage of the several transistors. Dynamic RAMs are implemented with the help of a capacitor and a single transistor. Such cells don't retain their state indefinitely hence they are called dynamic RAMs.

- **ROM(Read Only Memory)**

  This is one of the major types of memory using personnel computers. ROM is a type of memory that normally can only be read as opposed to RAM which can be both read and written. It is a non- volatile memory that is contents will retain even when the power is switched off. ROM can again be divided into three types:

  - **Programmable ROM(PROM)**

    This is a type of ROM that can be programmed using special equipment; it can be written to, but only once. This is useful for companies that make their own ROMs from software they write, because when they change their code they can create new PROMs without requiring expensive equipment. This is similar to the way a CD-ROM recorder works by letting you "burn" programs onto blanks once and then letting you read from them many times. In fact, programming a PROM is also called burning, just like burning a CD-R, and it is comparable in terms of its flexibility.

  - **Erasable Programmable ROM(EPROM)**

    An EPROM is a ROM that can be erased and reprogrammed. A little glass window is installed in the top of the ROM package, through which you can actually see the chip that holds the memory. Ultraviolet light of a specific frequency can be shined through this window for a specified period of time, which will erase the EPROM and allow it to be reprogrammed again. Obviously this is much more useful than a regular PROM, but it does require the erasing light. Continuing the "CD" analogy, this technology is analogous to a reusable CD-RW.

- **Electrically Erasable Programmable ROM(EEPROM)**
  The next level of erasability is the EEPROM, which can be erased under software control. This is the most flexible type of ROM, and is now commonly used for holding BIOS programs. When you hear reference to a "flash BIOS" or doing a BIOS upgrade by "flashing", this refers to reprogramming the BIOS EEPROM with a special software program. Here we are blurring the line a bit between what "read-only" really means, but remember that this rewriting is done maybe once a year or so, compared to real read-write memory (RAM) where rewriting is done often many times per second.

✓ **Secondary Memory**

This memory is used when large amount of data and programs have to be stored. Compared to the primary memory these are cheaper but performs at low speed. Examples are CD ROMS, Magnetic disk, USB drives etc.

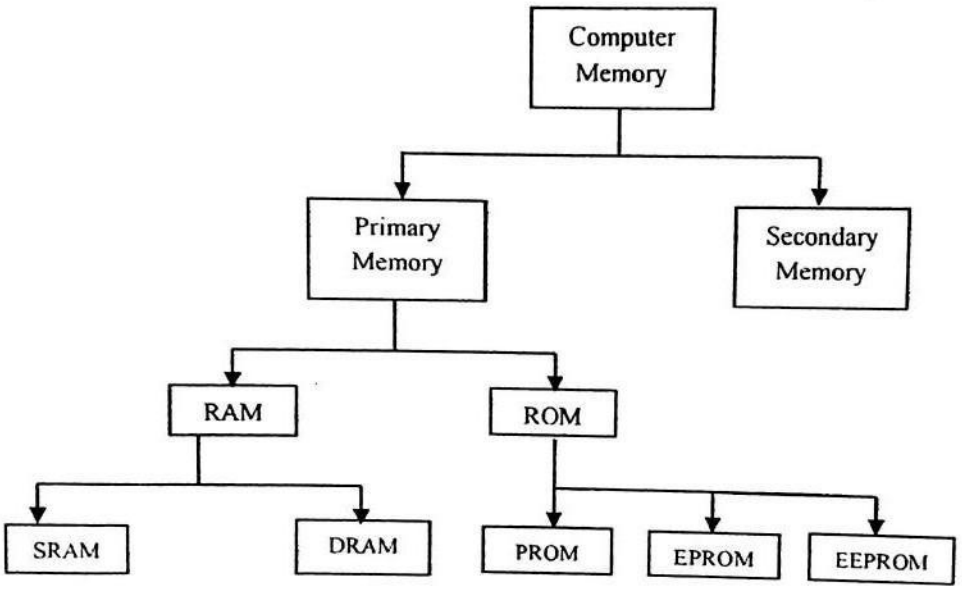


Fig 1.1: Classification of Memories

- **Arithmetic and Logic unit**
  This is the main part of the processing unit of a computer. The desired operations are performed by this unit. Arithmetic and logic operations can be separated with the help of a mode selector. When the mode selector bit is zero it performs arithmetic operations and performs logical operations when the bit is one. To perform the operation the required operands have to be bringing into the processor, where they are stored in registers.
- **Control unit**
  This is one of another core part of a processing unit of a computer. It is known as the nerve centre of a computer system. It coordinates the operation of all other units in computer system. It sends the control signals to other units and senses their states. Example of control signals are read, write etc.
- **Output unit**
  The processed results are sending to the outside world through output device. Example: Printer.

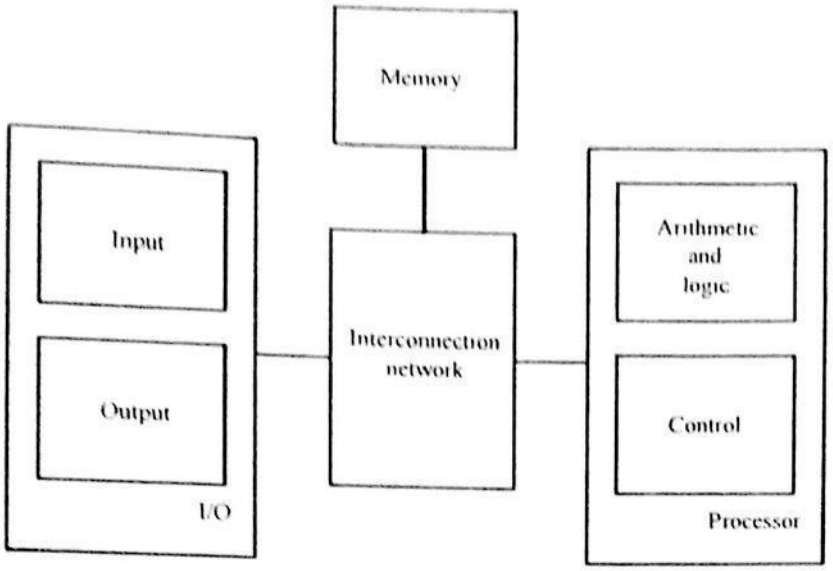


Fig 1.2: Basic Functional units of a computer

1.1.2 Basic Operational Concepts

In order to execute an operation in a processor the required instructions have to be brought out from the memory to the processor. Transfers between memory and processor are started by sending the address of the memory location to be accessed to the memory unit and issuing the appropriate control signals. Then the data are transferred to or from the memory. The following figure shows the connection between the memory and the processor.

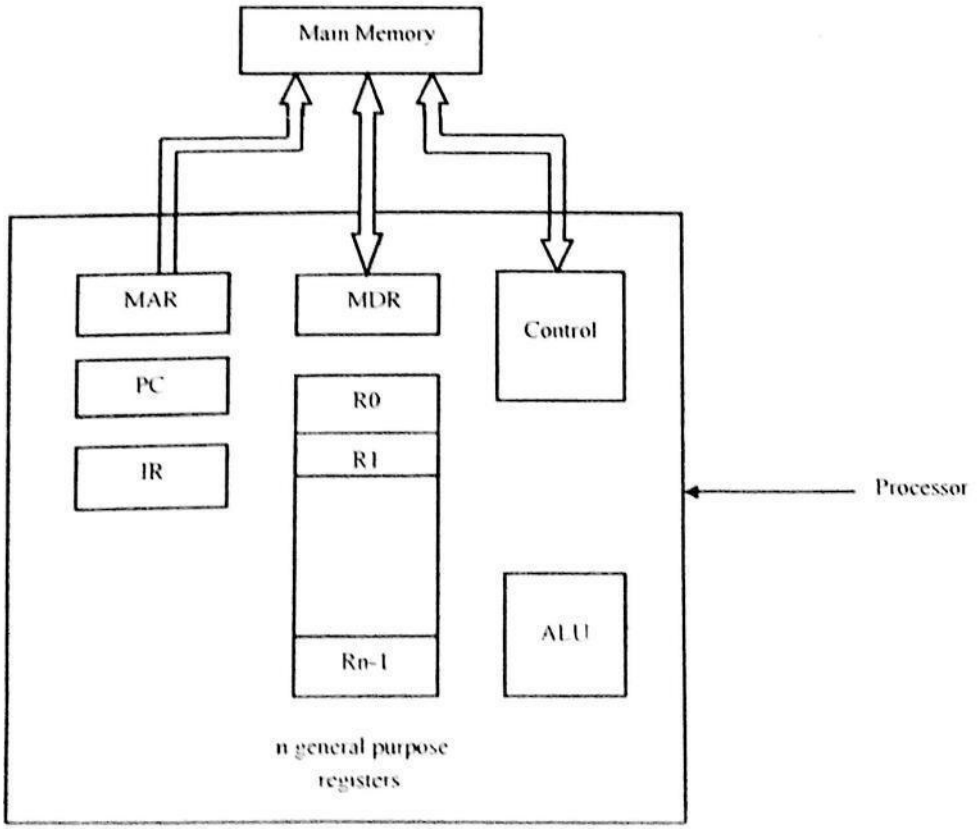


Fig 1.3: Connection between the processor and main memory

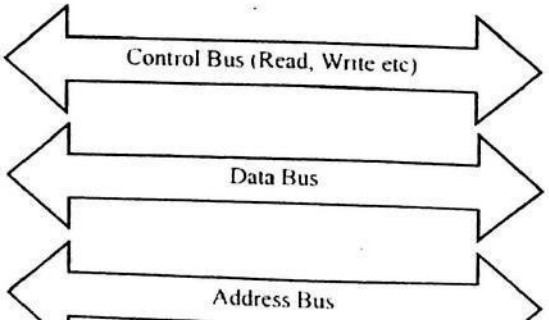
Memory Address Register (MAR) and Memory Data Register (MDR) are the two registers which are facilitating the communication between the processor and memory. In order to read an information from memory the address of the memory location in which the information is residing have to be put in MAR and a Read control signal will be sent to memory unit. When the memory unit sees the address in address line of the bus and read signal in control line of the bus memory will start the read operation from the concerned address and the result will be sent to the MDR. From there the information can be transferred to any other registers inside the processor through internal processor bus. Similarly for Write operation initially the data to be written into the memory has to be placed in MDR and the address in which the desired information have to be kept in memory will be placed in MAR and a Write control signal will be sent to the memory unit. When the memory unit sees the address, data and write signal in the external memory bus it will start the corresponding write operation.

Other than MAR and MDR few registers like PC(Program Counter), IR(Instruction Register) and some general purpose registers R_0, R_1,....R_{n-1} are there within the processor unit. PC is a register which holds the address of the next instruction to be fetched. Initially PC will be assigned by the starting program address and after fetching of each instruction it will be incremented by a size of word byte. IR is a register which holds the decoded instruction to be executed.

Normal execution of programs may be pre-empted if some device requires urgent servicing. In order to deal with the situation immediately the normal execution of the current program must be interrupted. To do this the device raises an interrupt signal. An interrupt is nothing but it is a request from an I/O device for service by the processor. The processor executes an interrupt service routine (ISR) to service the same. Before servicing an interrupt the current state of the processor must be saved in memory locations. After ISR is completed the state of the processor is restored so that the interrupted program may continue.

1.1.3 Bus Structures

Different functional units of a computer can be connected using a structure called bus system structure. There are external bus structures and internal bus structures. Bus interface between memory and processor is called external memory bus and bus structure inside the processor is called internal bus structures. Bus consists of three lines of carries, one for holding address, one for holding data and one for carrying control information.



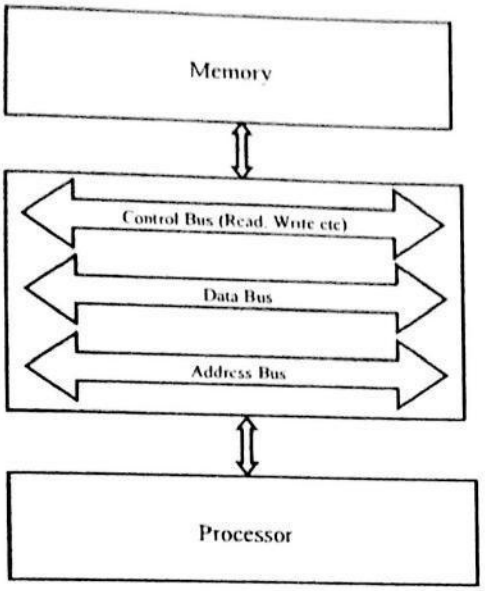


Fig 1.5: External processor bus

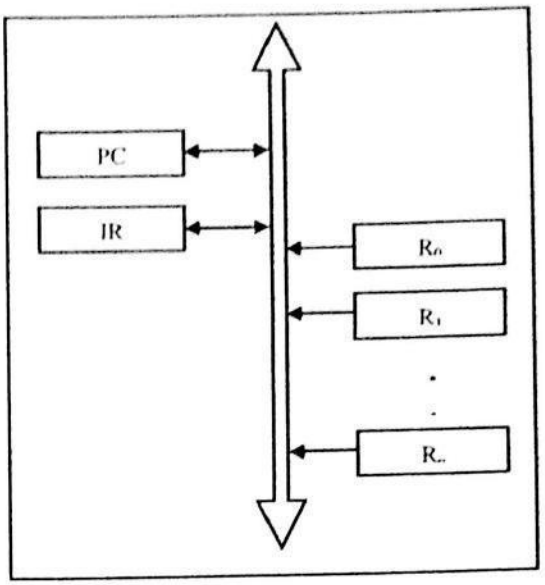


Fig 1.6: Internal processor bus

Bus Structures can be classified into two:

- **Single bus structure**
  Only two units can actively participate at a time, that is only one transfer takes place at a time.
  Advantages
  - Low cost
  - Flexibility for attaching peripheral devices

  Disadvantages
  - Only one transfer at a time
- **Multiple bus structure**
  It contains multiple buses, so that more than one transfer can takes place.
  Advantages
  - More concurrency in operations.
  - Better performance

  Disadvantages
  - Increased cost

The different units connected to the processor are having different speeds. These timing differences can be smoothened by including buffer registers with the devices to hold information during transfers. Once the buffer is full, the device can start the operation without further intervention by the bus and the processor.

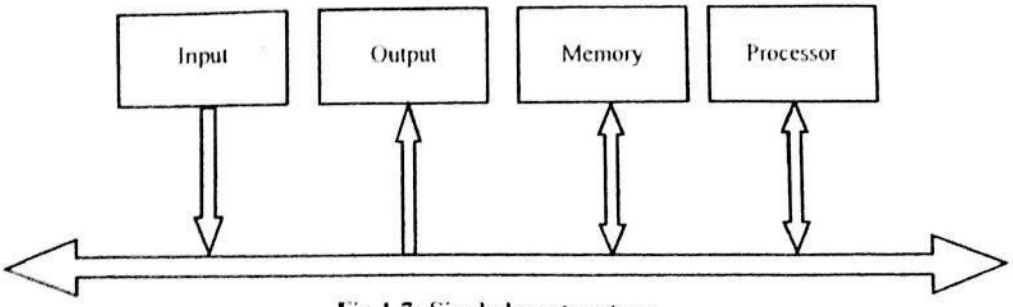


Fig 1.7: Single bus structure

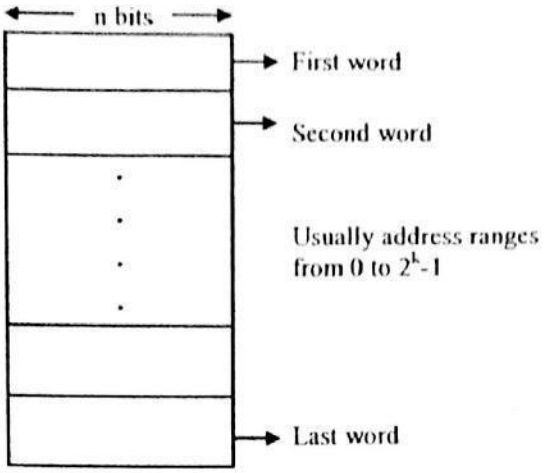


Fig 1.9: Memory Address

Note

16 bit address: - Creates an address space of 2^{16} addresses

32 bit address: - Creates an address space of 2^{32} addresses

Byte Addressability

Basic information quantities are, bit, byte and word. Byte is of size 8 bit (always this is constant).Word length may from 16 to 64 bits. So normally distinct addresses will be assigned to each byte location. This is known as byte addressability.

For example, if word length is 16 bits, successive words are located at addresses 0 and 4. If 32 bits, successive words will be located at addresses 0, 4, 8 ..., with each word consisting of four bytes.

Big-Endian and Little-Endian Assignments

Byte addresses can be assigned across words in two ways:

- **Big – Endian**
  Lower byte addresses are used for most significant bytes of the word.
- **Little – Endian**
  Lower byte addresses are used for least significant bytes of the word.

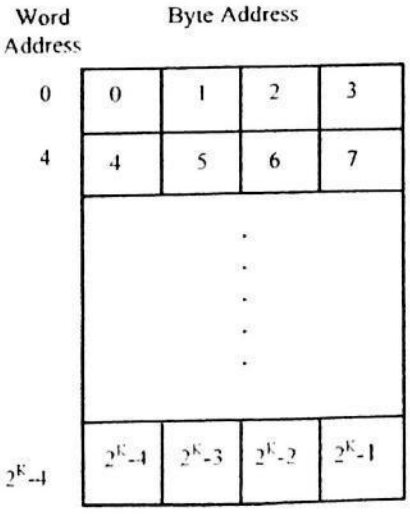


Fig 1.10(a): Big- Endian Assignment

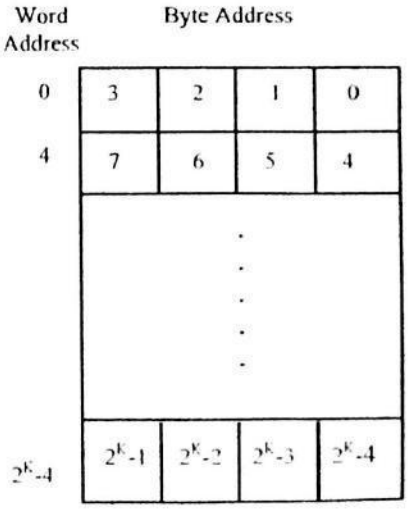


Fig 1.10(b): Little- Endian Assignment

Word Alignment

The words are said to be aligned in memory if they begin at a byte address that is a multiple of number of bytes in a word. The number of bytes in a word is power of 2.

Example

If word length is 16 bits, aligned words begin at byte addresses 0, 4, 8....

If word length is 64 bits, aligned words begin at byte addresses 0,8,16....

Words are said to have unaligned addresses if the words begin at arbitrary byte address.

Accessing Numbers, Characters and Character Strings

A number can be accessed in the memory by specifying its word address. Individual characters can be accessed by their byte address. Character Strings can be of variable length. The beginning of the string is indicated by giving the address of the byte containing the first character. A successive byte location contains successive characters of the string. End of string (a special control character) can be used as the last character in the string.

1.3 Memory Operations

To execute an instruction, the words containing the instruction have to be brought out to the processor from memory. Operands and results also have to be moved in between of main memory and processor. Main operations are:

- **Load(Read or Fetch)**
  Transfer copy of the contents of a specific memory location to the processor. The memory contents remain unchanged.
- **Store(Write)**
  It transfers information from processor to memory. It will destroy the earlier content of the memory location. Information can be transferred between processor and memory in terms of bytes or words. One byte or one word can be transferred in a single operation.

1.4 Instructions and Instruction Sequencing

A computer must have instructions capable of performing four type of operations:

- Data transfer between processor and memory registers
- Arithmetic and logic operations on data
- Program sequencing and control
- I/O transfers

➢ **Register- Transfer Notation**

Information can be transferred to and fro between memory locations, processor registers and registers with I/O. We can identify a location by a symbolic name.

Example

LOC, PLACE, A (Address of memory locations)
R0, R1, R5 (Processor register names)
DATA IN, OUT STATUS (I/O Registers)

Register- Transfer Examples

$R1 \leftarrow [LOC]$ ——— (1)

The contents of the memory location are transferred to processor register R1. Square bracket indicates the contents of the location.

$$R3 \leftarrow [R1] + [R2] \quad \text{——} (2)$$

Adds the contents of registers R1 and R2 and places the sum in register R3.The type of notations (1) and (2) are known as register transfer notation.

- **Assembly Language Notation**

The register transfer notation R1 ←[LOC] can be notated in assembly language as:

Move LOC, R1

Here the content of LOC is unchanged, R1 will be over written. Similarly,

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

This can be notated in assembly language as

Add R1, R2, R3

- **Basic Instruction Types**

Consider the statement C=A+B. To execute this statement, the operands A and B have to be fetched from memory to the processor. ALU computes the operation and the result will be sent back to memory and stored in location C.

- **Three Address Instructions**

Syntax

Operation source1, source2, destination

Example

Add A, B, C

Where, A and B are source operands and C is destination operand. Add is the operation to be performed on operands. Suppose that K bits are needed to specify the memory address of each operand, and then totally 3K bits are needed totally to specify the memory address of all operands in the above sample case. In addition, some bits are needed to denote Add instruction also. Three address instruction is too large to fit in one word for most cases. An alternative approach is to use two address instructions.

- **Two Address Instructions**

Syntax

Operation source, destination

Example

Add A, B

Which performs the operation B ←[A] + [B]. Square bracket indicates the contents of the location specified. That is, it adds the contents of A and B and the result is stored back to B. Here the value of location B will be overwritten. To preserve the value of B, we can go for another instruction Move B, C. It moves the content of B to C, leaving the

- **Data transfer between different locations**

Instructions used for data transfer between different locations are Move.

Syntax

Move Source, Destination

Example

Move A, Ri→same as Load A, Ri
Move Ri, A→same as Store Ri, A

Question

Write the instruction sequence for C=A+B (suppose that arithmetic operations are allowed only on register operands)

Answer

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

Question

Write the instruction sequence for C=A+B(suppose that one operand in memory, other in register)

Answer

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

- **Zero Address Instructions**

Instructions can be used with zero operands in which the locations of all operands are specified implicitly. (It is possible by storing the operands in a structure called pushdown stack).

- **Instruction Execution and Straight line sequencing**

This section deals with the flow of execution of a program .Consider a sample memory space for the program C←[A] + [B]

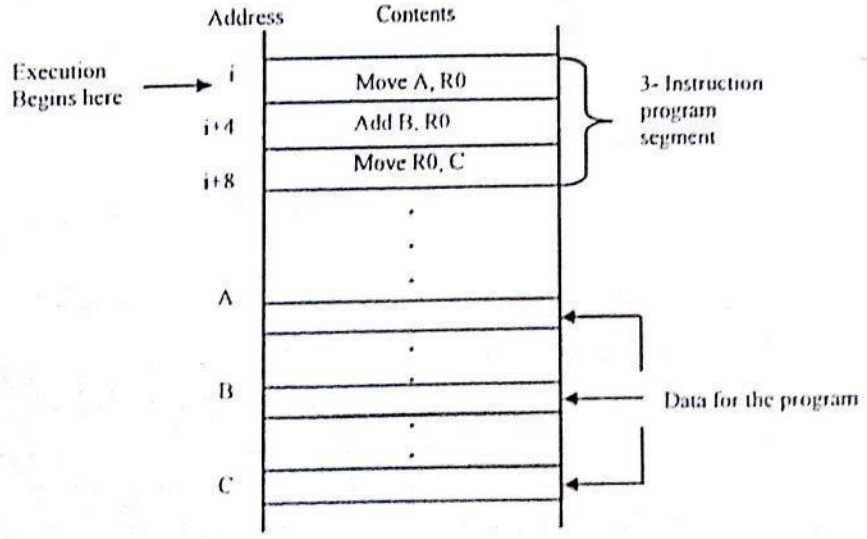


Fig 1.11: Program execution

Initially PC (Program Counter) contains the address of the next instruction to be executed. In this example initially address i is placed in PC. The processor control circuitry use the information in PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called straight line sequencing .While the instruction is being executed, the value of PC will be updated by incrementing 4 bytes, (because here length of instruction is 4 bytes). Instruction execution consists of 2 phases:

- **Instruction Fetch**
  Instruction is fetched from memory location whose address is in PC. This instruction is placed in IR (Instruction Register).
- **Instruction Execute**
  Instruction in IR (Instruction Register) is analyzed to see which operation have to be performed by the processor. This may include several operations like fetching of operands from memory (or from processor registers), performing an ALU operation, storing of result into memory etc.

After the execution phase, PC will contain the address of the next instruction to be executed. Then a new instruction fetch phase will begin.

➢ **Branching**

Consider a program of adding a list of n numbers (num1, num2....num n) and stores the result in memory location sum. By straight line sequencing approach, it can be done as follows:

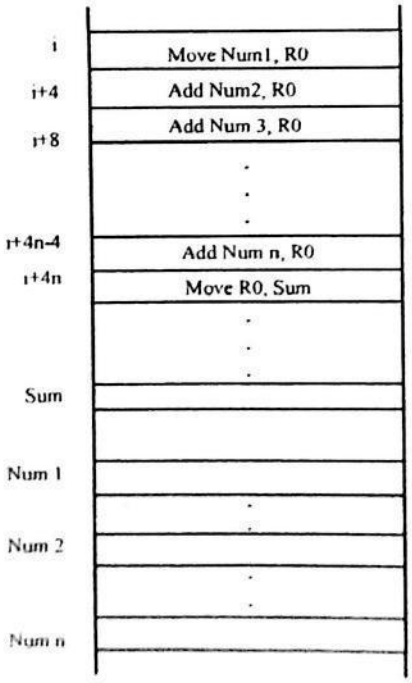


Fig 1.12: Straight Line Sequencing Program for adding 'n' numbers

Initially the value of first number (Num1) is moved to R0. Then it will be added with Num2 and result is stored in R0.Num3 will be read next and added with R0 (R0 now contains num1+num2+num3).The process will be continued with n numbers. After this, R0 contains a value (Num1+Num2+.....Num n).We have to store the result into location Sum. So, we are in need of an instruction,

Move R0, Sum

Here, a long list of add instruction is there. To remove this, we are going for a looping concept. The loop is a straight line sequence of instructions executed as many times as needed. It starts at the location Loop and ends at the instruction Branch>0.

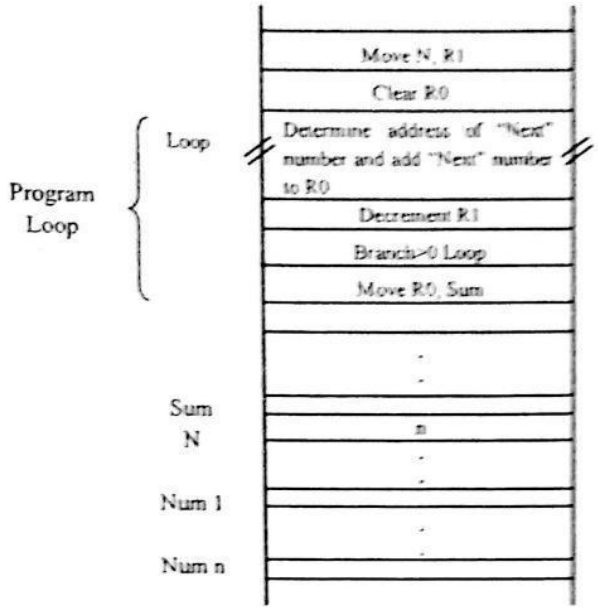


Fig 1.13: Using loop to add 'n' numbers

No of entries in the list (n) is stored at memory location N. After each addition operation this value is decremented by one. This value is a number greater than zero that means again there are numbers to be added to the list.

Branch Instruction loads a new value into the program counter .The processor fetches and executes the instructions from these addresses (known as branch target) instead of loading the instruction from address of PC. A conditional branch instruction causes a branch only if the specified condition is satisfied. In other cases, PC is incremented in the normal way and the next instruction in sequential address order is fetched and executed.

- **Condition Codes**

Whenever a conditional branch instruction is executed, it may require the results of various operations (to check the condition). Processors keeping this information in individual bits called condition code flags. These flags are grouped together in a special processor register called condition code Register or Status register.

Four commonly used flags are:

- N (Negative) → Set to 1 if result is negative, otherwise zero.
- Z (Zero) → Set to 1 if result is zero, otherwise cleared to zero.
- V (Overflow) → Set to 1 if arithmetic overflow occur, otherwise cleared to zero.
- C (Carry) → Set to 1 if a carry after the operation, otherwise cleared to zero.

In the above example, Branch>0 tests the condition code flags N and Z .That is ,the branch is taken only if register R either contain a negative or zero value.

1.5 Addressing Modes

The different ways in which the location of an operand is specified in an instruction is called as Addressing mode.

Generic Addressing Modes

- Immediate mode
- Register mode
- Absolute mode
- Indirect mode
- Index mode
- Base with index
- Base with index and offset
- Relative mode
- Auto-increment mode
- Auto-decrement mode

➢ **Implementation of Variables and Constants**

- **Variables**

  The value can be changed as needed using the appropriate instructions. There are 2 accessing modes to access the variables. They are

  - **Register Mode**

    The operand is the contents of the processor register. The name (address) of the register is given in the instruction.

  - **Absolute Mode (Direct Mode)**

    The operand is in new location. The address of this location is given explicitly in the instruction.

  Example

  MOVE LOC, R2

  The above instruction uses the register and absolute mode. The processor register is the temporary storage where the data in the register are accessed using register mode. The absolute mode can represent global variables in the program.

Mode Assembler	Syntax	Addressing Function
Register mode	Ri	EA=Ri
Absolute mode	LOC	EA=LOC

Where, EA is Effective Address

- **Constants**

  Address and data constants can be represented in assembly language using Immediate Mode.

➢ **Immediate mode**

The operand is given explicitly in the instruction.

Example

Move 200 immediate, R0

It places the value 200 in the register R0.The immediate mode used to specify the value of source operand. In assembly language, the immediate subscript is not appropriate so # symbol is used. It can be re-written as:

Move #200, R0

Assembly Syntax	Addressing Function
Immediate #value	Operand =value

- **Indirection and Pointers**

  Instruction does not give the operand or its address explicitly. Instead it 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 effective address of the operand is the contents of a register. We denote the indirection by the name of the register or new address given in the instruction.

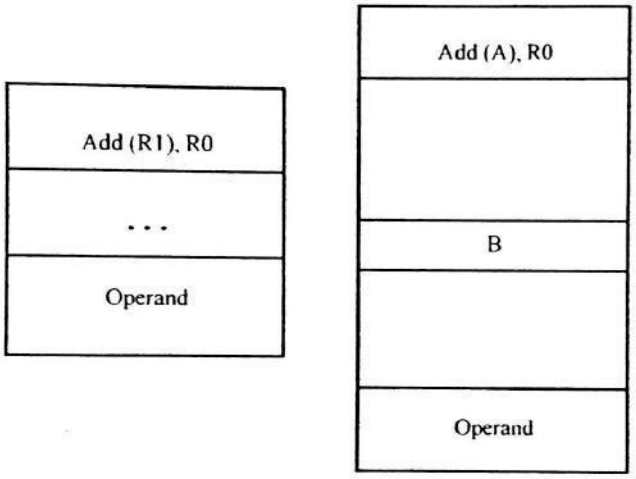


Fig 1.14: Indirect mode

Address of an operand (B) is stored into R1 register. If we want this operand, we can get it through register R1 (indirection). The register or new location that contains the address of an operand is called the pointer.

Mode	Assembler Syntax	Addressing Function
Indirect	Ri, LOC	EA=[Ri] or EA=[LOC]

➢ **Indexing and Arrays**

- **Index Mode**

  The effective address of an operand is generated by adding a constant value to the contents of a register. The constant value uses either special purpose or general purpose register. We indicate the index mode symbolically as,

  X(Ri)

  Where,

  X – denotes the constant value contained in the instruction

  Ri – It is the name of the register involved

  The Effective Address of the operand is,

  EA=X + [Ri]

  The index register R1 contains the address of a new location and the value of X defines an offset (also called a displacement).

To find operand,

1. First go to Reg R1 (using address)-read the content from R1=1000
2. Add the content 1000 with offset 20 get the result.

   1000+20=1020
3. Here the constant X refers to the new address and the contents of index register define the offset to the operand.
4. The sum of two values is given explicitly in the instruction and the other is stored in register.

   Example

   Add 20(R1) , R2 (or) EA=>1000+20=1020

Index Mode	Assembler Syntax	Addressing Function
Index	X(Ri)	EA=[Ri]+X
Base with Index	(Ri,Rj)	EA=[Ri]+[Rj]
Base with Index and offset	X(Ri,Rj)	EA=[Ri]+[Rj] +X

➢ **Relative Addressing**

It is same as index mode. The difference is, instead of general purpose register, here we can use program counter (PC).

- **Relative Mode**

  The Effective Address is determined by the Index mode using the PC in place of the general purpose register (gpr). This mode can be used to access the data operand. But its most common use is to specify the target address in branch instruction.

  Example

  Branch>0 Loop

  It causes the program execution to goto the branch target location. It is identified by the name loop if the branch condition is satisfied.

Mode	Assembler Syntax	Addressing Function
Relative	X(PC)	EA=[PC]+X

➢ **Additional Modes**

There are two additional modes. They are

- **Auto-increment mode**

  The Effective Address of the operand is the contents of a register in the instruction. After accessing the operand, the contents of this register is automatically incremented to point to the next item in the list.

Mode	Assembler Syntax	Addressing Function
Auto-increment	(Ri)+	EA=[Ri]; Increment Ri

- **Auto-decrement mode**

  The Effective Address of the operand is the contents of a register in the instruction. After accessing the operand, the contents of this register is automatically decremented to point to the next item in the list.

Mode	Assembler Syntax	Addressing Function
Auto-decrement	-(Ri)	EA=[Ri]; Decrement Ri

1.6 ARM

Advanced Risc Machine (ARM) Limited has designed a family of microprocessors. All ARM processors share the same machine instruction set.

Registers, Memory access and Data transfer

In ARM architectures, memory is byte addressable using 32 bit addresses. Processor registers are also 32 bits long. In moving of data between processor registers and memory, operand length may be 8 bit (byte) or words (32 bit).Both little Endian and big Endian addressing schemes are supported. Memory is accessed only by Load and Store instructions. All arithmetic and logic instructions operate only on data in processor registers. This arrangement is a basic feature of RISC Architectures.

Register Structure

There are sixteen 32 bit registers labelled R0 to R15. R0 to R14 are general purpose registers and one is dedicated as a program counter (PC). General purpose registers can hold either memory operands or data operands. The Current Program Status Register (CPSR) or simply status register holds the condition code flags, interrupt disable flags and processor mode bits. as described in the following figure.

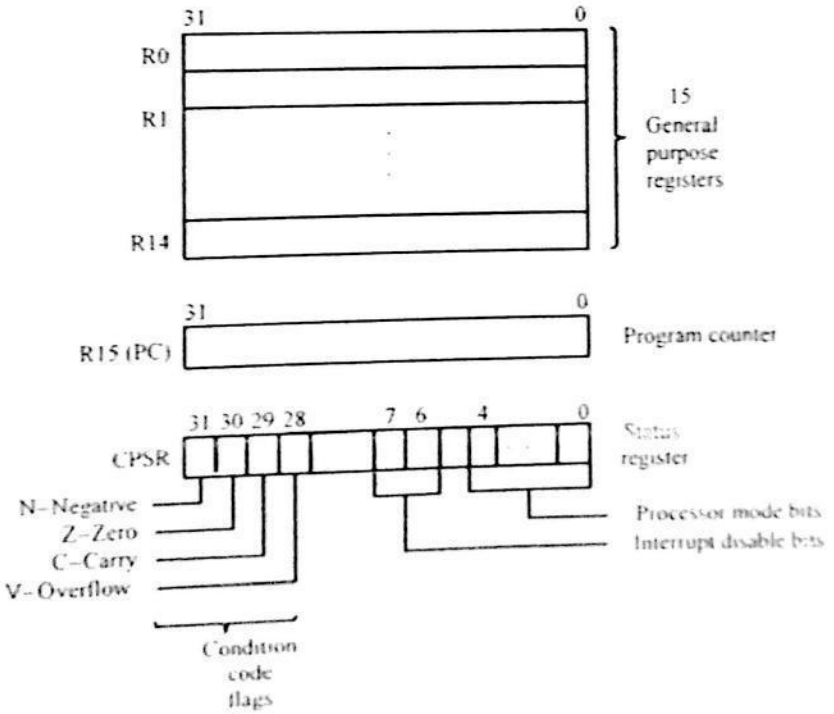


Fig 1.15: ARM Register Structure

There are 15 additional general purpose registers called the banked registers. They are duplicates of some of the R0 to R14 registers. They are used when the processor switches into supervisor mode.

Memory Access Instructions and addressing modes

In ARM, access to memory is provided with only Load and Store instructions. The basic encoding format is shown as in the following figure:

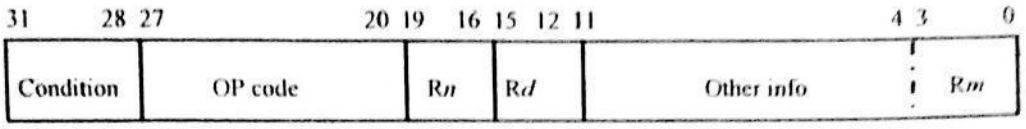


Fig 1.16: ARM instruction format

- **Conditional execution of instructions**

  Unlike others, in ARM processors all instructions are conditionally executed, depending on the condition specified in the instruction. Instruction is executed only when the condition flag is true. Otherwise the processor proceeds to the next instruction. One of the conditions is used to indicate that the instruction is always executed.

- **Memory addressing modes**

  For addressing memory operands one of the basic method is generate an EA(Effective Address) of the operand by adding a signed offset to the contents of the base register Rn(which is specified in the instruction).The magnitude of offset may be either an immediate value or the contents of the register Rm.

  Examples

  LDR Rd, [Rn, #offset]

  It performs the operation Rd ←[[Rn] +offset]

  LDR Rd, [Rn, Rm]

  It performs the operation Rd ←[[Rn] + [Rm]]

  If a negative offset is used, Rm must be preceded by a minus sign. An offset of zero doesn't have to specify explicitly. That is,

  LDR Rd, [Rn]

  It performs the operation Rd ←[[Rn]]

  A byte operand can be moved by using the Opcode LDRB. Similarly Store has the mnemonics STR and STRB.

  Example

  STR Rd, [Rn]

  It performs the operation [Rn] ←[Rd]

  Generally we can define three addressing modes in ARM processors.

  - **Pre-indexed mode**:

    Effective address of the operand is the sum of contents of base register Rn and an offset value.

  - **Pre-indexed with write back mode**:

    It is working in the same way as pre-indexed mode except that effective address is written back to Rn.

  - **Post-indexed mode**:

    The effective address of the operand is the contents of Rn. The offset is then added to this address and the result is written back into Rn.

Register Move Instructions

To copy the contents of register Rm into register Rd, ARM uses the following instruction

MOV Rd, Rm

To load an immediate value in the register Rd, the instruction will be

MOV Rd, #immediate value

Example

MOV R0, #70

Places the value 70 in register R0.

Arithmetic and Logic Instructions

ARM instruction set has a number of arithmetic and logic operations. The operands may be in general purpose registers or may give as an immediate operand. Memory operands are not allowed in these instructions.

- **Arithmetic Instructions**

  The general format for arithmetic instruction is,

  OP code Rd, Rn, Rm

  Operation specified by the OP code is performed on operands in general purpose registers Rn and Rm. The result is placed in register Rd.

  Example

  ADD R0, R2, R4

  Adds the content of R2 and R4 and places the sum in register R0.

  It performs the operation, R0←[R2] + [R4]

  ADD R0, R3, #17

  Adds the content of R3 and 17 and stores the sum in R0.

  It performs the operation, R0←[R3] +17

  The immediate value is contained in the 8 bit field on bits b_{7-0} of the instruction. The second operand can be shifted or rotated before being used in the instruction. When a shift or rotation is required, it is specified last in the assembly language expression for the instruction.

  Example

  ADD R0, R1, R5, LSL #4

  The second operand contained in register R5 is shifted left 4 bit positions and it is then added to the contents of register R1 and sum is placed in Register R0. Two versions of multiply instructions are there.

  1. Multiplies the contents of two registers and places the low order 32 bits of the product in a third register. Higher order bits of the product, if any, are discarded.

     MUL R0, R1, R2

     It performs the operation, R0 ←[R1] * [R2]

  2. Second version called Multiply Accumulate specifies a fourth register whose contents are added to the product before storing the result in the destination register.

     MLA R0, R1, R2, R3

     It performs the operation. R0 ←[R1] * [R2] + [R3]

     This method is often used in numerical algorithms for digital signal processing.

1.7 Basic I/O Operations

Input (I)/Output (O) operations are essential in a computer system, because data have to be transferred from memory of a computer to the outside world. The way in which I/O is performed has a significant role in the performance of a computer system.

Consider a task that reads a character from the keyboard and produces character output on a display screen. A simple way of performing such I/O task is to use a method known as program controlled I/O. Whenever a key is pressed on the keyboard, that character code has to be moved to the processor. Similarly, for display the same, that character code have to be moved from processor to display device. But when this transfer takes place, processor is very fast compared to the I/O device (keyboard and display). So, some sort of synchronization mechanism we are in need off. A solution to this problem is as follows:

The processor waits for a signal from the keyboard indicating that a character key has been struck and that its code is available in some buffer register associated with the keyboard. This register is known as DATAIN (8 bit buffer register).A status flag SIN is used to signal the processor. A program monitors SIN value, 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.

Similarly, with the output device, processor sends the first character and then waits for a signal from the display to know that character has been received. A buffer register DATAOUT and a status control flag SOUT is used for this transfer. When SOUT is 1, the display is ready to receive the character. A program monitors the value of SOUT and when this is equal to 1, processor transfers a character code to DATAOUT. Transfer of a character to DATAOUT clears SOUT to 0. When the display device is ready to receive a second character, SOUT is again set to 1. The above specified I/O transfers are accomplished with the help of machine instructions.

A processor can monitor SIN flag and transfer character from DATAIN to register R1 with the following sequence of operations.

```
READWAIT    Branch to READWAIT if SIN=0
            Input from DATAIN to R1
```

Analogous sequence of operations for transferring output to the display is

```
WRITEWAIT   Branch to WRITEWAIT if SOUT=0
            Output from R1 to DATAOUT
```

Branch operation is normally implemented by two machine instructions. The first instruction checks the status flag and the second instruction performs branch. Status flags are monitored by executing a short wait loop. We assume that initial state of SIN is 0 and SOUT is 1. This will be done by the device control circuits.

If the scheme used is memory mapped I/O instead of program control I/O, it is possible to use the same instruction set used for memory access. In such cases the contents of DATAIN and DATAOUT can be moved to R1 by using the instruction Move. (Load, store etc can also be used) as in the following instruction.

```
Movebyte        DATAIN, R1
```

Similarly contents of DATAOUT can be moved to R1 as,

```
Movebyte        R1, DATAOUT
```

Status flag SIN and SOUT are automatically cleared when the buffer registers are referenced. (Note: Difference between Move and Movebyte is in Move operand is word operands and for Movebyte operand size is byte.) Status flags also can addressed as part of memory address space by assigning distinct addresses. But the common practice is including SIN and SOUT in device status registers. Bit b3 in registers INSTATUS and OUTSTATUS is used for this purpose.

Read operation can now be implemented with the machine instruction sequence as shown below:

```
READWAIT    Testbit     #3, INSTATUS
            Branch=0    READWAIT
            Movebyte    DATAIN, R1
```

Write operation may implemented as,

```
WRITEWAIT   Testbit     #3, OUTSTATUS
            Branch=0    WRITEWAIT
            Movebyte    R1, DATAOUT
```

Testbit instruction checks the value of status flags. If the value of test bit is 0, then the condition of the branch instruction is true, and a branch is made to the beginning of the wait loop. When the device is ready (at that time bit tested will be equal to 1) the data are read from the input or written into the output buffer.

```
            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 in the keyboard buffer DATAIN
            Branch=0    READ
            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 output 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 back and read another character
            Branch≠0    READ            Also increment the pointer to store the next character
```

Table 1.1: A program that reads a line of characters and displays it

1.8 Stack Subroutine Calls

In this section we are dealing with the concepts of subroutines, stacks and how a subroutine call can be performed with the help of stack data structure.

Stacks

Stack is a data structure which contains a list of data elements usually words or bytes, with the accessing restriction that elements can be added or removed at one end of the list only. This end is called top of the stack, and the other end is called the bottom. The structure is also refered as push down stack or LIFO (Last In First Out) stack. The operations on stack are push (inserting an item into the stack) and pop (remove an item from the stack). Data stored in the memory of a computer can be organized as a stack, with successive elements occupying successive memory locations. First element is placed in location BOTTOM and when new data are added they are placed in successively lower address locations. Here we are considering a stack which grows in the direction of decreasing memory addresses.

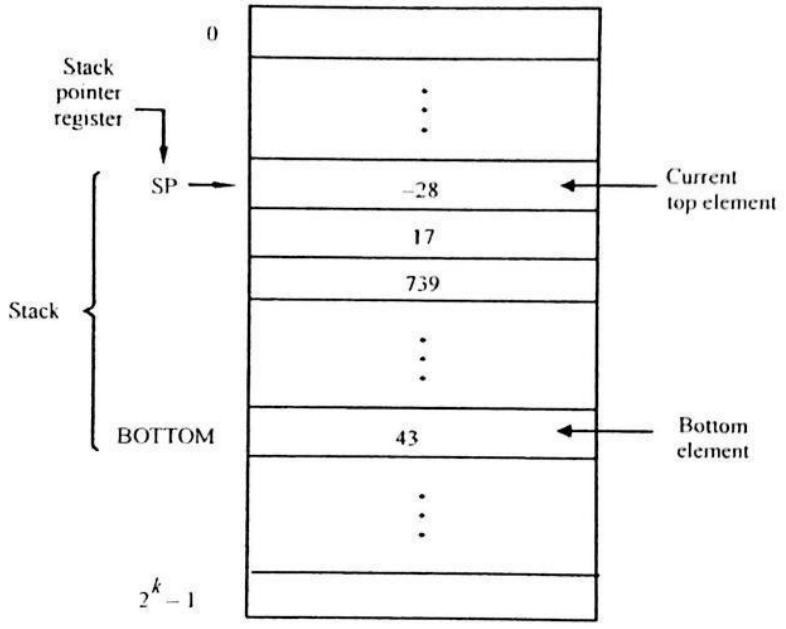


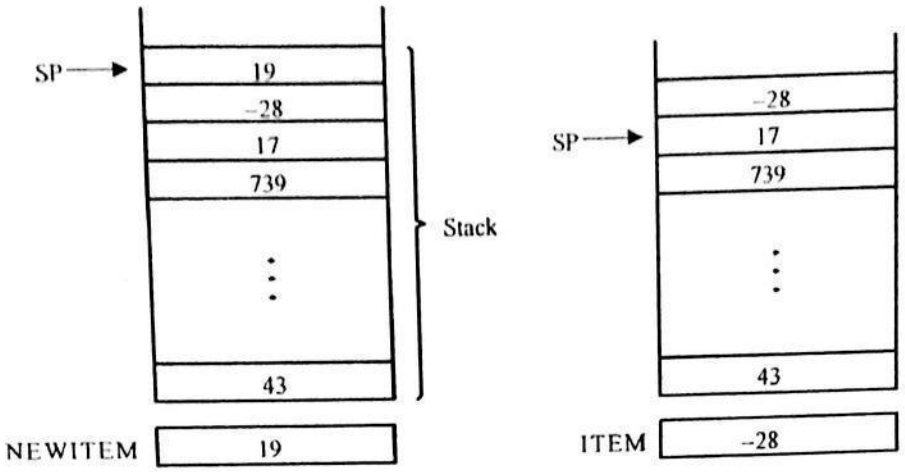
Fig 1.18: A stack of words in the memory

A processor register is used to keep track of the address of the element of the stack that is at top at any given time. This register is called stack pointer (SP). Push and Pop operations can be implemented as follows (Assume that 32 bit word length that is 4 bytes word length)

Push:	Subtract	#4, SP
	Move	NEWITEM, (SP)
Pop:	Move	(SP), ITEM
	Add	#4, SP

If the processor has auto increment and auto decrement addressing modes, the above operations can be replaced with a single instruction as follows:

Push:	Move	NEWITEM, - (SP)
Pop:	Move	(SP) +, ITEM



(a) After push from NEWITEM (b) After pop into ITEM

Fig 1.19: Effect of stack operations on the stack in fig 1.18

When a stack is used in a program, it is usually allocated a fixed amount of space in the memory. In this case, we have to avoid pushing a data item when the stack has reached its maximum size. Similarly always have to avoid a pop operation from an empty stack (otherwise it will lead to program error).

Subroutines

A computer program often needs to perform a particular subtask many times with different data values. This subtask is known as subroutines. Subroutine may consist of a block of instructions. At every place of the program where this subroutine is needed we can place these entire block of instructions. But this will waste more space. So, to save space only one copy of the instructions that constitute the subroutine is placed in the memory. Any program that requires the use of this subroutine simply branches to the starting location of subroutine. When a program branches to a subroutine it is known as calling a subroutine. The instruction that performs this branch operation is known as CALL instruction.

After a subroutine has been executed, the calling program has to resume the execution continuing immediately after the instruction that called the subroutine. At the end of the subroutine a RETURN instruction will be there, which transfer the flow of control to the appropriate location. In order to do so, these location addresses have to store somewhere. Normally this will be the address of the location pointed to by the updated PC (Program Counter) while the CALL instruction is being executed. Hence, the contents of the PC must be saved by the CALL instruction to enable correct return to the calling program.

- **Subroutine Linkage**

  The way in which computers make it possible to call and return from a subroutine is referred as subroutine linkage. Here, a register named as link register is used to store the

return address. When a subroutine completes its task, return instruction return to the calling program by branching indirectly through link register.

CALL instruction:
1. Store the contents of PC to link register
2. Branch to the target address specified by the instruction

RETURN instruction:
1. Branch to the address contained in the link register.

Sub Routine Nesting and Processor Stack

One subroutine call in another subroutine is known as subroutine nesting. 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 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 happen at any depth. The return address are generated and used in a last in first out order. So, a stack may be the best option to store the return address associated with subroutine calls. Stack pointer register is used for this purpose. Stack pointer points to a stack called processor stack. The call instruction pushes the contents of PC onto the processor stack and loads the subroutine address into PC. The Return instruction pops the return address from processor stack into the PC.

Parameter Passing

While calling a subroutine, calling program has to pass the necessary operands or address to subroutine for processing the information. Similarly subroutines may return the results of computation to the calling program. This exchange of information between calling program and subroutine is termed as parameter passing.

Parameter passing can be done in several ways.
1. Parameters may place in registers.
2. Parameters may place in memory locations
3. Parameters may place on processor stack.

Parameters may pass with two mechanisms:
1. Passed by value: actual data is passed to the subroutine
2. Passed by reference: address of actual data is passed to the subroutine.

Stack Frame

A private work space (in stack) for the subroutine, which is created at the time the subroutine is entered and freed up when the subroutine returns control to the calling program is called stack frame. If the subroutine requires more space for local memory variables, they can also be allocated on the stack. A common layout of stack frame is shown in below figure.

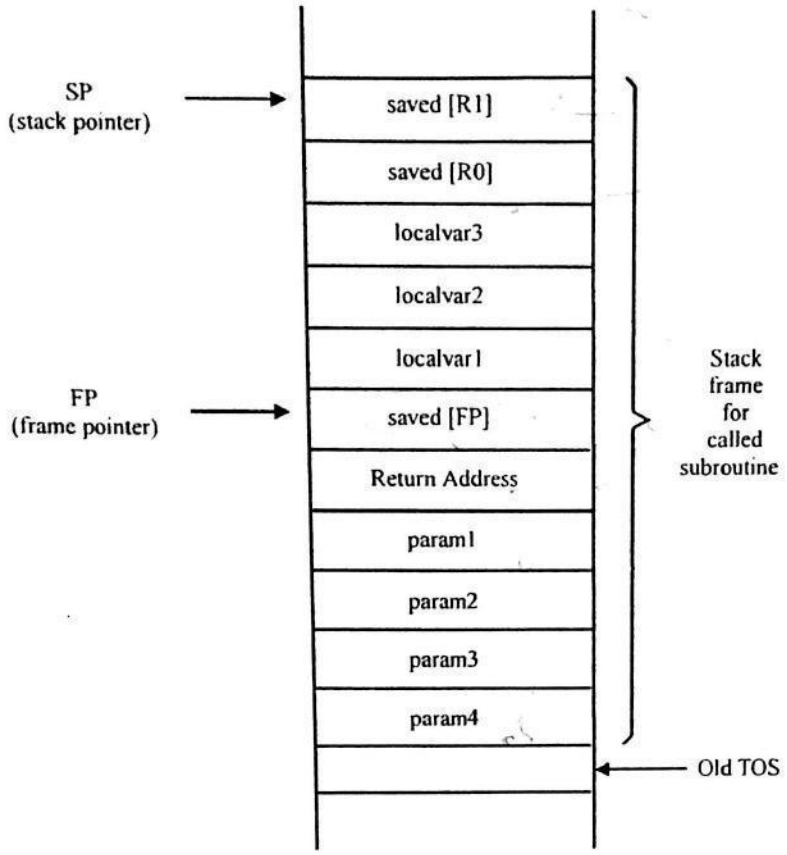


Fig 1.20: A subroutine stack frame example

In addition to stack pointer SP, it is useful to have another pointer FP (Frame Pointer). FP is useful for the accessing of parameters passed to the subroutine and the local memory variables used by the subroutine.

The contents of FP remain fixed throughout the execution of the subroutine. But stack pointer SP always point to the current top element of the stack which may vary. As FP is fixed we can use the indexed addressing mode to access the parameters and local variables. (For example, -4(FP), 8(FP))

Initially SP is pointed to the old TOS. Before the subroutine is called the calling program pushes the four parameters onto the stack. Then the CALL instruction is executed. Then return address is pushed onto the stack. Now SP points to this return address, and the first instruction of the subroutine is about to be executed. At this point FP is set to contain the proper memory address. Since FP is a general purpose register, it may contain information of use to the calling program; its contents are saved by pushing them on to the stack. Since SP now points to this position, its contents are copied into FP. Thus, the first two instruction executed in the subroutine are,

Move FP, - (SP)
Move SP, FP

After these instructions are executed, both SP and FP point to the saved FP contents. Space for three local variables are allocated by executing the instruction

Subtract #12, SP

2.1 Basic Processing Unit

The information's accepted from the users are processed with the help of processing unit. The instruction set of a processor may vary depends on the architecture. In this section we are dealing with how the instructions are processed within the processing unit.

2.1.1 Fundamental Concepts

To execute a program, the processor fetches one instruction at a time and performs the specified operations. Program Counter (PC) and Instruction Register (IR) are the main CPU registers who is participating in this task. PC contains the address of the instruction which is going to be executed next. The decoded instruction which is going to be currently executed is stored in register IR. The following are the general steps used to execute an instruction:

- Fetch the contents of memory location pointed by PC. This is actually referred as the instruction to be executed. Then they are loaded into IR. This can be symbolically represented as:

$$IR \leftarrow [[PC]]$$

- Update the value of PC. Suppose that every instruction has 4 bytes. Then PC will be updated by 4. Symbolically it can be represented as;

$$PC \leftarrow [PC] + 4$$

(The content of PC will be incremented by 4)

- Carry out the action specified in IR.

The fig 2.1 shows the single bus organization of data path inside the CPU. The ALU registers and the internal processor bus together known as "data path". The internal processor bus is used for communication between the various units inside the CPU. A separate external memory bus will be there to communicate between processor and memory.

Normally in the instruction execution following operations may be happen in different sequences:

- Register transfer
- Performing an arithmetic and logic operation
- Fetching a word from memory
- Storing a word in memory

The order of these operations may be different in separate sequences. In some of the operations, some steps may not be necessary too. Here, we are discussing each of these operations in detail.

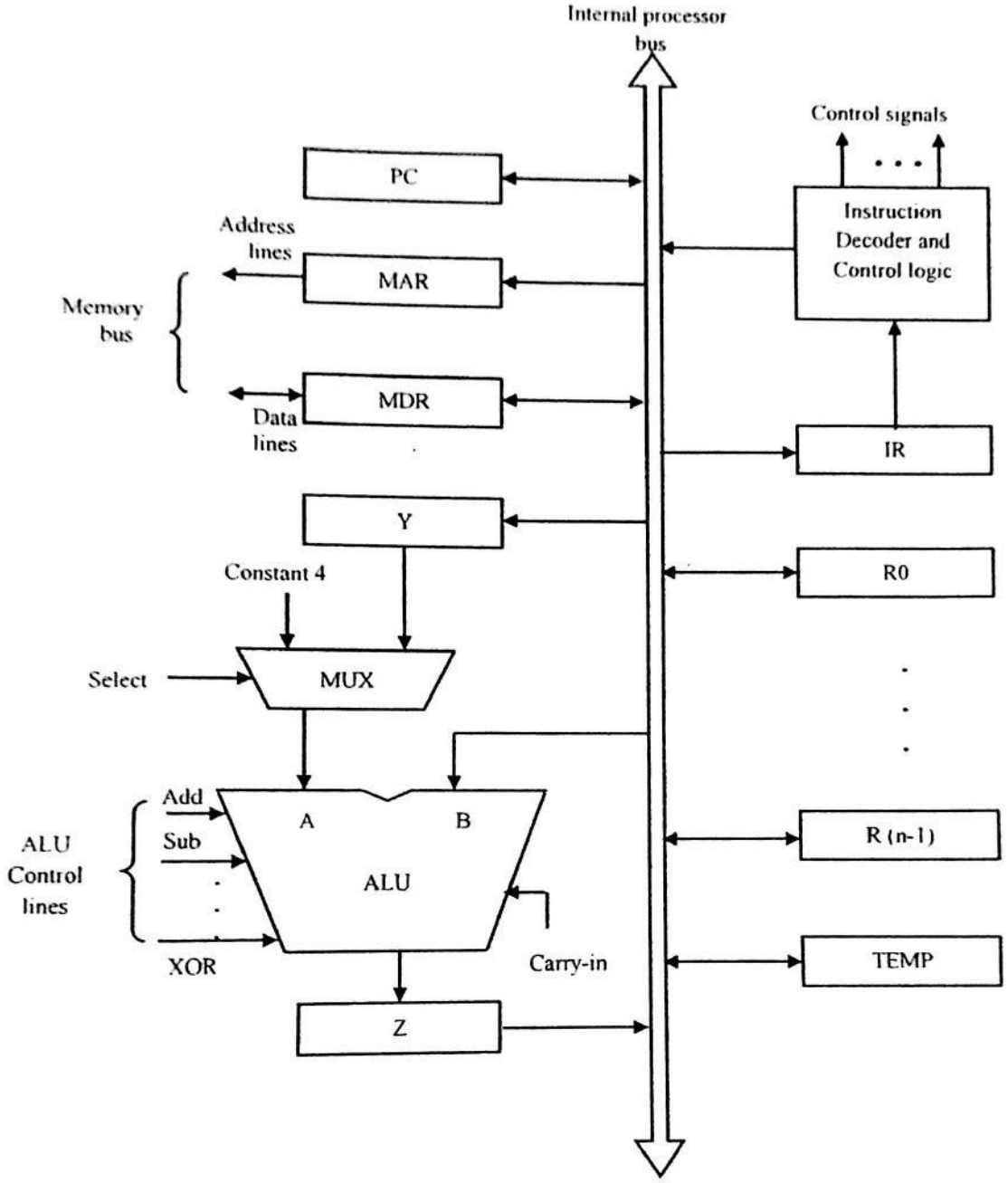


Fig 2.1: Single-bus organization of the datapath inside a processor

a) Register transfer

Associating with each registers there are two signals in and out. The input and output of the register R_i are connected to the bus via switches controlled by R_{in} and R_{iout}. When R_{in} is set to 1 the data on the bus are loaded in R_i. If R_{iout} is set to 1 the contents registers R_i are placed on bus.

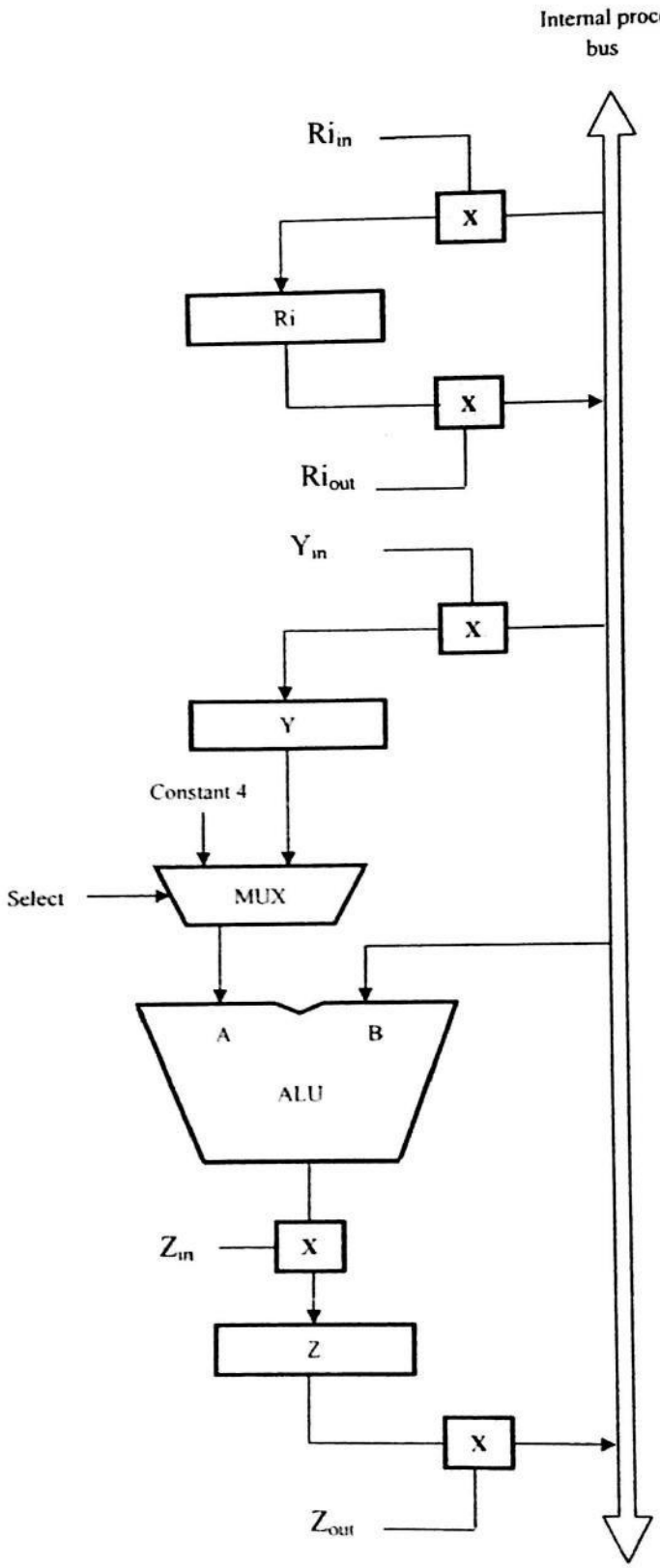


Fig 2.2: Input and output gating for the registers in Fig 2.1

Example

Transfer the contents of register R_1 to R_4.

- Set R_{1out} to 1. This places the contents of R_1 to the bus.

- Set R_{4in} to 1. This loads data from the processor bus to register R_4

b) Performing an arithmetic and logic operation

ALU is a combinational circuit and has no internal storage. It performs arithmetic and logic operations on the two operands applied to its A and B inputs. In the above figure we can see that, one of the operand is the output of the MUX and the other operand is obtained directly from the bus. The result produced by ALU is temporarily stored in register 'Z'. Therefore a sequence of operations to add the contents of register R_1 to those of register R_2 and store the result in register R3 is:

- R_{1out} , Y_{in}
- R_{2out} ,Select Y ,add, Z_{in}
- Z_{out} , R_{3in}

Example

Subtract the contents of R4 from R_5 and store the result in R_6.

- R_{4out} , Y_{in}
- R_{5out} ,Select Y ,sub, Z_{in}
- Z_{out} , R_{6in}

c) Fetching a word from memory

To fetch a word of information from the memory the processor has to specify the memory location, where this information is stored and requests a read operation. This applies whether the information to be fetched represents an instruction in a program or an operand specified by an instruction. The processor transfers the required address to MAR whose output is connected to address lines of external memory bus. At the same time the processor uses the control lines of the memory bus to indicate that a read operation is needed. When the requested data is received from the memory they are stored in the register MDR. From there they can be transferred to other registers in the processor. The following figure shows the connection and control signals for register MDR.

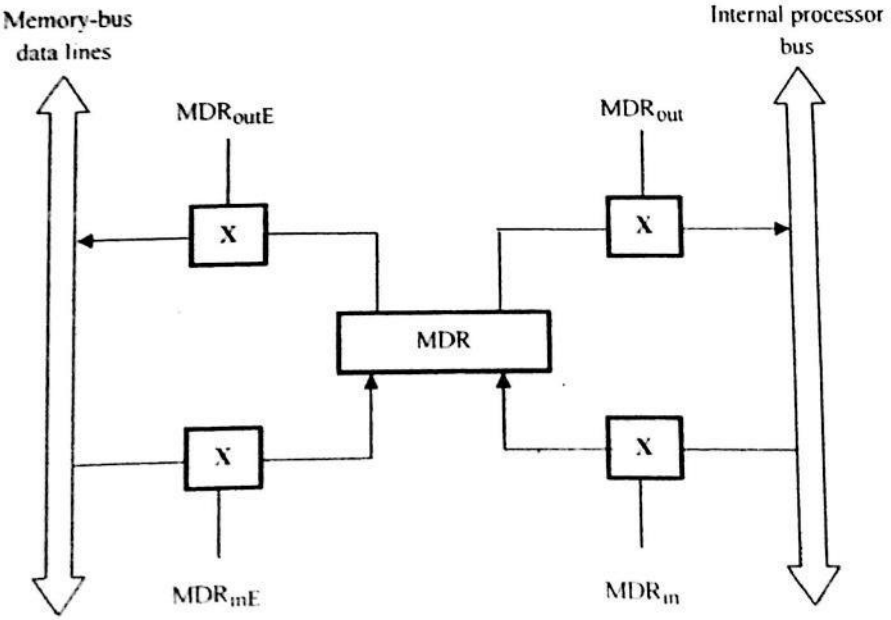


Fig 2.3: Connection and control signals for register MDR

Example MOVE (R_1), R_2

$R1_{out}$, MAR_{in}, READ
MDR_{inE}, WMFC
MDR_{out}, $R2_{in}$

MFC (Memory Function Complete) signal is used for indicating that the requested read or write operation has been completed.

d) Storing a word in memory

Writing a word into a memory location follows a similar procedure. The desired address will be loaded in MAR, the data to be written are loaded in MDR, and then a write command is issued.

Example MOVE R2, (R1)

$R1_{out}$, MAR_{in}
$R2_{out}$, MDR_{in}, WRITE
MDR_{outE}, WMFC

The action specified in one step can be completed in one clock cycle. Exemption may come for some steps like MFC depending on the speed of the addressed device.

2.1.2 Instruction Cycle

An instruction cycle (sometimes called a fetch–decode–execute cycle) is the basic operational process of a computer. It is the process by which a computer retrieves a program instruction from its memory, determines what actions the instruction dictates, and carries out those actions. In simpler CPUs the instruction cycle is executed sequentially, each instruction being processed before the next one is started. In most modern CPUs the instruction cycles are executed concurrently, and often in parallel, through an instruction pipeline: the next instruction starts being processed before the previous instruction has finished, which is possible because the cycle is broken up into separate steps.

The main steps in the instruction cycle are Fetch and Execution. Initially an instruction is fetched from the memory unit based on the address contained in PC. The PC value is updated by a value of size of the instruction so that PC can point to the next instruction to be executed. Then the decoded instruction is placed in IR, which is ready to be executed. (Execution phase: Refer2.1.3)

2.1.3 Execution of a Complete Instruction

A sequence of elementary operations we have discussed so far which are required to execute an instruction. They can put together to execute one instruction.
Consider the instruction,

ADD (R3), R1

Which adds the contents of the memory location pointed by R3 to register R1.The execution of this instruction requires the following actions:

1. Fetch the instruction.

2. Fetch the first operand (contents of the memory location pointed by R3)
3. Perform addition.
4. Load the result into R1.

The following figure shows the sequence of control steps to perform these operations for the single bus architecture.

Step	Action
1	PC_{out}, MAR_{in}, READ, SELECT 4, ADD, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	R_{1out}, MAR_{in}, READ
5	R_{1out}, Y_{in}, WMFC
6	MDR_{out}, SELECT Y, ADD, Z_{in}
7	Z_{out}, R_{1in}, END

Table 2.1: Control sequence for execution of the instruction Add (R3), R1

Explanation: The first three steps are common to every instruction. Instruction fetch operation is initiated by loading the contents of PC into MAR (Memory Address Register) and sending a read request to the memory. The select signal is set to 4 so that the multiplexer select the input as constant 4. This value is added to the operand at B (this is the value of PC) and the result is storing in register Z. Then this updated value is moved to PC from register Z. This is specified in control sequence step 2. In step 3, word fetched from the memory is loaded into IR.

From step 4 onwards the sequence (execution sequences) will be different for every instruction. Here, in the example instruction, contents of register R3 are transferred to MAR in step 4, and a read operation is specified. Then the contents of register R1 are transferred to register Y in step 5. When the read operation is completed, memory operand is available in MDR and the addition operation can be performed. The contents of MDR are gated to the bus, and are taken as input B of ALU circuit. Register Y is selected as the second input to the ALU by choosing select Y. The sum will be getting in register Z and will transfer to register R1 in step 7. The End signal causes a new instruction fetch cycle to begin by returning to step 1.

In the above execution of add instruction there is no need of Yin signal in step 2.But in the case of branch instruction ,the updated value of PC is needed to compute the branch target address. For this purpose only we are transferring the value of PC into register Y. This is the reason for adding Y_{in} in step 2. The fetch phase is common for all instructions.

Branch Instructions

Usually the address of the next instruction to be executed will be obtained from PC register. When branching takes place, the address is obtained by adding an offset X (which is given in the branch instruction) to the updated value of PC.

The following figure shows the control sequence that implements an unconditional branch instruction. Fetch phase is same as the above instruction. In step4, offset value is extracted from IR by the instruction decoding circuit. Updated PC value is already available in register Y. Offset X is gated onto the bus in step 4 and an addition operation is performed. The result, which is branch target address, is loaded into the PC.

Step	Action
1	PC_{out}, MAR_{in}, READ, SELECT 4, ADD, Z_{in}
2	Z_{out}, PC_{in}, Y_{in}, WMFC
3	MDR_{out}, IR_{in}
4	Offset-field –of-IR_{out}, Add, Z_{in}
5	Z_{out}, PC_{in}, END

Table 2.2: Control sequence for an unconditional Branch instruction

Now a conditional branch can consider. Here we need to check the status of condition codes before loading a new value into the PC. This can be done by the control sequence

Offset-field –of-IR_{out}, Add, Z_{in}, If N=0 then End

in step 4 of above table 2.2. In this case if N=0 the processor returns to step1 immediately after step4. If N=1, step 5 is performed to load a new value into the PC, thus performing the branch operation.

2.1.4 Multiple-Bus Organization

In the case of single bus organization, only one data item can be transferred over the bus in a clock cycle. The resulting control sequence will be lengthy in such cases. To reduce the number of steps needed, most of the processors are providing multiple internal paths so that several transfers are possible during a clock cycle.

The following figure shows a three bus structure. Registers and the ALU are connected to this bus. All general purpose registers are combined into a single block called register file. The register files have 3 ports. There are two output ports and one input port. Contents of two different registers can be accessed simultaneously and have their contents placed on buses A and B by using two output ports. The third port allows the data on bus C to be loaded into a third register during the same clock cycle.

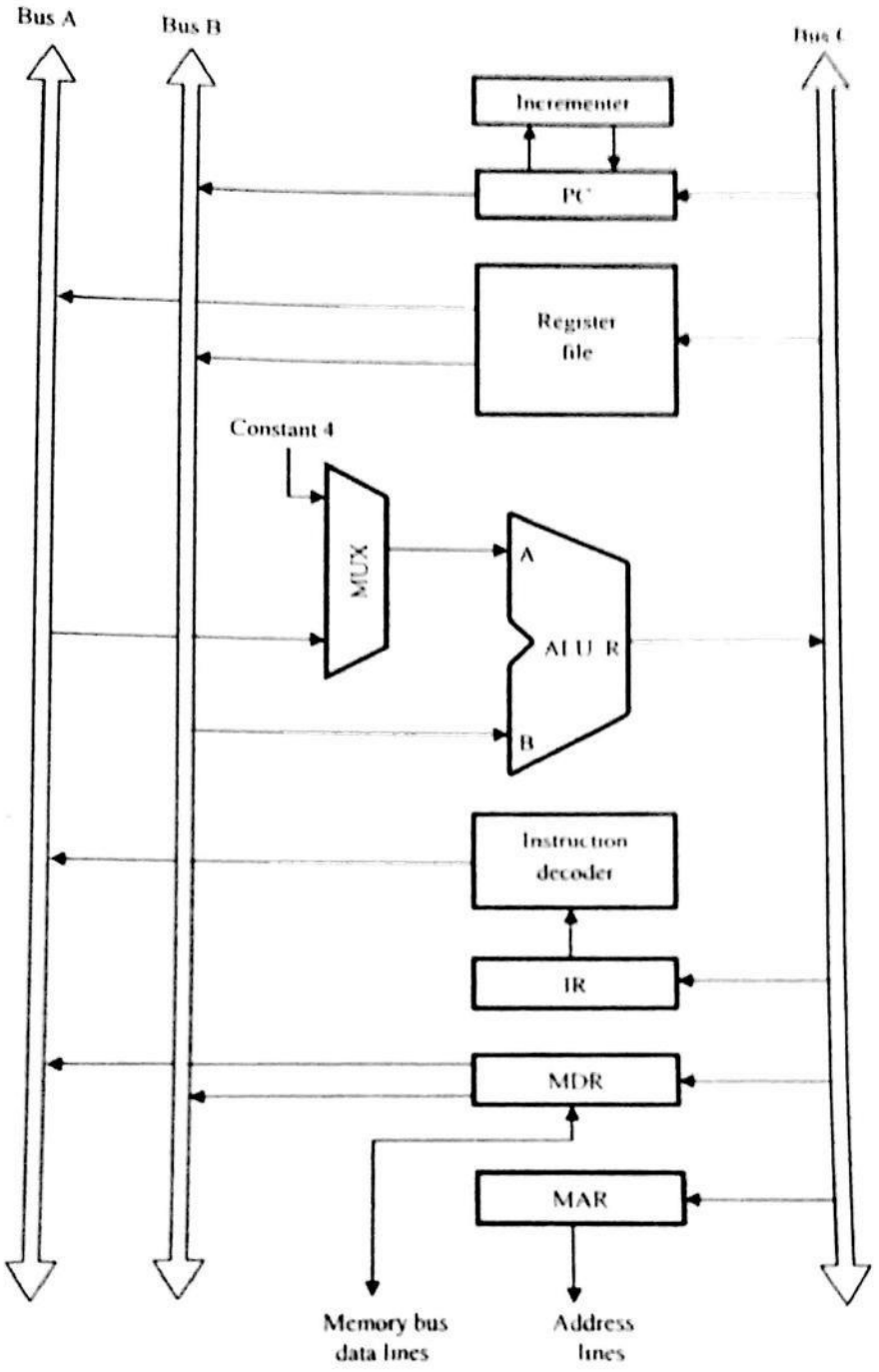


Fig 2.4: Three-bus organization of the datapath

The need of temporary registers Y and Z are not there with three bus arrangement. Buses A and B are used to transfer the source operands to the A and B inputs of the ALU.ALU performs the operation and the result is transferred to the destination over bus C.

Another feature of multiple buses is the introduction of incrementer unit. The purpose of this unit to increment the value of PC by 4. (We are assuming the word size as 4 bytes). This eliminates the need to add 4 to the PC using the main ALU. The constant4 in the ALU input multiplexer is still useful because it can be used to increment other memory addresses in LoadMultiple and StoreMultiple instructions.

Example

Write the control sequence for the instruction Add R4, R5, R6

Ans

Step	Action
1	PC_{out}, R=B, MARin, Read, IncPC
2	WMFC
3	MDR_{outB}, R=B, IR_{in}
4	$R4_{outA}$, $R5_{outB}$, Select A, Add, $R6_{in}$, END

Table 2.3: Control sequence for the instruction Add R4, R5, R6 for the three-bus organization in fig 2.4

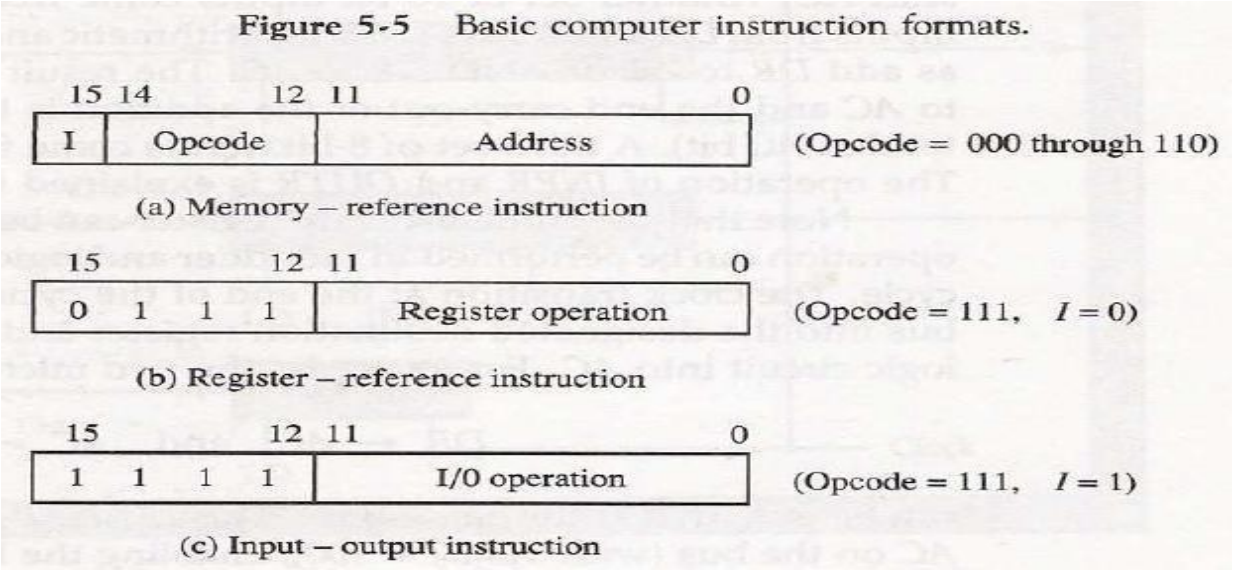
Explanation: In step 1, the contents of the PC are passed through the ALU using R=B signal, and loaded into MAR to start a memory read operation. At the same time PC is incremented by 4. The value loaded into MAR is the original content of PC. The incremented value is loaded into the PC at the end of the clock cycle only. In step2 the processor waits for MFC signal and loads the data received into MDR. The data in MDR is transferred to IR in step3. The execution phase needs only one step as in step4.Content of R4 (one operand of add) is placed on bus A, content of R5 (one operand of add) in bus B and then addition operation is performed on the specified operands. Result is transferred to register R6.

The number of clock cycles for instruction execution is significantly reduced in multiple bus organization.

1. Computer Instructions:

The basic computer has three instruction code formats, as shown in Fig. 5-5. Each format has 16 bits.

Figure 5-5 Basic computer instruction formats.



The operation code (opcode) part of the instruction contains three bits and the meaning of the remaining 13 bits depends on the operation code encountered.

A memory-reference instruction uses 12 bits to specify an address and one bit to specify the addressing mode I.

I is equal to 0 for direct address and to 1 for indirect address.

The register-reference instructions are recognized by the operation code 1.11 with a 0 in the leftmost bit (bit 15) of the instruction.

A register-reference instruction specifies an operation on the AC register. So an operand from memory is not needed. Therefore, the other 12 bits are used to specify the operation to be executed.

An input—output instruction does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction.

The remaining 12 bits are used to specify the type of input—output operation.

2. Timing and Control:

The timing for all registers in the basic computer is controlled by a master clock generator.

The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit.

The clock pulses do not change the state of a register unless the register is enabled by a control signal.

The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and microoperations for the accumulator.

There are two major types of control organization:

- *Hardwired control*
- *Microprogrammed control*

The differences between hardwired and microprogrammed control are

Hardwired control	**Microprogrammed control**
✔ The control logic is implemented with gates, flip-flops, decoders, and other digital circuits.	✔ The control information is stored in a control memory. The control memory is programmed to initiate the required sequence of microoperations.
✔ The advantage that it can be optimized to produce a fast mode of operation.	✔ Compared with the hardwired control operation is slow.
✔ Requires changes in the wiring among the various components if the design has to be modified or changed.	✔ Required changes or modifications can be done by updating the microprogram in control memory.

As an example, consider the case where *SC* is incremented to provide timing signals T_0, T_1, T_2, T_3 and T_4 in sequence. At time T_4, *SC* is cleared to 0 if decoder output D3 is active. This is expressed symbolically by the statement

D_3T_4: SC□0

The timing diagram of Fig. 5-7 shows the time relationship of the control signals.

The sequence counter *SC* responds to the positive transition of the clock.

Initially, the CLR input of *SC* is active. The first positive transition of the clock clears *SC* to 0, which in turn activates the timing signal T_0 out of the decoder. T_0 is active during one clock cycle.

SC is incremented with every positive clock transition, unless its CLR input is active.

This produces the sequence of timing signals T_0, T_1, T_2, T_3, $T_4$and so on, as shown in the diagram.

The last three waveforms in Fig.5-7 show how *SC* is cleared when $D_3T_4 = 1$.

Output D_3 from the operation decoder becomes active at the end of timing signal T_2.

When timing signal T_4 becomes active, the output of the AND gate that implements the control function D_3T_4 becomes active.

This signal is applied to the CLR input of *SC.* On the next positive clock transition (the one marked T4 in the diagram) the counter is cleared to 0.

This causes the timing signal T_0 to become active instead of T_5 that would have been active if *SC* were incremented instead of cleared.

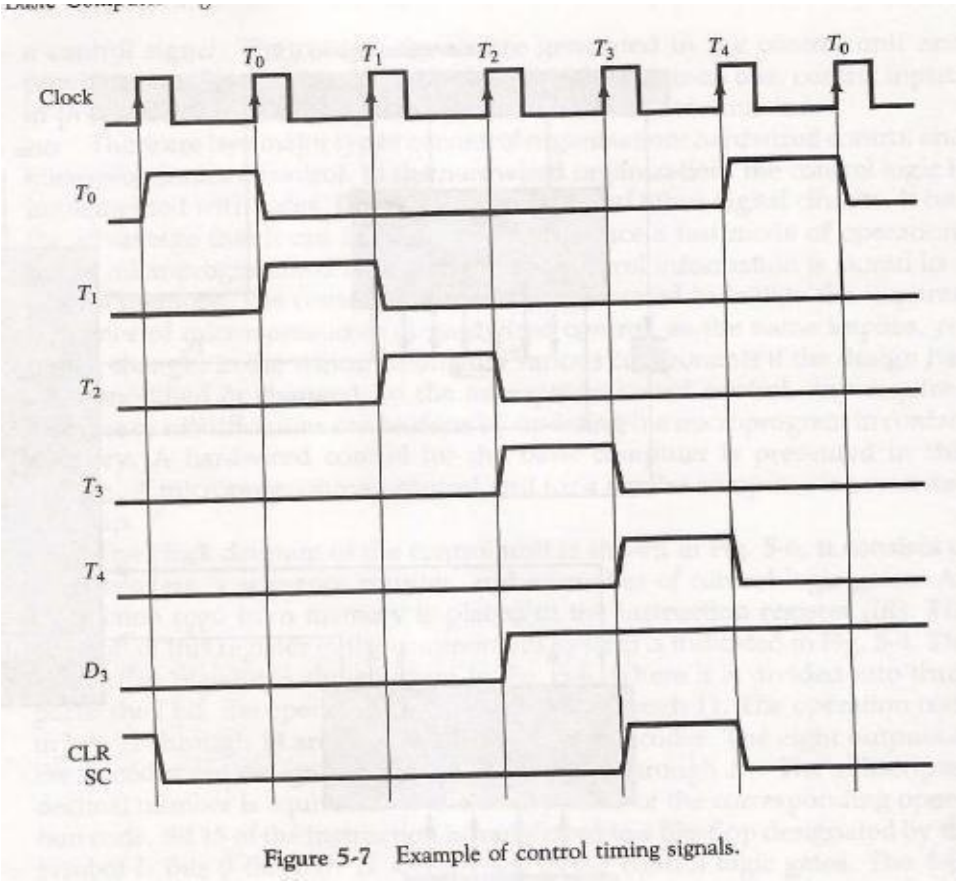


Figure 5-7 Example of control timing signals.

3. **Instruction Cycle**:

A program residing in the memory unit of the computer consists of a sequence of instructions.

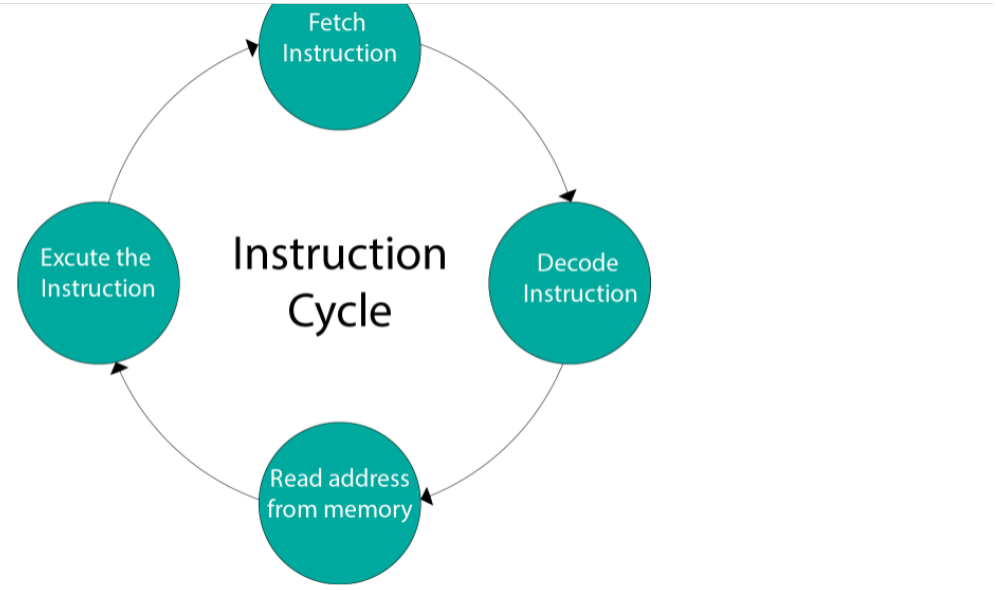
The program is executed in the computer by going through a cycle for each instruction.

Each instruction cycle in turn is subdivided into a sequence of sub cycles or phases.

In the basic computer each instruction cycle consists of the following phases:

1. Fetch an instruction from memory.
2. Decode the instruction.
3. Read the effective address from memory if the instruction has an indirect address.
4. Execute the instruction.

Upon the completion of step 4, the control goes back to step 1 to fetch, decode, and execute the next instruction.



Fetch and Decode:

Initially, the program counter PC is loaded with the address of the first instruction in the program.

The sequence counter *SC* is cleared to 0, providing a decoded timing signal T_0.

The microoperations for the fetch and decode phases can be specified by the following register transfer statements.

Determine the Type of Instruction:

The timing signal that is active after the decoding is T_3.

During time T_3, the control unit determines the type of instruction that was read from the memory.

The flowchart of fig.5-9 shows the initial configurations for the instruction cycle and also how the control determines the instruction cycle type after the decoding.

Decoder output D_7 is equal to 1 if the operation code is equal to binary 111.

If D_7=1, the instruction must be a register-reference or input-output type.

If *D7* = 0, the operation code must be one of the other seven values 000 through 110, specifying a memory-reference instruction.

Control then inspects the value of the first bit of the instruction, which is now available in flip-flop I.

If *D7* = 0 and I = 1, indicates a memory-reference instruction with an indirect address. So it is then necessary to read the effective address from memory.

If *D7* = 0 and I = 0, indicates a memory-reference instruction with a direct address.

If *D7* = 1 and I = 0, indicates a register-reference instruction.

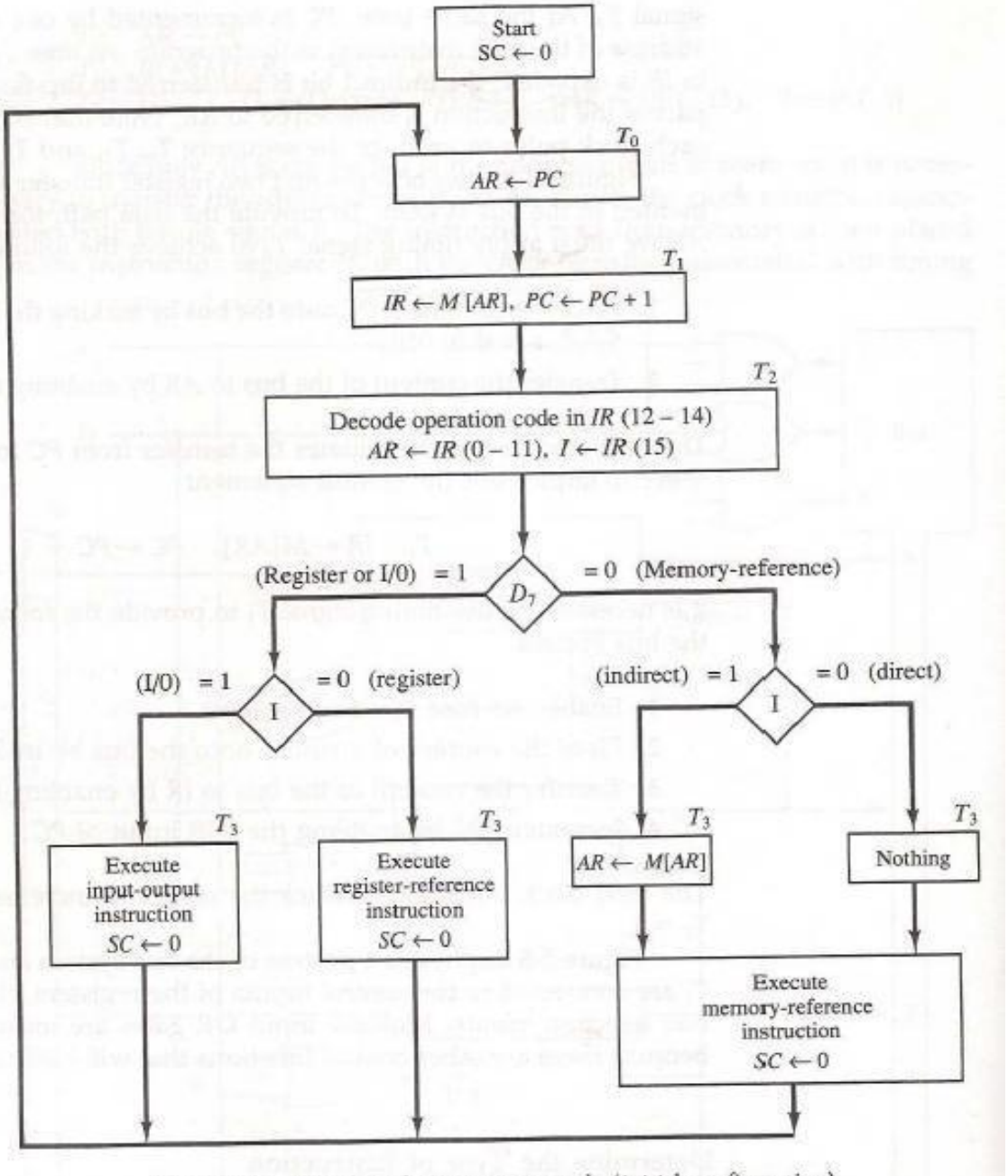
If *D7* = 01and I = 1, indicates an input-output instruction.

The three instruction types are subdivided into four separate paths.

The selected operation is activated with the clock transition associated with timing signal T_3.

This can be symbolized as follows:

$D_7' I T_3$: $AR \leftarrow M[AR]$
$D_7' I' T_3$: Nothing
$D_7 I' T_3$: Execute a register-reference instruction
$D_7 I T_3$: Execute an input–output instruction



Register-Reference Instructions:

Register-reference instructions are recognized by the control when D7 = 1 and I=0.

These instructions use bits 0 through 11 of the instruction code to specify one of 12 instructions.

These 12 bits are available in IR (0-11).

The control functions and microoperations for the register-reference instructions are listed in Table 5-3.

These instructions are executed with the clock transition associated with timing variable T_3.

Control function needs the Boolean relation $D_7I'T_3$, which we designate for convenience by the symbol r.
By assigning the symbol B_i to bit i of *IR,* all control functions can be simply denoted by rB_i.

TABLE 5-3 Execution of Register-Reference Instructions

$D_7I'T_3 = r$ (common to all register-reference instructions)
$IR(i) = B_i$ [bit in IR(0–11) that specifies the operation]

	r:	$SC \leftarrow 0$	Clear SC
CLA	rB_{11}:	$AC \leftarrow 0$	Clear AC
CLE	rB_{10}:	$E \leftarrow 0$	Clear E
CMA	rB_9:	$AC \leftarrow \overline{AC}$	Complement AC
CME	rB_8:	$E \leftarrow \overline{E}$	Complement E
CIR	rB_7:	$AC \leftarrow$ shr AC, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$	Circulate right
CIL	rB_6:	$AC \leftarrow$ shl AC, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$	Circulate left
INC	rB_5:	$AC \leftarrow AC + 1$	Increment AC
SPA	rB_4:	If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$	Skip if positive
SNA	rB_3:	If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$	Skip if negative
SZA	rB_2:	If $(AC = 0)$ then $PC \leftarrow PC + 1$	Skip if AC zero
SZE	rB_1:	If $(E = 0)$ then $(PC \leftarrow PC + 1)$	Skip if E zero
HLT	rB_0:	$S \leftarrow 0$ (S is a start–stop flip-flop)	Halt computer

4. Memory-Reference Instructions:

Table 5-4 lists the seven memory-reference instructions.

The decoded output D_i for i = 0, 1, 2, 3, 4, 5, and 6 from the operation decoder that belongs to each instruction is included in the table.

The effective address of the instruction is in the address register AR and was placed there during timing signal T_2 when I= 0, or during timing signal T_3 when I = 1.

The execution of the memory-reference instructions starts with timing signal T_4.

The symbolic description of each instruction is specified in the table in terms of register transfer notation.

TABLE 5-4 Memory-Reference Instructions

Operation