COMPUTER ORGANIZATION (15A05402)

LECTURE NOTES

B.TECH

(III-YEAR& I-SEM)

Prepared by:

Ms. M. Latha Reddy, Assistant Professor

Department of Computer Science and Engineering

VEMU INSTITUTE OF TECHNOLOGY

(Approved By AICTE, New Delhi and Affiliated to JNTUA, Ananthapuramu)
Accredited By NAAC & ISO: 9001-2015 Certified Institution
Near Pakala, P. Kothakota, Chittoor- Tirupathi Highway
Chittoor, Andhra Pradesh - 517 112
Web Site: www.vemu.org

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY ANANTAPUR

B. Tech III-I Sem. (ECE)

L	T	P	C
3	1	0	3

(15A05402) COMPUTER ORGANIZATION

C311_1: Explain the organization of computer, different instruction formats and addressing modes.

C311_2: Explain the functional units of the processor such as register file and ALU

C311_3: Explain the concepts of memory and I/O devices and virtual memory effectively

C311_4: Describe memory hierarchy and modes of data transfer.

C311_5: Implement the algorithm for exploring the pipelling and basic characteristics of multiplexing

UNIT-I: Computer types, Functional units, basic operational concepts, Bus structures, Data types, Software: Languages and Translators, oaders, Linkers, Operating systems. Memory locations – addresses and encoding of information – main memory operations– Instruction formats and instruction sequences – Addressing modes and instructions –Simple input programming – pushdown stacks – subroutines**.**

UNIT-II: Register transfer Language, Register transfer, Bus and Memory Transfers, Arithmetic Micro operations, Logic Micro operations, shift Micro operations, Arithmetic Logic Shift Unit. Stack organization, instruction formats, Addressing modes, Data transfer and manipulation, Execution of a complete instruction, Sequencing of control signals, Program Control.

UNIT-III: Control Memory, address Sequencing, Micro Program Example, Design of Control Unit. Addition and Subtraction, Multiplication Algorithms, Division Algorithms, Floating Point Arithmetic Operations, Decimal Arithmetic Unit, Decimal Arithmetic Operations.

UNIT-IV: Peripheral Devices, Input-Output Interface, Asynchronous Data Transfer, Modes of Transfer, Priority Interrupt, Direct Memory Access (DMA), Input-Output Processor (IOP), Serial Communication. Memory hierarchy, main memory, auxiliary memory, Associative memory, Cache memory, Virtual memory, Memory management hardware.

UNIT-V: Parallel Processing, Pipelining, Arithmetic Pipeline, Instruction Pipeline, RISC Pipeline Vector Processing, Array Processors. Characteristics of Multiprocessors, Interconnection Structures, Interprocessor Arbitration, Inter-processor Communication and Synchronization, Cache Coherence.

Text Books:

1. M. Morris Mano, "Computer system Architecture", Prentice Hall of India (PHI), Third edition.
2. William Stallings,"Computer organization and programming", Prentice Hall of India(PHI) Seventh Edition, Pearson Education(PE) Third edition, 2006.

Reference Books:

1. Carl Hamacher, ZvonksVranesic, SafwatZaky, "Computer Organization" 5thEdition, McGraw Hill, 2002.
2. Andrew S.Tanenbaum, "Structured Computer Organization", 4th Edition PHI/Pearson
3. John L.Hennessy and David A.Patterson, "Computer Architecture a quantitative approach", Fourth Edition Elsevier
4. joseph D.Dumas II, "Computer Architecture: Fundamentals and Principals of ComputerDesign", BS Publication.

UNIT – 1

BASIC STRUCTURE OF COMPUTERS

1.1. COMPUTER TYPES

A computer can be defined as a fast electronic calculating machine that accepts the (data) digitized input information process it as per the list of internally stored instructions and produces the resulting information.

List of instructions are called programs & internal storage is called computer memory.

The different types of computers are

1. **Personal computers: -** This is the most common type found in homes, schools, Business offices etc., It is the most common type of desk top computers with processing and storage units along with various input and output devices.
2. **Note book computers: -** These are compact and portable versions of PC
3. **Work stations: -** These have high resolution input/output (I/O) graphics capability, but with same dimensions as that of desktop computer. These are used in engineering applications of interactive design work.
4. **Enterprise systems: -** These are used for business data processing in medium to large corporations that require much more computing power and storage capacity than work stations. Internet associated with servers have become a dominant worldwide source of all types of information.
5. **Super computers: -** These are used for large scale numerical calculations required in the applications like weather forecasting etc.,

1.2 FUNCTIONAL UNIT

A computer consists of five functionally independent main parts input, memory, arithmetic logic unit (ALU), and output and control unit.

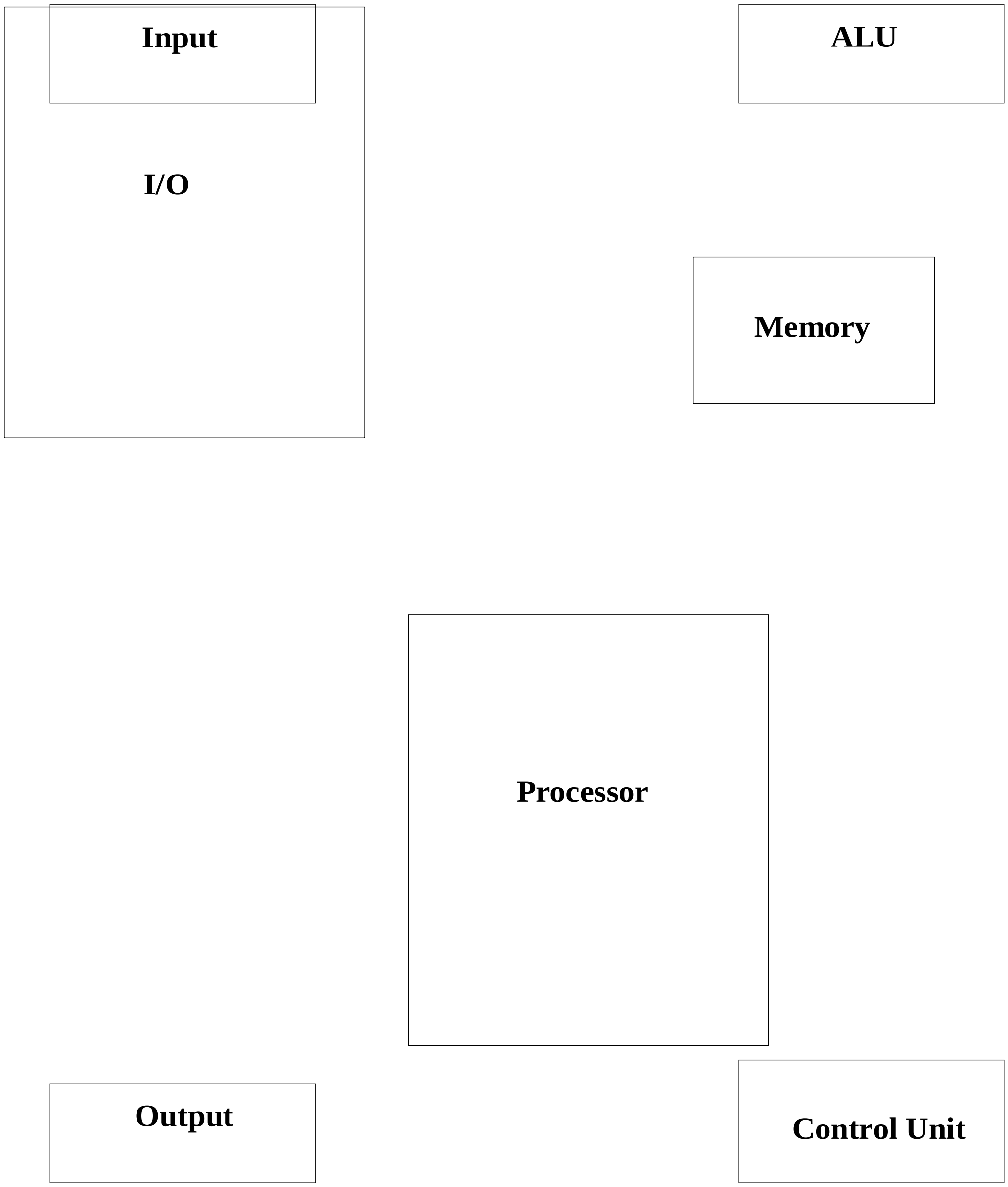


Fig a: Functional units of computer

Input device accepts the coded information as source program i.e. high level language. This is either stored in the memory or immediately used by the processor to perform the desired operations. The program stored in the memory determines the processing steps. Basically the computer converts one source program to an object program. i.e. into machine language.

Finally the results are sent to the outside world through output device. All of these actions are coordinated by the control unit.

Input unit: -

The source program/high level language program/coded information/simply data is fed to a computer through input devices keyboard is a most common type. Whenever a key is pressed, one corresponding word or number is translated into its equivalent binary code over a cable & fed either to memory or processor.

Joysticks, trackballs, mouse, scanners etc are other input devices.

Memory unit: -

Its function into store programs and data. It is basically to two types

1. **Primary memory**
2. **Secondary memory**

1. Primary memory: - Is the one exclusively associated with the processor and operates at the electronics speeds programs must be stored in this memory while they are being executed. The memory contains a large number of semiconductors storage cells. Each capable of storing one bit of information. These are processed in a group of fixed site called word.

To provide easy access to a word in memory, a distinct address is associated with each word location. **Addresses are** numbers that identify memory location.

Number of bits in each word is called word length of the computer. Programs must reside in the memory during execution. Instructions and data can be written into the memory or read out under the control of processor.

Memory in which any location can be reached in a short and fixed amount of time after specifying its address is called random-access memory (RAM).

The time required to access one word in called memory access time. Memory which is only readable by the user and contents of which can't be altered is called read only memory (ROM) it contains operating system.

Caches are the small fast RAM units, which are coupled with the processor and are aften contained on the same IC chip to achieve high performance. Although primary storage is essential it tends to be expensive.

2. Secondary memory: - Is used where large amounts of data & programs have to be stored, particularly information that is accessed infrequently.

Examples: - Magnetic disks & tapes, optical disks (ie CD-ROM's), floppies etc.,

Arithmetic logic unit (ALU):-

Most of the computer operators are executed in ALU of the processor like addition, subtraction, division, multiplication, etc. the operands are brought into the ALU from memory and stored in high speed storage elements called register. Then according to the instructions the operation is performed in the required sequence.

The control and the ALU are many times faster than other devices connected to a computer system. This enables a single processor to control a number of external devices such as key boards, displays, magnetic and optical disks, sensors and other mechanical controllers.

Output unit:-

These actually are the counterparts of input unit. Its basic function is to send the processed results to the outside world.

Examples:- Printer, speakers, monitor etc.

Control unit:-

It effectively is the nerve center that sends signals to other units and senses their states. The actual timing signals that govern the transfer of data between input unit, processor, memory and output unit are generated by the control unit.

1.3. BASIC OPERATIONAL CONCEPTS

To perform a given task an appropriate program consisting of a list of instructions is stored in the memory. Individual instructions are brought from the memory into the processor, which executes the specified operations. Data to be stored are also stored in the memory.

Examples: - Add LOCA, R_0

This instruction adds the operand at memory location LOCA, to operand in register R_0 & 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 R_0
3. Finally the resulting sum is stored in the register R_0

The preceding add instruction combines a memory access operation with an ALU Operations. In some other type of computers, these two types of operations are performed by separate instructions for performance reasons.

```
Load LOCA, R1
Add  R1, R0
```

Transfers between the memory and the processor are started by sending the address of the memory location to be accessed to the memory unit and issuing the appropriate control signals. The data are then transferred to or from the memory.

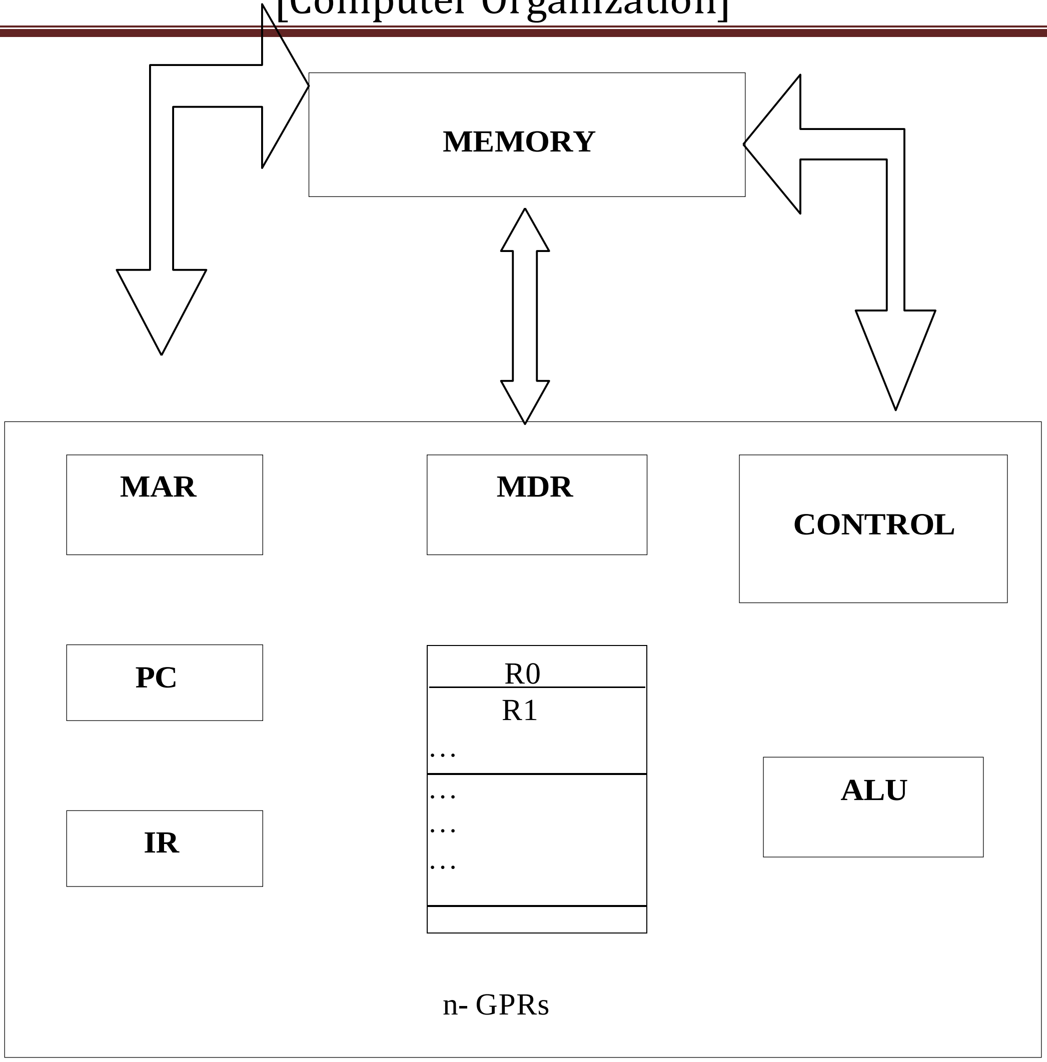


Fig b: Connections between the processor and the memory

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 R_{n-1}.

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.

Normal execution of a program may be preempted (temporarily interrupted) if some devices require urgent servicing, to do this one device raises an Interrupt signal.

An interrupt is a request signal from an I/O device for service by the processor. The processor provides the requested service by executing an appropriate interrupt service routine.

The Diversion may change the internal stage of the processor its state must be saved in the memory location before interruption. When the interrupt-routine service is completed the state of the processor is restored so that the interrupted program may continue.

1.4 BUS STRUCTURE

The simplest and most common way of interconnecting various parts of the computer. To achieve a reasonable speed of operation, a computer must be organized so that all its units can handle one full word of data at a given time.A group of lines that serve as a connecting port 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 purpose. Simplest way to interconnect is to use the single bus as shown

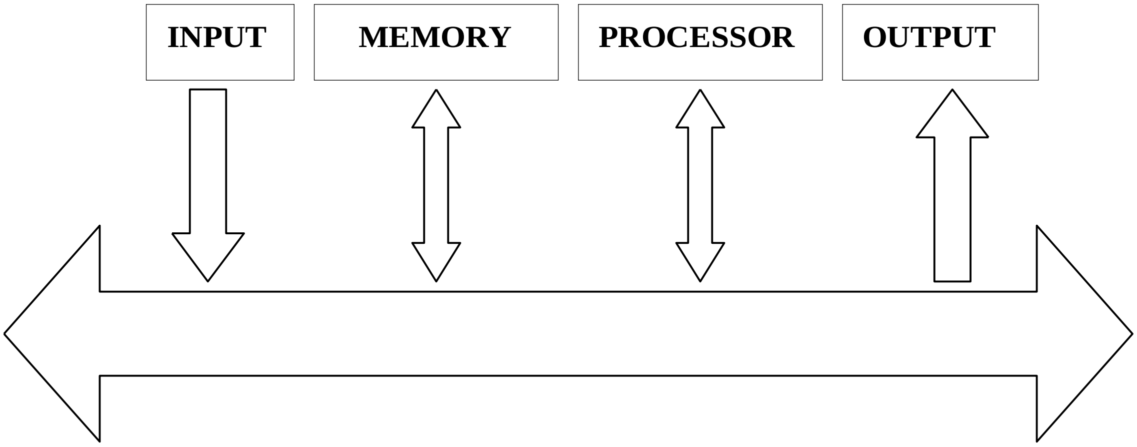


Fig c: Single bus structure

Since the bus can be used for only one transfer at a time, only two units can actively use the bus at any given time. Bus control lines are used to arbitrate multiple requests for use of one bus.

Single bus structure is

- Low cost
- Very flexible for attaching peripheral devices

Multiple bus structure certainly increases, the performance but also increases the cost significantly.

All the interconnected devices are not of same speed & time, leads to a bit of a problem. This is solved by using cache registers (ie buffer registers). These buffers are electronic registers of small capacity when compared to the main memory but of comparable speed.

The instructions from the processor at once are loaded into these buffers and then the complete transfer of data at a fast rate will take place.

1.5 PERFORMANCE

The most important measure of the performance of a computer is how quickly it can execute programs. The speed with which a computer executes program is affected by the design of its hardware. For best performance, it is necessary to design the compiles, the machine instruction set, and the hardware in a coordinated way.

The total time required to execute the program is elapsed time is a measure of the performance of the entire computer system. It is affected by the speed of the processor, the disk and the printer. The time needed to execute a instruction is called the processor time.

Just as the elapsed time for the execution of a program depends on all units in a computer system, the processor time depends on the hardware involved in the execution of individual machine instructions. This hardware comprises the processor and the memory which are usually connected by the bus as shown in the fig c.

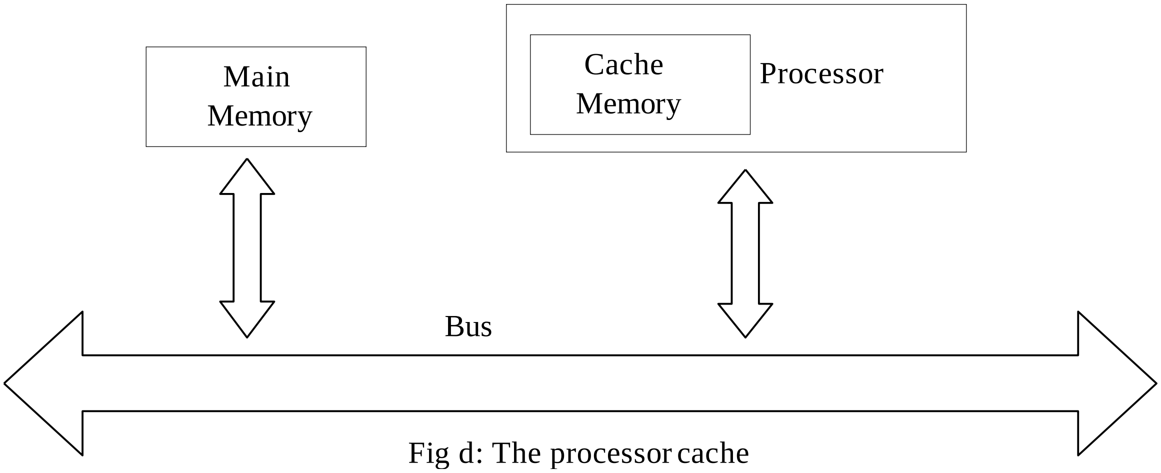


Fig d: The processor cache

The pertinent parts of the fig. c are repeated in fig. d which includes the cache memory as part of the processor unit.

Let us examine the flow of program instructions and data between the memory and the processor. At the start of execution, all program instructions and the required data are stored in the main memory. As the execution proceeds, instructions are fetched one by one over the bus into the processor, and a copy is placed in the cache later if the same instruction or data item is needed a second time, it is read directly from the cache.

The processor and relatively small cache memory can be fabricated on a single IC chip. The internal speed of performing the basic steps of instruction processing on chip is very high and is considerably faster than the speed at which the instruction and data can be fetched from the main memory. A program will be executed faster if the movement of instructions and data between the main memory and the processor is minimized, which is achieved by using the cache.

For example:- Suppose a number of instructions are executed repeatedly over a short period of time as happens in a program loop. If these instructions are available in the cache, they can be fetched quickly during the period of repeated use. The same applies to the data that are used repeatedly.

Application software

- Programs designed to perform specific tasks that are transparent to the user

System software

- Programs that support the execution and development of other programs
-

Two major types

- Operating systems
- Translation systems

1.1.8 Application Software

- Application software is the software that has made using computers indispensable and popular

Common application software

- Word processors
- Desktop publishing programs
- Spreadsheets
- Presentation managers
- Drawing programs

1.1.9 Operating System

Examples

- Windows®, UNIX®, Mac OS X®
- Controls and manages the computing resources
- Important services that an operating system provides
- □**File system**

✓ Directories, folders, files

- Commands that allow for manipulation of the file system

✓ Sort, delete, copy

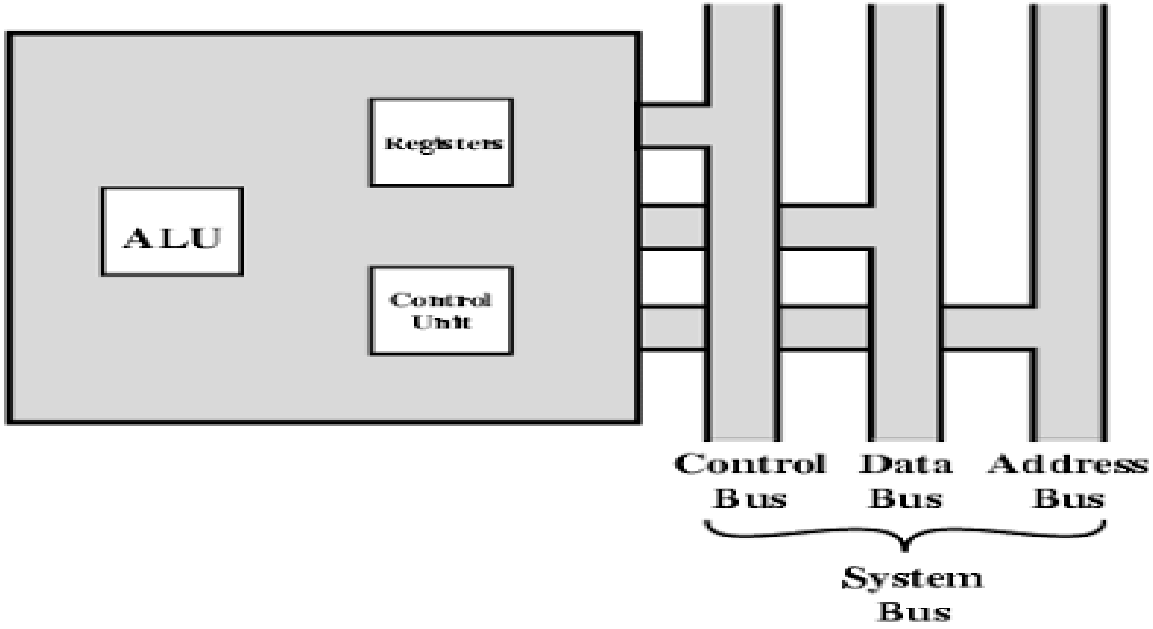
- Ability to perform input and output on a variety of devices
- Management of the running systems

1.2. CPU Organization

1.2.1 CPU Structure and Function Processor Organization

- Things a CPU must do:

  - Fetch Instructions
  - Interpret Instructions
  - Fetch Data
  - Process Data
  - Write Data



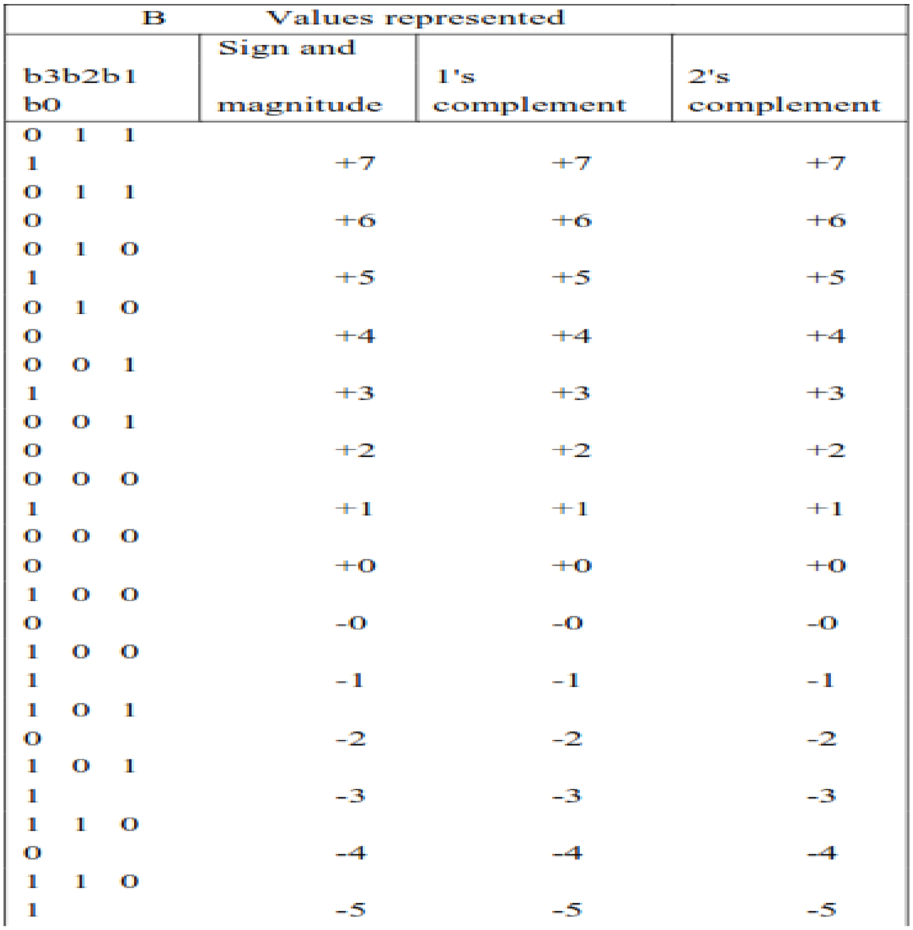
- A small amount of internal memory, called the registers, is needed by the CPU to fulfil these requirements

1.7 NUMBER REPRESENTATION

We obviously need to represent both positive and negative numbers. Three systems are used for representing such numbers :

- Sign-and-magnitude
- 1’s-complement
- 2’s-complement

In all three systems, the leftmost bit is 0 for positive numbers and 1 for negative numbers. Fig 2.1 illustrates all three representations using 4-bit numbers. Positive values have identical representations in al systems, but negative values have different representations. In the sign-and-magnitude systems, negative values are represented by changing the most significant bit (b_3 in figure 2.1) from 0 to 1 in the B vector of the corresponding positive value. For example, +5 is represented by 0101, and -5 is represented by 1101. In 1’s-complement representation, negative values are obtained by complementing each bit of the corresponding positive number. Thus, the representation for -3 is obtained by complementing each bit in the vector 0011 to yield 1100. clearly, the same operation, bit complementing, is done in converting a negative number to the corresponding positive value. Converting either way is referred to as forming the 1’s-complement of a given number. Finally, in the 2’s-complement system, forming the 2’s-complement of a number is done by subtracting that number from 2^n.



B	Values represented		
b3b2b1 b0	Sign and magnitude	1's complement	2's complement
0 1 1 1	+7	+7	+7
0 1 1 0	+6	+6	+6
0 1 0 1	+5	+5	+5
0 1 0 0	+4	+4	+4
0 0 1 1	+3	+3	+3
0 0 1 0	+2	+2	+2
0 0 0 1	+1	+1	+1
0 0 0 0	+0	+0	+0
1 0 0 0	-0	-0	-0
1 0 0 1	-1	-1	-1
1 0 1 0	-2	-2	-2
1 0 1 1	-3	-3	-3
1 1 0 0	-4	-4	-4
1 1 0 1	-5	-5	-5

Hence, the 2’s complement of a number is obtained by adding 1 to the 1’s complement of that number.

Addition of Positive numbers:-

Consider adding two 1-bit numbers. The results are shown in figure 2.2. Note that the sum of 1 and 1 requires the 2-bit vector 10 to represent the value 2. We say that the sum is 0 and the carry-out is 1. In order to add multiple-bit numbers, we use a method analogous to that used for manual computation with decimal numbers. We add bit pairs starting from the low-order (right) and of the bit vectors, propagating carries toward the high-order (left) end.

0	1	0	1
+ 0	+ 0	+ 1	+ 1
_____	_____	____	____
0	1	1	1 0

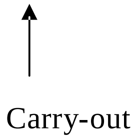


Figure 2.2 Addition of 1-bit numbers.

Memory locations and addresses

Number and character operands, as well as instructions, are stored in the memory of a computer. The memory consists of many millions of storage cells, each of which can store a bit of information having the value 0 or 1. Because a single bit represents a very small amount of information, bits are seldom handled individually. The usual approach is to deal with them in groups of fixed size. For this purpose, the memory is organized so that a group of n bits can be stored or retrieved in a single, basic operation. Each group of n bits is referred to as a word of information, and n is called the word length. The memory of a computer can be schematically represented as a collection of words as shown in figure (a).

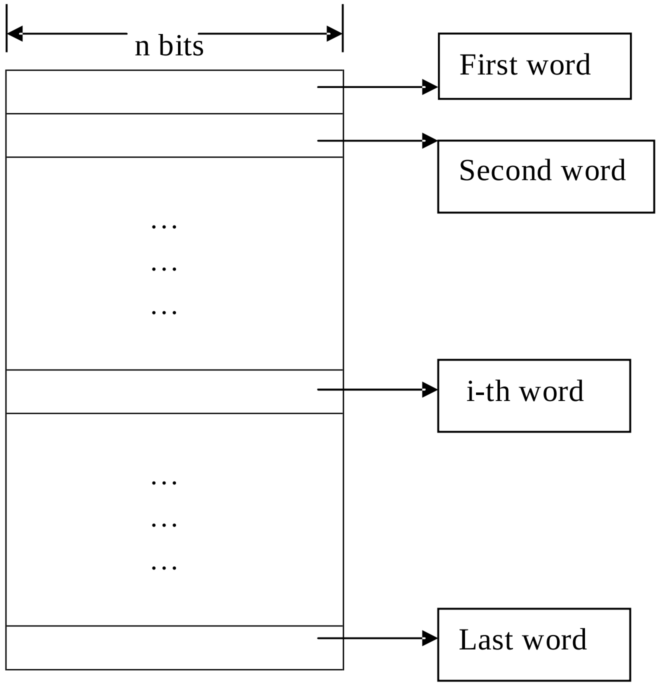
Modern computers have word lengths that typically range from 16 to 64 bits. If the word length of a computer is 32 bits, a single word can store a 32-bit 2’s complement number or four ASCII characters, each occupying 8 bits. A unit of 8 bits is called a byte.

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. It is customary to use numbers from 0 through 2^K-1, for some suitable values of k, as the addresses of successive locations in the memory. The 2^k addresses constitute the address space of the computer, and the memory can have up to 2^k addressable locations. 24-bit address generates an address space of 2^{24} (16,777,216) locations. A 32-bit address creates an address space of 2^{32} or 4G (4 giga) locations.

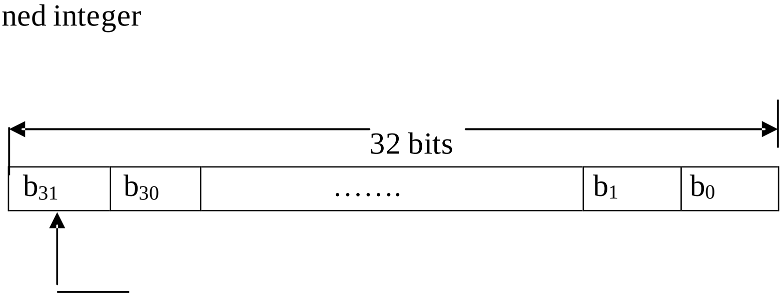
BYTE ADDRESSABILITY:-

We now have 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

Fig a Memory words



(a) A signed integer



Sign bit: b_{31} = 0 for positive numbers
b_{32} = 1 for negative numbers

(b) Four characters

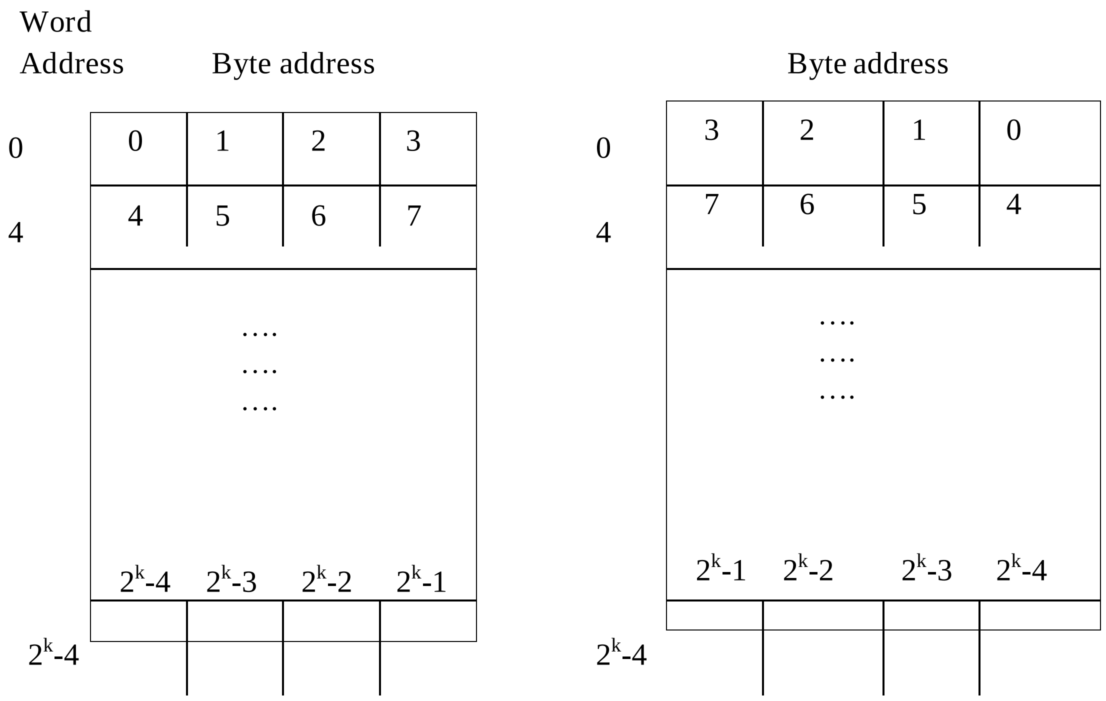
8 bits	8 bits	8 bits	8 bits
ASCII Character	ASCII Character	ASCII character	ASCII character

Locations in the memory. This is the assignment used in most modern computers, and is the one we will normally use in this book. The term byte-addressable memory is use for this assignment. Byte locations have addresses 0,1,2, …. Thus, if the word length of the machine is 32 bits, successive words are located at addresses 0,4,8,…., with each word consisting of four bytes.

BIG-ENDIAN AND LITTLE-ENDIAN ASIGNMENTS:-

There are two ways that byte addresses can be assigned across words, as shown in fig b. The name big-endian is used when lower byte addresses are used for the more significant bytes (the leftmost bytes) of the word. The name little-endian is used for the opposite ordering, where the lower byte addresses are used for the less significant bytes (the rightmost bytes) of the word.

In addition to specifying the address ordering of bytes within a word, it is also necessary to specify the labeling of bits within a byte or a word. The same ordering is also used for labeling bits within a byte, that is, b_7, b_6, …., b_0, from left to right.



(a) Big-endian assignment (b) Little-endianassignment

WORD ALIGNMENT:-

In the case of a 32-bit word length, natural word boundaries occur at addresses 0, 4, 8, …, as shown in above fig. We say that the word locations have aligned addresses . in general, words are said to be aligned in memory if they begin at a byte address that is a multiple of the number of bytes in a word. The memory of bytes in a word is a power of 2. Hence, if the word length is 16 (2 bytes), aligned words begin at byte addresses 0,2,4,…, and for a word length of 64 (23 bytes), aligned words begin at bytes addresses 0,8,16 ….

There is no fundamental reason why words cannot begin at an arbitrary byte address. In that case, words are said to have unaligned addresses. While the most common case is to use aligned addresses, some computers allow the use of unaligned word addresses.

ACCESSING NUMBERS, CHARACTERS, AND CHARACTER STRINGS:-

A number usually occupies one word. It can be accessed in the memory by specifying its word address. Similarly, individual characters can be accessed by their byte address.

In many applications, it is necessary to handle character strings of variable length. The beginning of the string is indicated by giving the address of the byte containing its first character. Successive byte locations contain successive characters of the string. There are two ways to indicate the length of the string. A special control character with the meaning “end of string” can be used as the last character in the string, or a separate memory word location or processor register can contain a number indicating the length of the string in bytes.

Memory operations

Both program instructions and data operands are stored in the memory. To execute an instruction, the processor control circuits must cause the word (or words) containing the instruction to be transferred from the memory to the processor. Operands and results must also be moved between the memory and the processor. Thus, two basic operations involving the memory are needed, namely, Load (or Read or Fetch) and Store (or Write).

The load operation transfers a copy of the contents of a specific memory location to the processor. The memory contents remain unchanged. To start a Load operation, the processor sends the address of the desired location to the memory and requests that its contents be read. The memory reads the data stored at that address and sends them to the processor.

The store operation transfers an item of information from the processor to a specific memory location, destroying the former contents of that location. The processor sends the address of the desired location to the memory, together with the data to be written into that location.

An information item of either one word or one byte can be transferred between the processor and the memory in a single operation. Actually this transfer in between the CPU register & main memory.

1.8 INSTRUCTIONS AND INSTRUCTION SEQUENCING

A computer must have instructions capable of performing four types of operations.

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

REGISTER TRANSFER NOTATION:-

Transfer of information from one location in the computer to another. Possible locations that may be involved in such transfers are memory locations that may be involved in such transfers are memory locations, processor registers, or registers in the I/O subsystem. Most of the time, we identify a location by a symbolic name standing for its hardware binary address.

Example, names for the addresses of memory locations may be LOC, PLACE, A, VAR2; processor registers names may be R0, R5; and I/O register names may be DATAIN, OUTSTATUS, and so on. The contents of a location are denoted by placing square brackets around the name of the location. Thus, the expression

R1 � [LOC]

Means that the contents of memory location LOC are transferred into processor register R1.

As another example, consider the operation that adds the contents of registers R1 and R2, and then places their sum into register R3. This action is indicated as

R3 � [R1] + [R2]

This type of notation is known as Register Transfer Notation (RTN). Note that the right-hand side of an RTN expression always denotes a value, and the left-hand side is the name of a location where the value is to be places, overwriting the old contents of that location.

ASSEMBLY LANGUAGE NOTATION:-

Another type of notation to represent machine instructions and programs. For this, we use an assembly language format. For example, an instruction that causes the transfer described above, from memory location LOC to processor register R1, is specified by the statement

Move LOC, R1

The contents of LOC are unchanged by the execution of this instruction, but the old contents of register R1 are overwritten.

The second example of adding two numbers contained in processor registers R1 and R2 and placing their sum in R3 can be specified by the assembly language statement

Add R1, R2, R3

BASIC INSTRUCTIONS:-

The operation of adding two numbers is a fundamental capability in any computer. The statement

C = A + B

In a high-level language program is a command to the computer to add the current values of the two variables called A and B, and to assign the sum to a third variable, C. When the program containing this statement is compiled, the three variables, A, B, and C, are assigned to distinct locations in the memory. We will use the variable names to refer to the corresponding memory location addresses. The contents of these locations represent the values of the three variables. Hence, the above high-level language statement requires the action.

C � [A] + [B]

To carry out this action, the contents of memory locations A and B are fetched from the memory and transferred into the processor where their sum is computed. This result is then sent back to the memory and stored in location C.

Let us first assume that this action is to be accomplished by a single machine instruction. Furthermore, assume that this instruction contains the memory addresses of the three operands – A, B, and C. This three-address instruction can be represented symbolically as

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. A general instruction of this type has the format.

Operation Source1, Source 2, Destination

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

An alternative approach is to use a sequence of simpler instructions to perform the same task, with each instruction having only one or two operands. Suppose that two-address instructions of the form

Operation Source, Destination

Are available. An Add instruction of this type is

Add A, B

Which performs the operation B ← [A] + [B].

A single two-address instruction cannot be used to solve our original problem, which is to add the contents of locations A and B, without destroying either of them, and to place the sum in location C. The problem can be solved by using another two-address instruction that copies the contents of one memory location into another. Such an instruction is

Move B, C

Which performs the operations C ← [B], leaving the contents of location B unchanged.

Using only one-address instructions, the operation C ← [A] + [B] can be performed by executing the sequence of instructions

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

Some early computers were designed around a single accumulator structure.

Most modern computers have a number of general-purpose processor registers – typically

8 to 32, and even considerably more in some cases. Access to data in these registers is much faster than to data stored in memory locations because the registers are inside the processor.

Let R_i represent a general-purpose register. The instructions

Load A, R_i
Store R_i, A
and
Add A, R_i

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

When a processor has several general-purpose registers, many instructions involve only operands that are in the register. In fact, in many modern processors, computations can be performed directly only on data held in processor registers. Instructions such as

Add Ri, Rj
Or
Add Ri, Rj, Rk

In both of these instructions, the source operands are the contents of registers Ri and Rj. In the first instruction, Rj also serves as the destination register, whereas in the second instruction, a third register, Rk, is used as the destination.

It is often necessary to transfer data between different locations. This is achieved with the instruction

Move Source, Destination

When data are moved to or from a processor register, the Move instruction can be used rather than the Load or Store instructions because the order of the source and destination operands determines which operation is intended. Thus,

Move A, Ri

Is the same as

Load A, Ri

And

Move Ri, A

Is the same as

Store Ri, A

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

```
Move   A, Ri
Move   B, Rj

Add    Ri, Rj
Move   Rj, C
```

In processors where one operand may be in the memory but the other must be in register, an instruction sequence for the required task would be

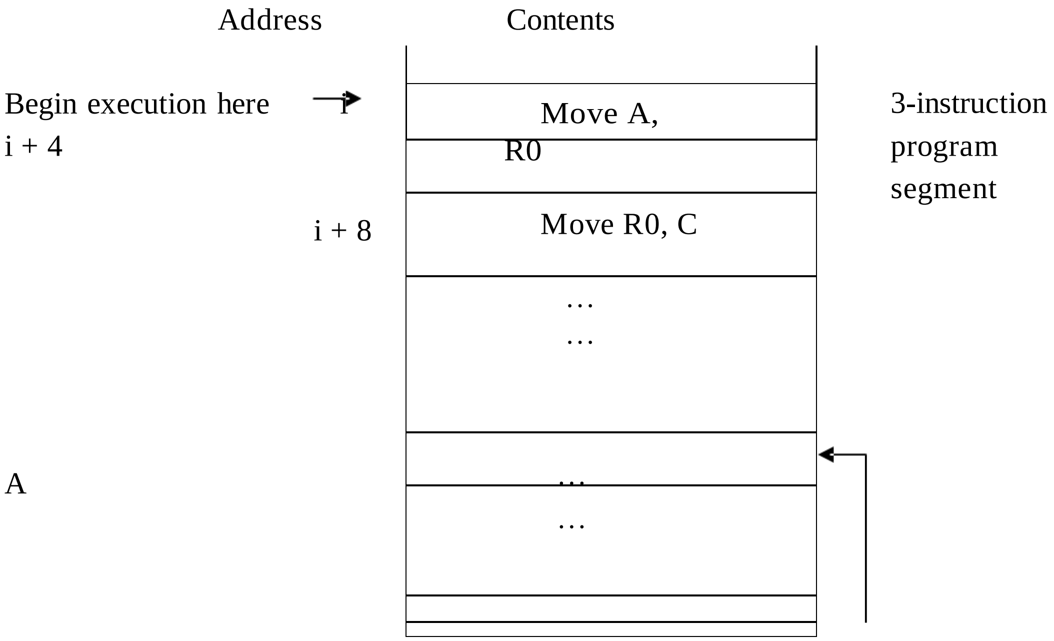
```
Move  A,  Ri
Add   B,   Ri
Move  Ri, C
```

The speed with which a given task is carried out depends on the time it takes to transfer instructions from memory into the processor and to access the operands referenced by these instructions. Transfers that involve the memory are much slower than transfers within the processor.

We have discussed three-, two-, and one-address instructions. It is also possible to use instructions in which the locations of all operands are defined implicitly. Such instructions are found in machines that store operands in a structure called a pushdown stack. In this case, the instructions are called zero-address instructions.

INSTRUCTION EXECUTION AND STRAIGHT-LINE SEQUENCING:-

In the preceding discussion of instruction formats, we used to task C � [A] + [B]. fig 2.8 shows a possible program segment for this task as it appears in the memory of a computer. We have assumed that the computer allows one memory operand per instruction and has a number of processor registers. The three instructions of the program are in successive word locations, starting at location i. since each instruction is 4 bytes long, the second and third instructions start at addresses i + 4 and i + 8.

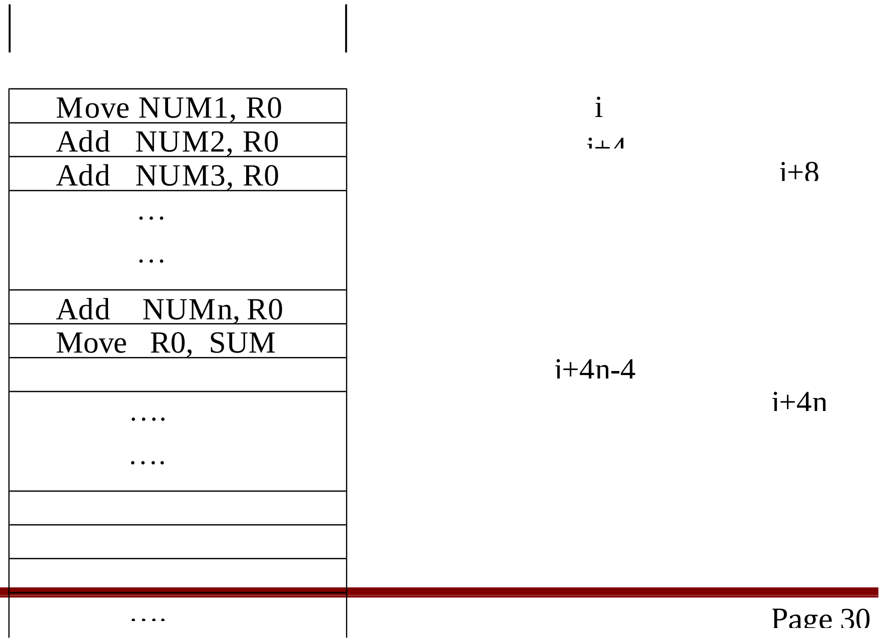


Let us consider how this program is executed. The processor contains a register called the program counter (PC), which holds the address of the instruction to be executed next. To begin executing a program, the address of its first instruction (I in our example) must be placed into the PC. Then, the processor control circuits use the information in the PC to fetch and execute instructions, one at a time, in the order of increasing addresses. This is called straight-line sequencing. During the execution of each instruction, the PC is incremented by 4 to point to the next instruction. Thus, after the Move instruction at location i + 8 is executed, the PC contains the value i + 12, which is the address of the first instruction of the next program segment.

Executing a given instruction is a two-phase procedure. In the first phase, called instruction fetch, the instruction is fetched from the memory location whose address is in the PC. This instruction is placed in the instruction register (IR) in the processor. The instruction in IR is examined to determine which operation is to be performed. The specified operation is then performed by the processor. This often involves fetching operands from the memory or from processor registers, performing an arithmetic or logic operation, and storing the result in the destination location.

BRANCHING:-

Consider the task of adding a list of n numbers. Instead of using a long list of add instructions, it is possible to place a single add instruction in a program loop, as shown in fig b. The loop is a straight-line sequence of instructions executed as many times as needed. It starts at location LOOP and ends at the instruction Branch > 0. During each pass through this loop, the address of the next list entry is determined, and that entry is fetched and added to



```
Move NUM1, R0
Add   NUM2, R0
Add   NUM3, R0
        …
        …
Add    NUMn, R0
Move   R0,  SUM

        ….
        ….



        ….
```

fig a A straight-line program for adding n numbers

	Move	N, R1
	Clear	R0
LOOP	Determine address of “Next” number and add “Next” number to R0	
Program loop	Decrement	R1
	Branch >0	LOOP
	Move	R0, SUM
	……. ……. …….	
	n	
	…… …...	

SUM N

NUM1 NUM2

NUMn

Fig b Using a loop to add n numbers

Assume that the number of entries in the list, n, is stored in memory location N, as shown. Register R1 is used as a counter to determine the number of time the loop is executed. Hence, the contents of location N are loaded into 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 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, called the branch target, instead of the instruction at the location that follows the branch instruction in sequential address order. 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.

Branch > 0 LOOP

(branch if greater that 0) is a conditional branch instruction that causes a branch to location LOOP if the result of the immediately preceding instruction, which is the decremented value in register R1, is greater that zero. This means that the loop is repeated, as long as there are entries in the list that are yet to be added to R0. at the end of the nth pass through the loop, the Decrement instruction produces a value of zero, and hence, branching does not occur.

CONDITION CODES:-

The processor keeps track of information about the results of various operations for use by subsequent conditional branch instructions. This is accomplished by recording the required information in individual bits, often called condition code flags. These flags are usually grouped together in a special processor register called the condition code register or status register. Individual condition code flags are set to 1 or cleared to 0, depending on the outcome of the operation performed.

Four commonly used flags are

N(negative)	Set to 1 if the result is negative; otherwise, cleared to 0
Z(zero)	Set to 1 if the result is 0; otherwise, cleared to 0
V(overflow)	Set ot1 if arithmetic overflow occurs; otherwise, cleared to 0
C(carry)	Set to 1 if a carry-out results from the operation; otherwise, cleared to 0

The instruction Branch > 0, discussed in the previous section, is an example of a branch instruction that tests one or more of the condition flags. It causes a branch if the value tested is neither negative nor equal to zero. That is, the branch is taken if neither N nor Z is 1. The conditions are given as logic expressions involving the condition code flags.

In some computers, the condition code flags are affected automatically by instructions that perform arithmetic or logic operations. However, this is not always the case. A number of computers have two versions of an Add instruction.

GENERATING MEMORY ADDRESSES:-

Let us return to fig b. The purpose of the instruction block at LOOP is to add a different number from the list during each pass through the loop. Hence, the Add instruction in the block must refer to a different address during each pass. How are the

addresses to be specified ? The memory operand address cannot be given directly in a single Add instruction in the loop. Otherwise, it would need to be modified on each pass through the loop.

The instruction set of a computer typically provides a number of such methods, called addressing modes. While the details differ from one computer to another, the underlying concepts are the same.

1.9 ADDRESSING MODES

In general, a program operates on data that reside in the computer's memory. These data can be organized in a variety of ways. If we want to keep track of students' names, we can write them in a list. Programmers use organizations called data structures to represent the data used in computations. These include lists, linked lists, arrays, queues, and so on.

Programs are normally written in a high-level language, which enables the programmer to use constants, local and global variables, pointers, and arrays. The different ways in which the location of an operand is specified in an instruction are referred to as addressing modes.

Table 2.1 Generic addressing modes

Name	Assembler syntax	Addressing function
Immediate	# Value	Operand = Value
Register	Ri	EA = Ri
Absolute (Direct)	LOC	EA = LOC
Indirect	(Ri)	EA = [Ri]
	(LOC)	EA = [LOC]
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	X(PC)	EA = [PC] + X
Autoincrement	(Ri)+	EA = [Ri]; Increment Ri
Autodecrement	-(Ri)	Decrement Ri; EA = [Ri]

EA = effective address
Value = a signed number

IMPLEMENTATION OF VARIABLE AND CONSTANTS:-

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

Register mode - The operand is the contents of a processor register; the name (address) of the register is given in the instruction.

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

The instruction

Move LOC, R2

Processor registers are used as temporary storage locations where the data is a register are accessed using the Register mode. The Absolute mode can represent global variables in a program. A declaration such as

Integer A, B;

Immediate mode – The operand is given explicitly in the instruction.
For example, the instruction

Move $200_{immediate}$, R0

Places the value 200 in register R0. Clearly, the Immediate mode is only used to specify the value of a source operand. Using a subscript to denote the Immediate mode is not appropriate in assembly languages. A common convention is to use the sharp sign (#) in front of the value to indicate that this value is to be used as an immediate operand. Hence, we write the instruction above in the form

Move #200, R0

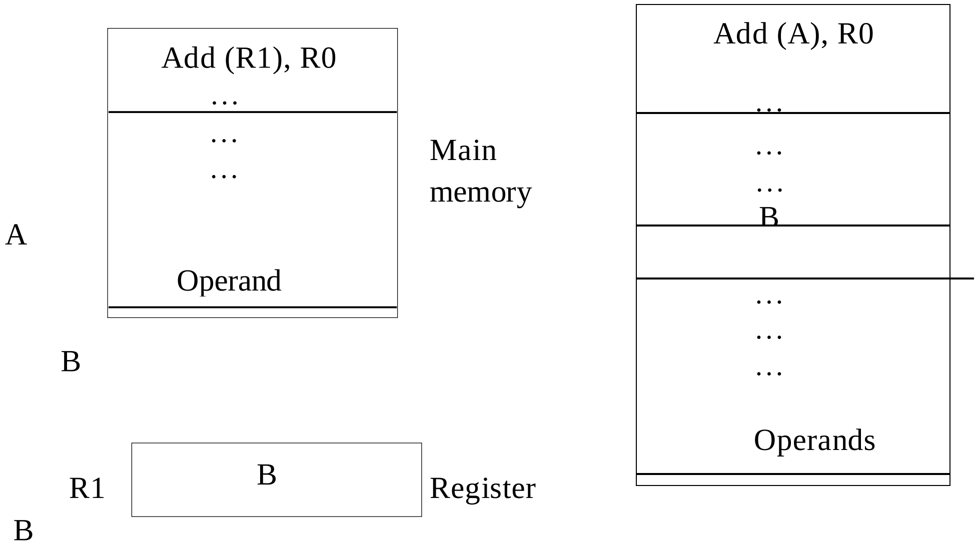
INDIRECTION AND POINTERS:-

In the addressing modes that follow, the instruction does not give the operand or its address explicitly, Instead, it provides information from which the memory address of the operand can be determined. We refer to this address as the effective address (EA) of the operand.

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

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

Fig (a) Through a general-purpose register (b) Through a memory location



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

The register or memory location that contains the address of an operand is called a pointer. Indirection and the use of pointers are important and powerful concepts in programming.

In the program shown Register R2 is used as a pointer to the numbers in the list, and the operands are accessed indirectly through R2. The initialization section of the program loads the counter value n from memory location N into R1 and uses the immediate addressing mode to place the address value NUM1, which is the address of the first number in the list, into R2. Then it clears R0 to 0. The first two instructions in the loop implement the unspecified instruction block starting at LOOP. The first time through the loop, the instruction **Add (R2), R0** fetches the operand at location NUM1 and adds it to R0. The second Add instruction adds 4 to the contents of the pointer R2, so that it will contain the address value NUM2 when the above instruction is executed in the second pass through the loop.

Where B is a pointer variable. This statement may be compiled into

```
Move    B,     R1
 Move   (R1), A
```

Using indirect addressing through memory, the same action can be achieved with

```
Move   (B), A
```

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

INDEXING AND ARRAYS:-

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

Index mode – the effective address of the operand is generated by adding a constant value to the contents of a register.

The register use may be either a special register provided for this purpose, or, more commonly, it may be any one of a set of general-purpose registers in the processor. In either case, it is referred to as index register. We indicate the Index mode symbolically as

X (Ri)

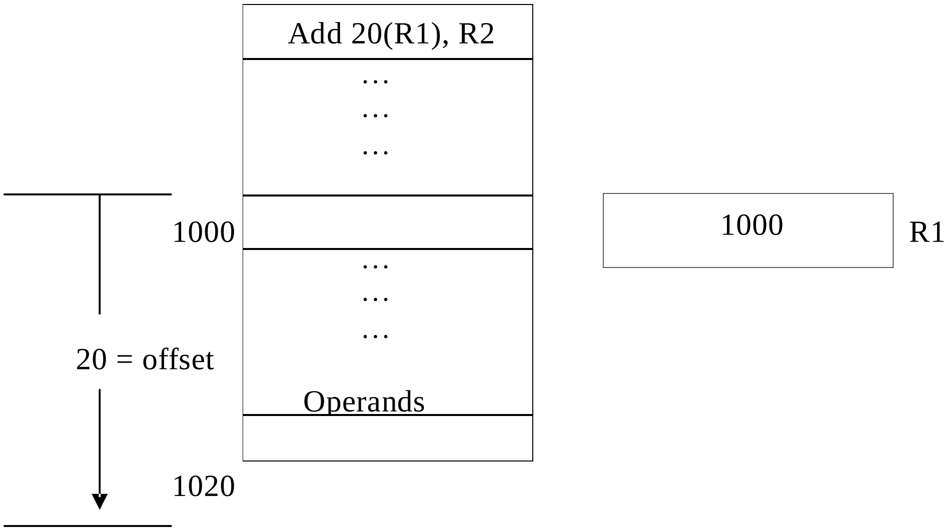
Where X denotes the constant value contained in the instruction and Ri is the name of the register involved. The effective address of the operand is given by

EA = X + [Rj]

The contents of the index register are not changed in the process of generating the effective address. In an assembly language program, the constant X may be given either as an explicit number or as a symbolic name representing a numerical value.

Fig a illustrates two ways of using the Index mode. In fig a, the index register, R1, contains the address of a memory location, and the value X defines an offset (also called a displacement) from this address to the location where the operand is found. An alternative use is illustrated in fig b. Here, the constant X corresponds to a memory address, and the contents of the index register define the offset to the operand. In either case, the effective address is the sum of two values; one is given explicitly in the instruction, and the other is stored in a register.

Fig (a) Offset is given as a constant



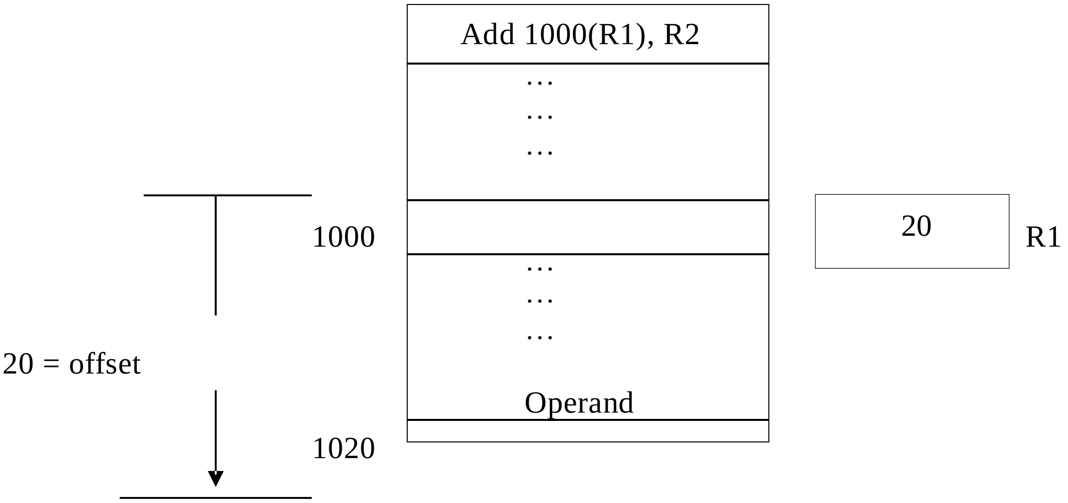


Fig (b) Offset is in the index register

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

In the most basic form of indexed addressing several variations of this basic form provide a very efficient access to memory operands in practical programming situations. For example, a second register may be used to contain the offset X, in which case we can write the Index mode as

(Ri, Rj)

The effective address is the sum of the contents of registers Ri and Rj. The second register is usually called the base register. This form of indexed addressing provides more flexibility in accessing operands, because both components of the effective address can be changed.

Another version of the Index mode uses two registers plus a constant, which can be denoted as

X(Ri, Rj)

In this case, the effective address is the sum of the constant X and the contents of registers Ri and Rj. This added flexibility is useful in accessing multiple components inside each item in a record, where the beginning of an item is specified by the (Ri, Rj) part of the addressing mode. In other words, this mode implements a three-dimensional array.

RELATIVE ADDRESSING:-

We have defined the Index mode using general-purpose processor registers. A useful version of this mode is obtained if the program counter, PC, is used instead of a general purpose register. Then, X(PC) can be used to address a memory location that is X bytes away from the location presently pointed to by the program counter.

Relative mode – The effective address is determined by the Index mode using the program counter in place of the general-purpose register Ri.

This mode can be used to access data operands. But, its most common use is to specify the target address in branch instructions. An instruction such as

Branch > 0 LOOP

Causes program execution to go to the branch target location identified by the name LOOP if the branch condition is satisfied. This location can be computed by specifying it as an offset from the current value of the program counter. Since the branch target may be either before or after the branch instruction, the offset is given as a signed number.

Autoincrement mode – the effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are automatically to point to the next item in a list.

(Ri)+

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

-(Ri)

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

Fig c The Autoincrement addressing mode used in the program of fig 2.12

ASSEMBLYLANGUAGE

Machine instructions are represented by patterns of 0s and 1s. Such patterns are awkward to deal with when discussing or preparing programs. Therefore, we use symbolic names to represent the pattern. So far, we have used normal words, such as Move, Add, Increment, and Branch, for the instruction operations to represent the corresponding binary code patterns. When writing programs for a specific computer, such words are normally replaced by acronyms called mnemonics, such as MOV, ADD, INC, and BR. Similarly, we use the notation R3 to refer to register 3, and LOC to refer to a memory location. A complete set of such symbolic names and rules for their use constitute a programming language, generally referred to as an assembly language.

Programs written in an assembly language can be automatically translated into a sequence of machine instructions by a program called an assembler. When the assembler program is executed, it reads the user program, analyzes it, and then generates the desired machine language program. The latter contains patterns of 0s and 1s specifying instructions that will be executed by the computer. The user program in its original alphanumeric text format is called a source program, and the assembled machine language program is called an object program.

ASSEMBLER DIRECTIVES:-

In addition to providing a mechanism for representing instructions in a program, the assembly language allows the programmer to specify other information needed to translate the source program into the object program. We have already mentioned that we need to assign numerical values to any names used in a program. Suppose that the name SUM is used to represent the value 200. This fact may be conveyed to the assembler program through a statement such as

```
SUM EQU 200
```

This statement does not denote an instruction that will be executed when the object program is run; in fact, it will not even appear in the object program. It simply informs the assembler that the name SUM should be replaced by the value 200 wherever it appears in the program. Such statements, called assembler directives (or commands), are used by the assembler while it translates a source program into an object program.

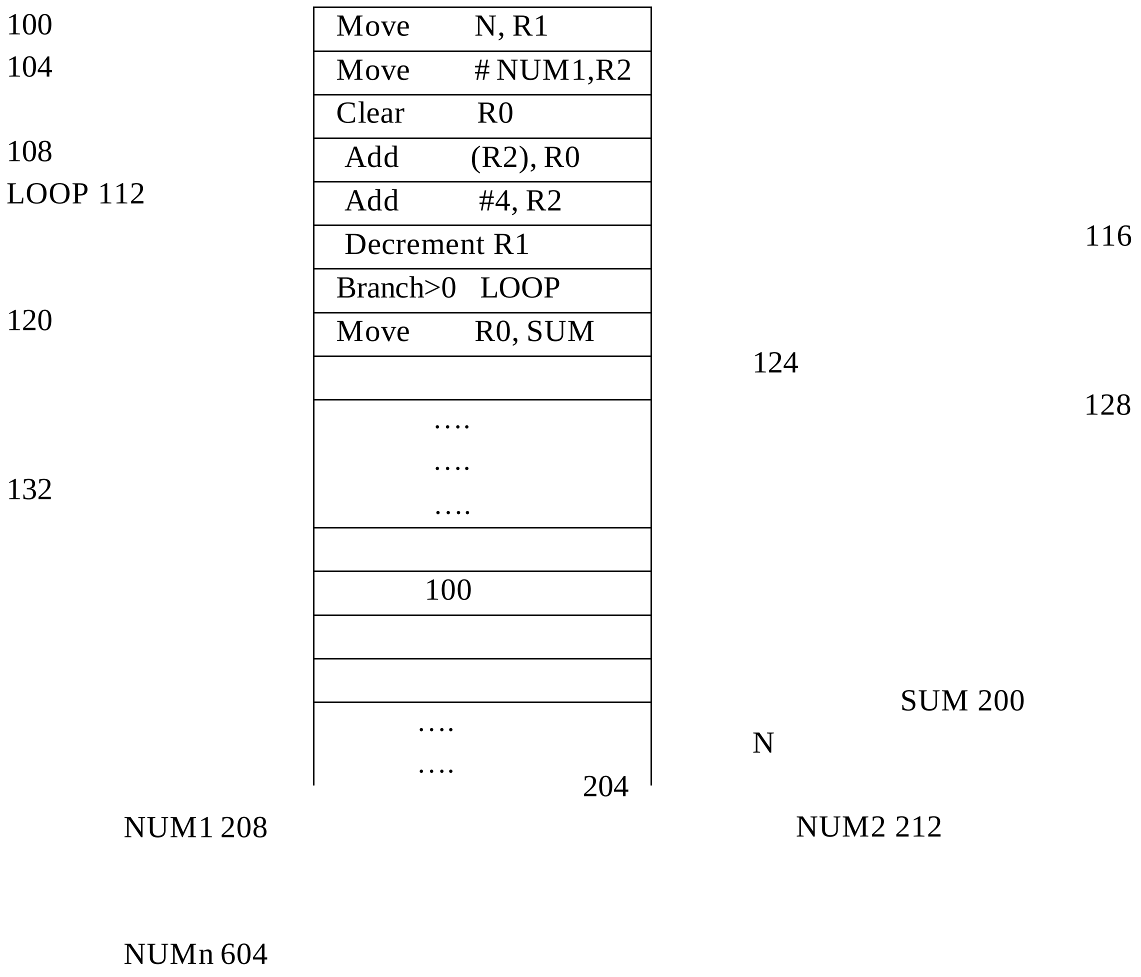

100
104
108
LOOP 112
116
120
124
128
132
Move N, R1
Move # NUM1,R2
Clear R0
Add (R2), R0
Add #4, R2
Decrement R1
Branch>0 LOOP
Move R0, SUM
….
….
….
100
….
….
SUM 200
N
204
NUM1 208
NUM2 212
NUMn 604

Fig 2.17 Memory arrangement for the program in fig b.

ASSEMBLY AND EXECUTION OF PRGRAMS:-

A source program written in an assembly language must be assembled into a machine language object program before it can be executed. This is done by the assembler program, which replaces all symbols denoting operations and addressing modes with the binary codes used in machine instructions, and replaces all names and labels with their actual values.

The assembler assigns addresses to instructions and data blocks, starting at the address given in the ORIGIN assembler directives. It also inserts constants that may be given in DATAWORD commands and reserves memory space as requested by RESERVE commands.

As the assembler scans through a source programs, it keeps track of all names and the numerical values that correspond to them in a symbol table. Thus, when a name appears a second time, it is replaced with its value from the table. A problem arises when

a name appears as an operand before it is given a value. For example, this happens if a forward branch is required. A simple solution to this problem is to have the assembler scan through the source program twice. During the first pass, it creates a complete symbol table. At the end of this pass, all names will have been assigned numerical values. The assembler then goes through the source program a second time and substitutes values for all names from the symbol table. Such an assembler is called a two-pass assembler.

The assembler stores the object program on a magnetic disk. The object program must be loaded into the memory of the computer before it is executed. For this to happen, another utility program called a loader must already be in the memory.

When the object program begins executing, it proceeds to completion unless there are logical errors in the program. The user must be able to find errors easily. The assembler can detect and report syntax errors. To help the user find other programming errors, the system software usually includes a debugger program. This program enables the user to stop execution of the object program at some points of interest and to examine the contents of various processor registers and memory locations.

NUMBER NOTATION:-

When dealing with numerical values, it is often convenient to use the familiar decimal notation. Of course, these values are stored in the computer as binary numbers. In some situations, it is more convenient to specify the binary patterns directly. Most assemblers allow numerical values to be specified in different ways, using conventions that are defined by the assembly language syntax. Consider, for example, the number 93, which is represented by the 8-bit binary number 01011101. If this value is to be used an immediate operand, it can be given as a decimal number, as in the instructions.

```
ADD   #93, R1
```

Or as a binary number identified by a prefix symbol such as a percent sign, as in

```
ADD   #%01011101, R1
```

Binary numbers can be written more compactly as hexadecimal, or hex, numbers, in which four bits are represented by a single hex digit. In hexadecimal representation, the decimal value 93 becomes 5D. In assembly language, a hex representation is often identified by a dollar sign prefix. Thus, we would write

```
ADD   #$5D, R1
```

1.10 simple input programming

We now examine the means by which data are transferred between the memory of a computer and the outside world. Input/Output (I/O) operations are essential, and the way they are performed can have a significant effect on the performance of the computer.

Consider a task that reads in character input from a keyboard and produces character output on a display screen. A simple way of performing such I/O tasks is to use a method known as program-controlled I/O. The rate of data transfer from the keyboard to a computer is limited by the typing speed of the user, which is unlikely to exceed a few characters per second. The rate of output transfers from the computer to the display is much higher. It is determined by the rate at which characters can be transmitted over the link between the computer and the display device, typically several thousand characters per second. However, this is still much slower than the speed of a processor that can execute many millions of instructions per second. The difference in speed between the processor and I/O devices creates the need for mechanisms to synchronize the transfer of data between them.

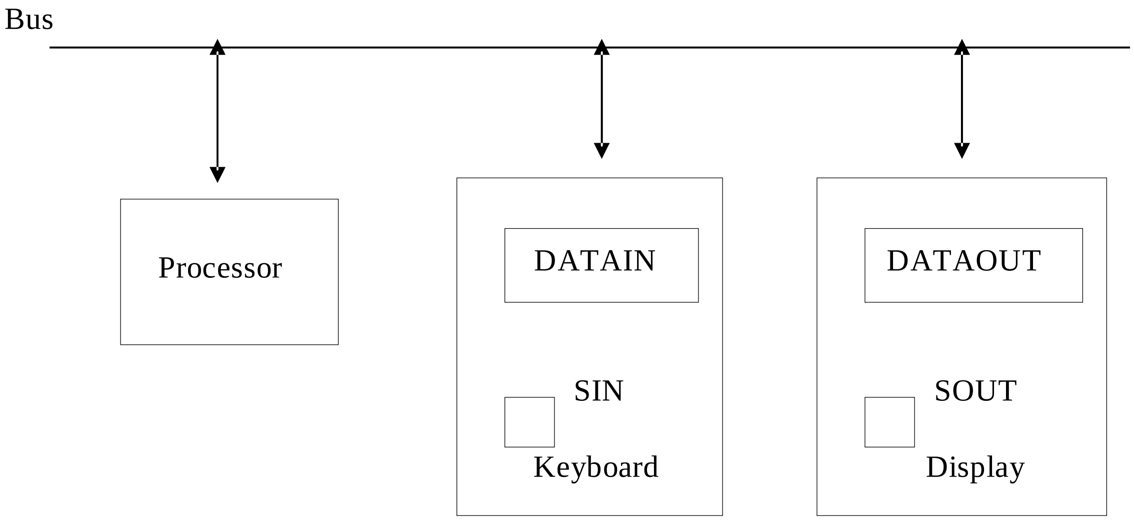


Fig a Bus connection for processor, keyboard, and display

The keyboard and the display are separate device as shown in fig a. the action of striking a key on the keyboard does not automatically cause the corresponding character to be displayed on the screen. One block of instructions in the I/O program transfers the character into the processor, and another associated block of instructions causes the character to be displayed.

Striking a key stores the corresponding character code in an 8-bit buffer register 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, the processor reads the contents of DATAIN. When the character is transferred to the processor, SIN is automatically cleared to 0. If a second character is entered at the keyboard, SIN is again set to 1, and the processor repeats.

An analogous process takes place when characters are transferred from the processor to the display. A buffer register, DATAOUT, and a status control flag, SOUT, are used for this transfer. When SOUT equals 1, the display is ready to receive a character.

In order to perform I/O transfers, we need machine instructions that can check the state of the status flags and transfer data between the processor and the I/O device. These instructions are similar in format to those used for moving data between the processor and the memory. For example, the processor can monitor the keyboard status flag SIN and transfer a character from DATAIN to register R1 by the following sequence of operations.

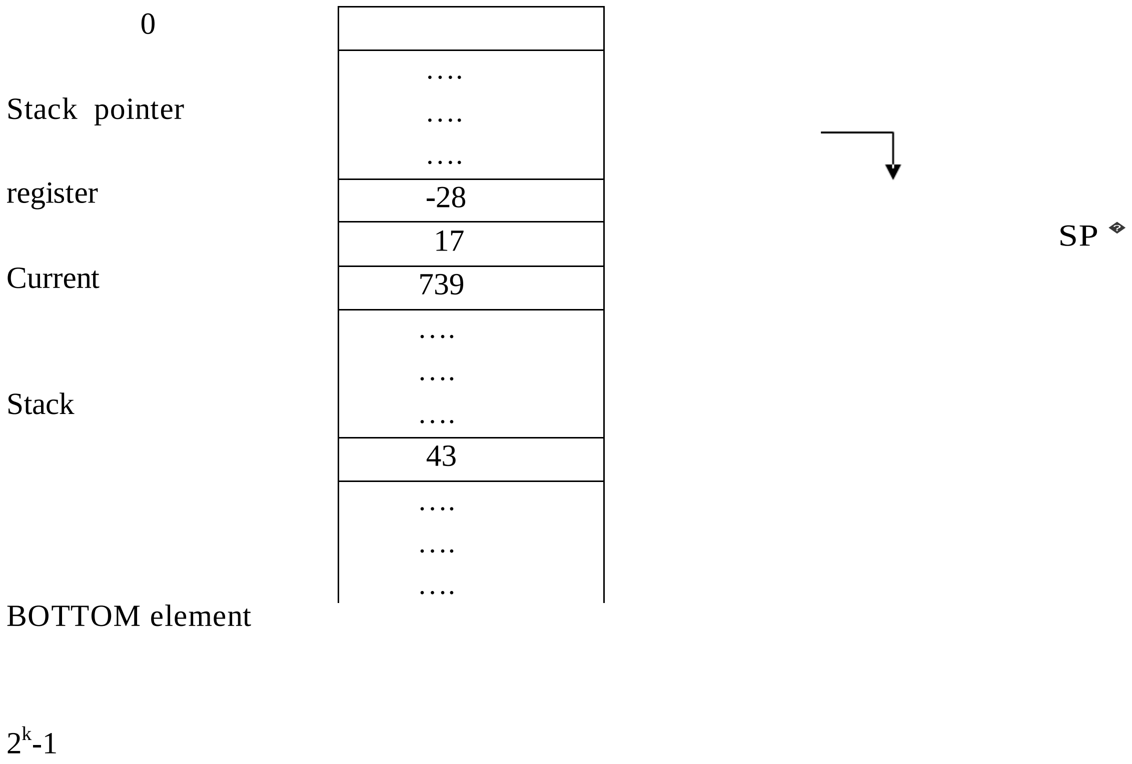
1.11 PUSH DOWN STACKS

A computer program often needs to perform a particular subtask using the familiar subroutine structure. In order to organize the control and information linkage between the main program and the subroutine, a data structure called a stack is used. This section will describe stacks, as well as a closely related data structure called a queue.

Data operated on by a program can be organized in a variety of ways. We have already encountered data structured as lists. Now, we consider an important data structure known as a stack. A stack is a list of data elements, usually words or bytes, with the accessing restriction that elements can be added or removed at one end of the list only. This end is called the top of the stack, and the other end is called the bottom. Another descriptive phrase, last-in-first-out (LIFO) stack, is also used to describe this type of storage mechanism; the last data item placed on the stack is the first one removed when retrieval begins. The terms push and pop are used to describe placing a new item on the stack and removing the top item from the stack, respectively.

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

Fig b A stack of words in the memory



Another useful data structure that is similar to the stack is called a queue. Data are stored in and retrieved from a queue on a first-in-first-out (FIFO) basis. Thus, if we assume that the queue grows in the direction of increasing addresses in the memory, which is a common practice, new data are added at the back (high-address end) and retrieved from the front (low-address end) of the queue.

There are two important differences between how a stack and a queue are implemented. One end of the stack is fixed (the bottom), while the other end rises and falls as data are pushed and popped. A single pointer is needed to point to the top of the stack at any given time. On the other hand, both ends of a queue move to higher addresses as data are added at the back and removed from the front. So two pointers are needed to keep track of the two ends of the queue.

Another difference between a stack and a queue is that, without further control, a queue would continuously move through the memory of a computer in the direction of higher addresses. One way to limit the queue to a fixed region in memory is to use a circular buffer. Let us assume that memory addresses from BEGINNING to END are assigned to the queue. The first entry in the queue is entered into location BEGINNING, and successive entries are appended to the queue by entering them at successively higher addresses. By the time the back of the queue reaches END, space will have been created at the beginning if some items have been removed from the queue. Hence, the back pointer is reset to the value BEGINNING and the process continues. As in the case of a stack, care must be taken to detect when the region assigned to the data structure is either completely full or completely empty.

1.12 SUBROUTINES

In a given program, it is often necessary to perform a particular subtask many times on different data-values. Such a subtask is usually called a subroutine. For example, a subroutine may evaluate the sine function or sort a list of values into increasing or decreasing order.

It is possible to include the block of instructions that constitute a subroutine at every place where it is needed in the program. However, to save space, only one copy of the instructions that constitute the subroutine is placed in the memory, and any program that requires the use of the subroutine simply branches to its starting location. When a program branches to a subroutine we say that it is calling the subroutine. The instruction that performs this branch operation is named a Call instruction.

After a subroutine has been executed, the calling program must resume execution, continuing immediately after the instruction that called the subroutine. The subroutine is said to return to the program that called it by executing a Return instruction.

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 .

Fig a illustrates this procedure

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

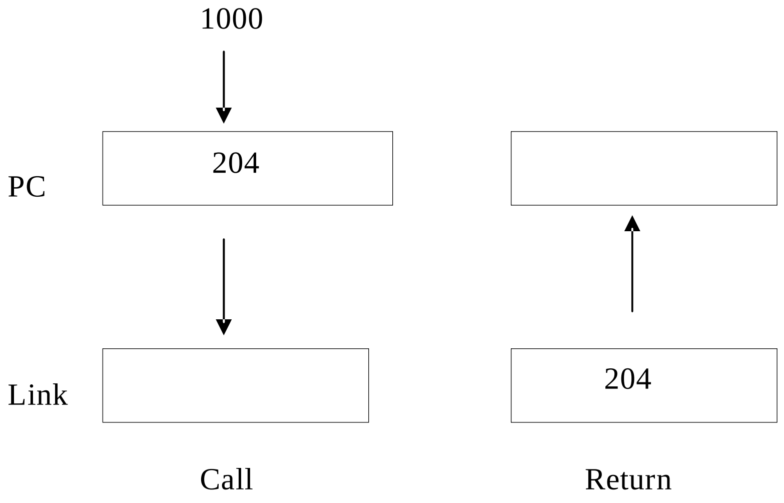


Fig b Subroutine linkage using a link register

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

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:-

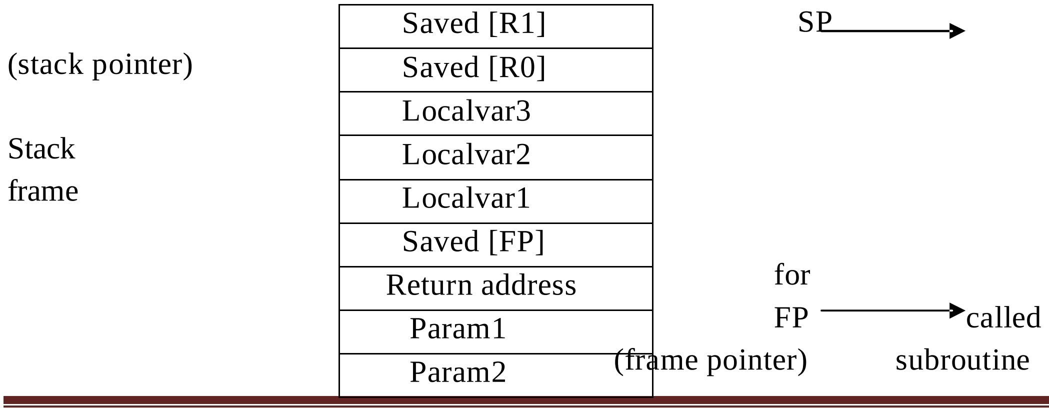
When calling a subroutine, a program must provide to the subroutine the parameters, that is, the operands or their addresses, to be used in the computation. Later, the subroutine returns other parameters, in this case, the results of the computation. This exchange of information between a calling program 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.



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 subroutine.

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.

Logic instructions

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

Not dst

SHIFT AND ROTATE INSTRUCTIONS:-

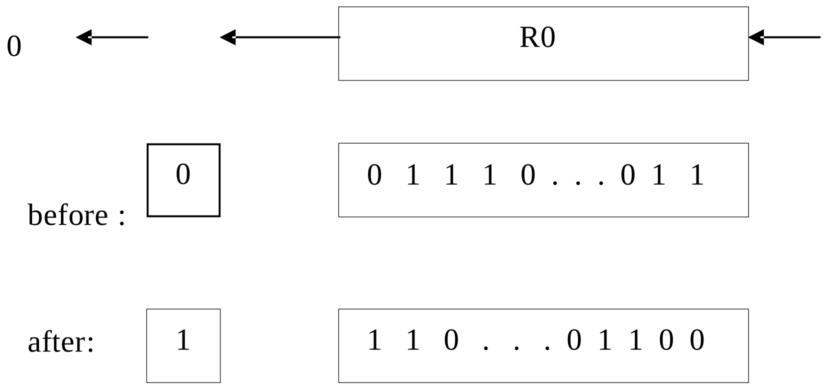
There are many applications that require the bits of an operand to be shifted right or left some specified number of bit positions. The details of how the shifts are performed depend on whether the operand is a signed number or some more general binary-coded information. For general operands, we use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

Logical shifts:-

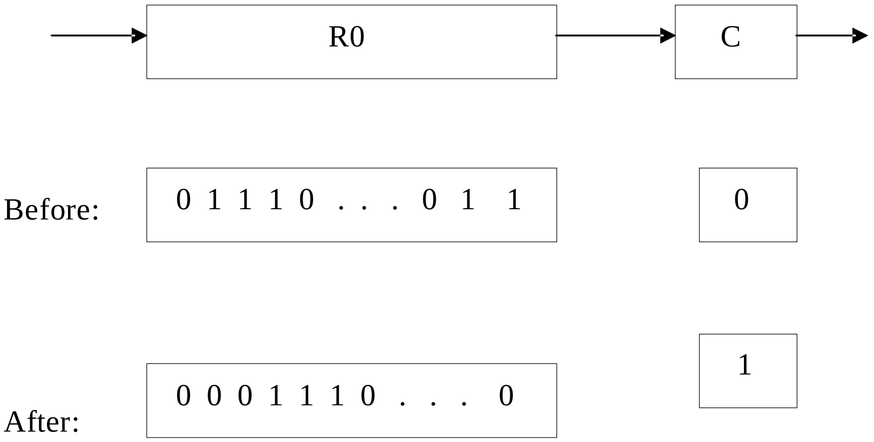
Two logical shift instructions are needed, one for shifting left (LShiftL) and another for shifting right (LShiftR). These instructions shift an operand over a number of bit positions specified in a count operand contained in the instruction. The general form of a logical left shift instruction is

LShiftL count, dst

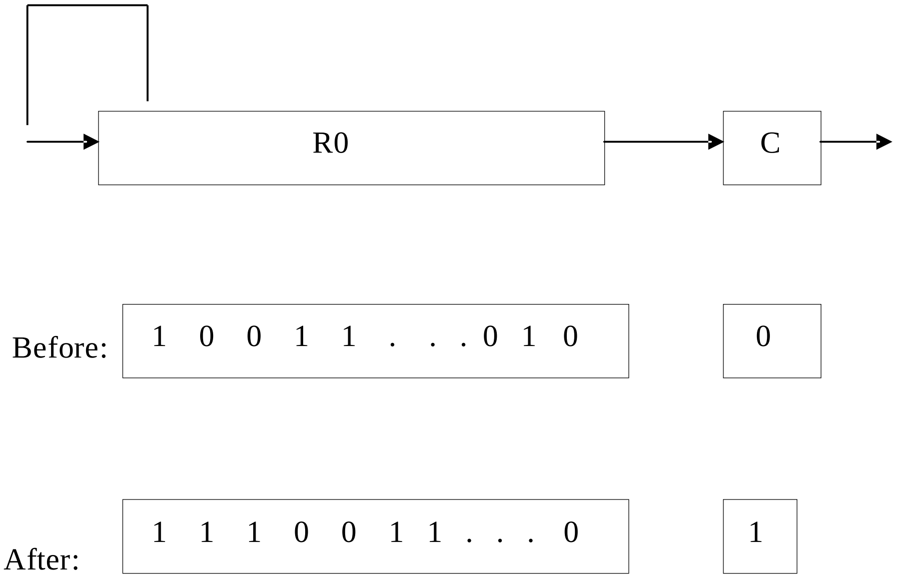
(a) Logical shift left LShiftL #2, R0



(b) Logical shift right LShiftR #2, R0



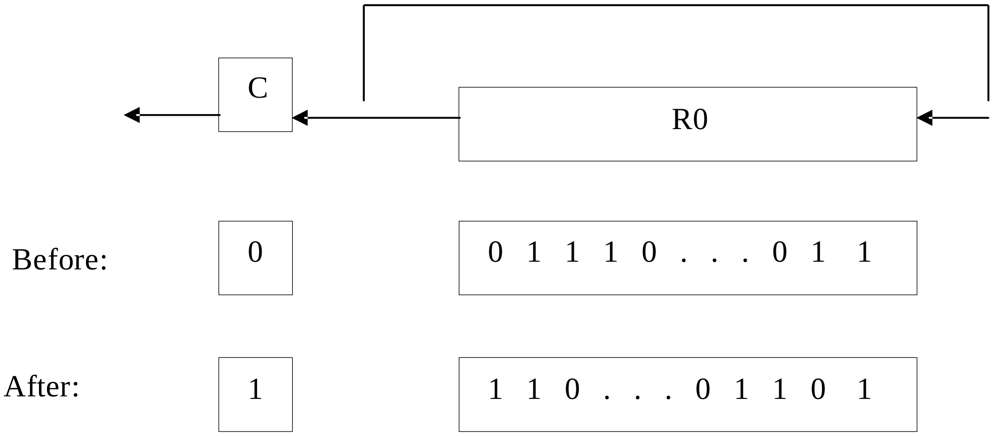
(c) Arithmetic shift right AShiftR #2, R0



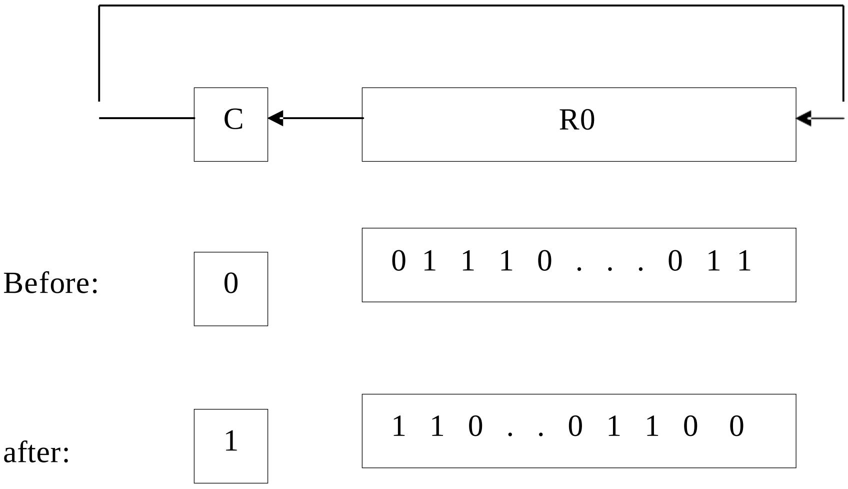
Rotate Operations:-

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

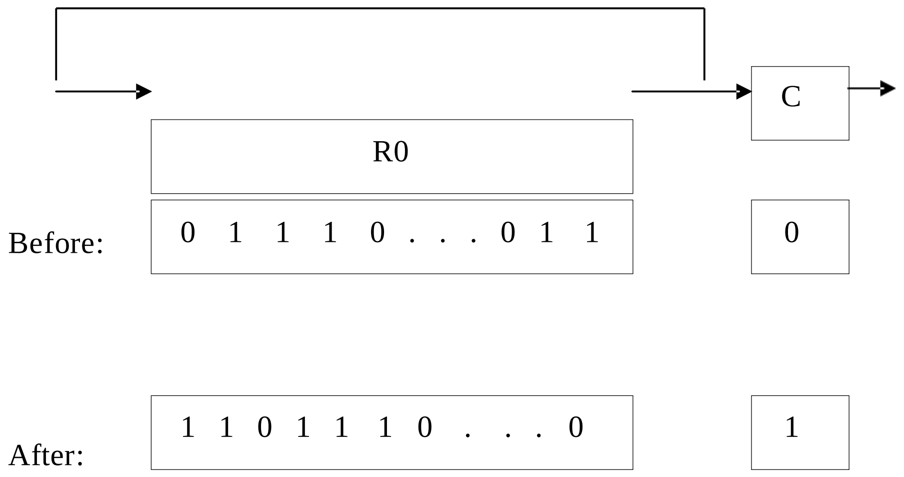
(a) Rotate left without carry RotateL #2, R0



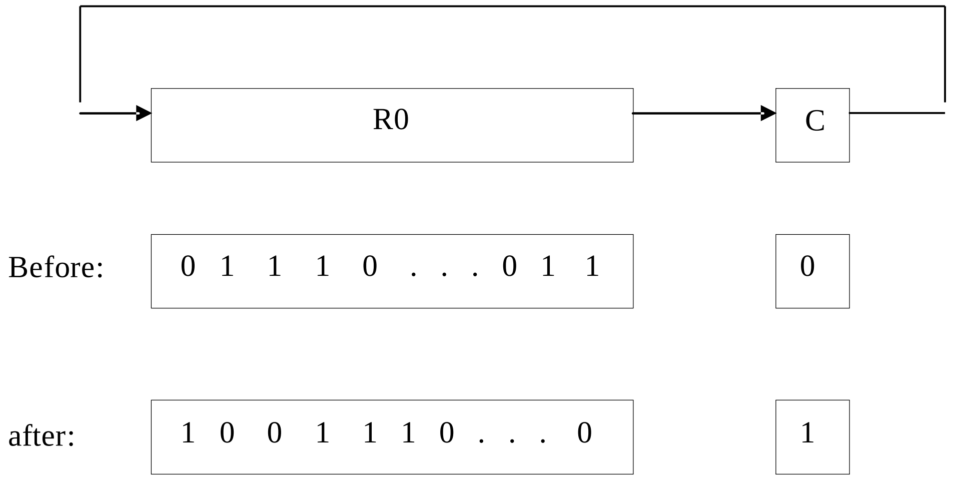
(b) Rotate left with carry RotateLC #2, R0



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



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



Encoding of machine instructions

We have introduced a variety of useful instructions and addressing modes. These instructions specify the actions that must be performed by the processor circuitry to carry out the desired tasks. We have often referred to them as machine instructions. Actually, the form in which we have presented the instructions is indicative of the form used in assembly languages, except that we tried to avoid using acronyms for the various operations, which are awkward to memorize and are likely to be specific to a particular commercial processor. To be executed in a processor, an instruction must be encoded in a compact binary pattern. Such encoded instructions are properly referred to as machine instructions. The instructions that use symbolic names and acronyms are called assembly language instructions, which are converted into the machine instructions using the assembler program.

We have seen instructions that perform operations such as add, subtract, move, shift, rotate, and branch. These instructions may use operands of different sizes, such as 32-bit and 8-bit numbers or 8-bit ASCII-encoded characters. The type of operation that is to be performed and the type of operands used may be specified using an encoded binary pattern referred to as the OP code for the given instruction. Suppose that 8 bits are allocated for this purpose, giving 256 possibilities for specifying different instructions. This leaves 24 bits to specify the rest of the required information.

Let us examine some typical cases. The instruction

Add R1, R2

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

The instruction

Move 24(R0), R5

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

The shift instruction

LShiftR #2, R0

And the move instruction

Move #$3A, R1

Have to indicate the immediate values 2 and #$3A, respectively, in addition to the 18 bits used to specify the OP code, the addressing modes, and the register. This limits the size of the immediate operand to what is expressible in 14 bits.

Consider next the branch instruction

Branch >0 LOOP

Again, 8 bits are used for the OP code, leaving 24 bits to specify the branch offset. Since the offset is a 2's-complement number, the branch target address must be within 223 bytes of the location of the branch instruction. To branch to an instruction outside this range, a different addressing mode has to be used, such as Absolute or Register Indirect. Branch instructions that use these modes are usually called Jump instructions.

In all these examples, the instructions can be encoded in a 32-bit word. Depicts a possible format. There is an 8-bit Op-code field and two 7-bit fields for specifying the source and destination operands. The 7-bit field identifies the addressing mode and the register involved (if any). The “Other info” field allows us to specify the additional information that may be needed, such as an index value or an immediate operand.

But, what happens if we want to specify a memory operand using the Absolute addressing mode? The instruction

Move R2, LOC

(a) One-word instruction

Opcode	Source	Dest	Other info

(b) Two-Word instruction

Opcode	Source	Dest	Other info
Memory address/Immediate operand			

(c) Three-operand instruction

Op code	Ri	Rj	Rk	Other info

Requires 18 bits to denote the OP code, the addressing modes, and the register. This leaves 14 bits to express the address that corresponds to LOC, which is clearly insufficient.

And #$FF000000. R2

In which case the second word gives a full 32-bit immediate operand.

If we want to allow an instruction in which two operands can be specified using the Absolute addressing mode, for example

```
Move    LOC1, LOC2
```

Then it becomes necessary to use tow additional words for the 32-bit addresses of the operands.

This approach results in instructions of variable length, dependent on the number of operands and the type of addressing modes used. Using multiple words, we can implement quite complex instructions, closely resembling operations in high-level programming languages. The term complex instruction set computer (CISC) has been used to refer to processors that use instruction sets of this type.

The restriction that an instruction must occupy only one word has led to a style of computers that have become known as reduced instruction set computer (RISC). The RISC approach introduced other restrictions, such as that all manipulation of data must be done on operands that are already in processor registers. This restriction means that the above addition would need a two-instruction sequence

```
Move   (R3), R1
Add     R1, R2
```

If the Add instruction only has to specify the two registers, it will need just a portion of a 32-bit word. So, we may provide a more powerful instruction that uses three operands

```
Add     R1, R2, R3
```

Which performs the operation

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

A possible format for such an instruction in shown in fig c. Of course, the processor has to be able to deal with such three-operand instructions. In an instruction set where all arithmetic and logical operations use only register operands, the only memory references are made to load/store the operands into/from the processor registers.

RISC-type instruction sets typically have fewer and less complex instructions than CISC-type sets. We will discuss the relative merits of RISC and CISC approaches in Chapter 8, which deals with the details of processor design.

UNIT-2

Register Transfer Language and Design of Control Unit

2.1. Register Transfer Language

• Digital systems are composed of modules that are constructed from digital components, such as registers, decoders, arithmetic elements, and control logic

• The modules are interconnected with common data and control paths to form a digital computer system

• The operations executed on data stored in registers are called micro operations • A micro operation is an elementary operation performed on the information stored in one or more registers

• Examples are shift, count, clear, and load

• Some of the digital components from before are registers that implement micro operations

• The internal hardware organization of a digital computer is best defined by specifying

- The set of registers it contains and their functions
- The sequence of micro operations performed on the binary information stored
- The control that initiates the sequence of micro operations

• Use symbols, rather than words, to specify the sequence of micro operations

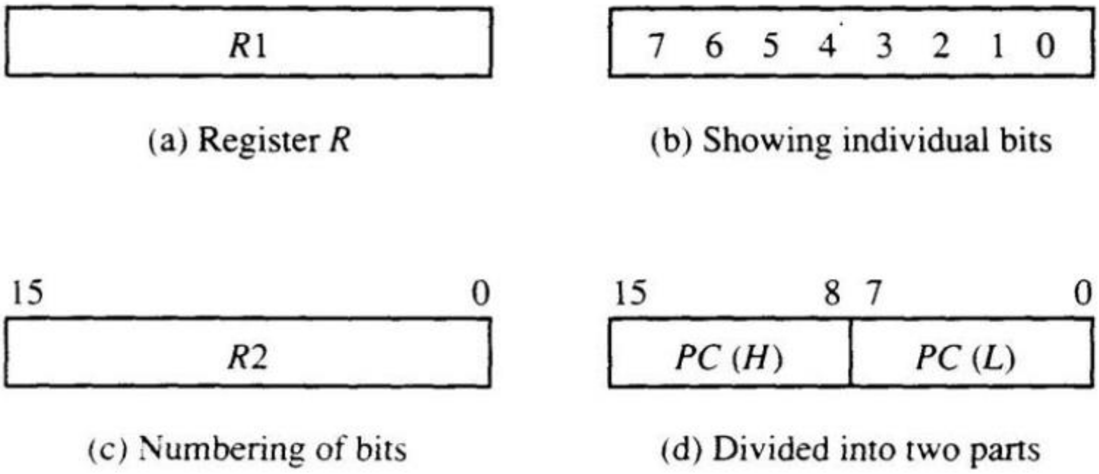
• The symbolic notation used is called a register transfer language

• A programming language is a procedure for writing symbols to specify a given computational process

• Define symbols for various types of micro operations and describe associated hardware that can implement the micro operations

2.2. Register Transfer

- Designate computer registers by capital letters to denote its function
- The register that holds an address for the memory unit is called MAR
- The program counter register is called PC
- IR is the instruction register and R1 is a processor register
- The individual flip-flops in an n-bit register are numbered in sequence from 0 to n-1
- Refer to Figure 4.1 for the different representations of a register



(a) Register R

(b) Showing individual bits

(c) Numbering of bits

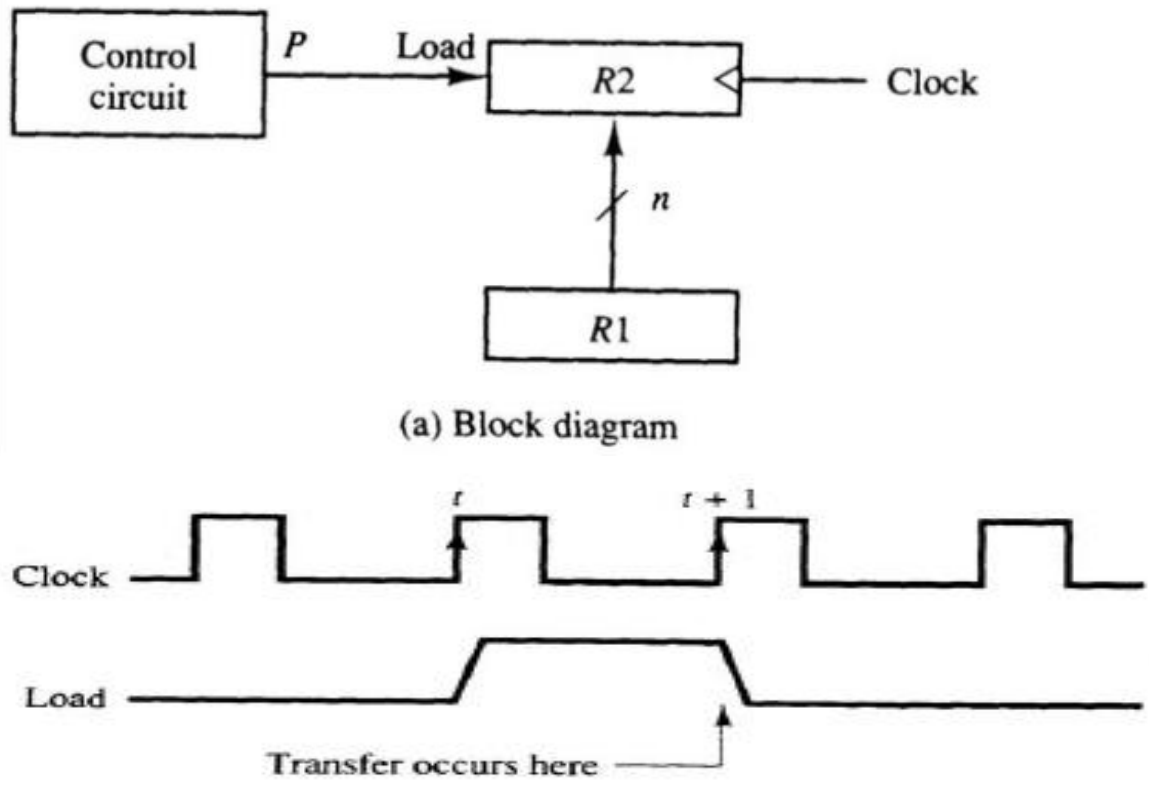
(d) Divided into two parts

- Designate information transfer from one register to another by R2 ← R1
- This statement implies that the hardware is available
    - The outputs of the source must have a path to the inputs of the destination
    - The destination register has a parallel load capability
- If the transfer is to occur only under a predetermined control condition, designate it by

If (P = 1) *then* (R2 ← R1) or,
P: R2← R1,
Where P is a control function that can be either 0 or 1

- Every statement written in register transfer notation implies the presence of the required hardware construction

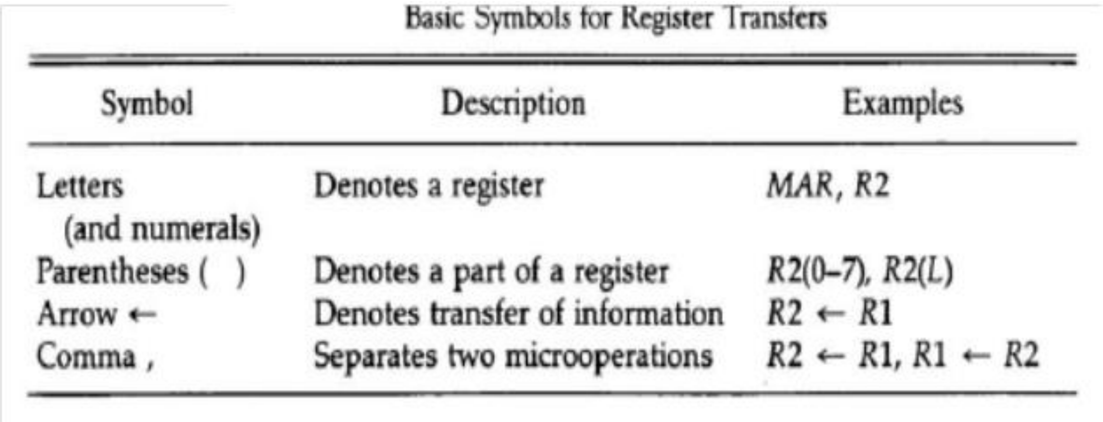
Transfer from R1 to R2 when P = 1.



(a) Block diagram

(b) Timing diagram

- It is assumed that all transfers occur during a clock edge transition
- All micro operations written on a single line are to be executed at the same time T: R2 ← R1, R1 ← R2



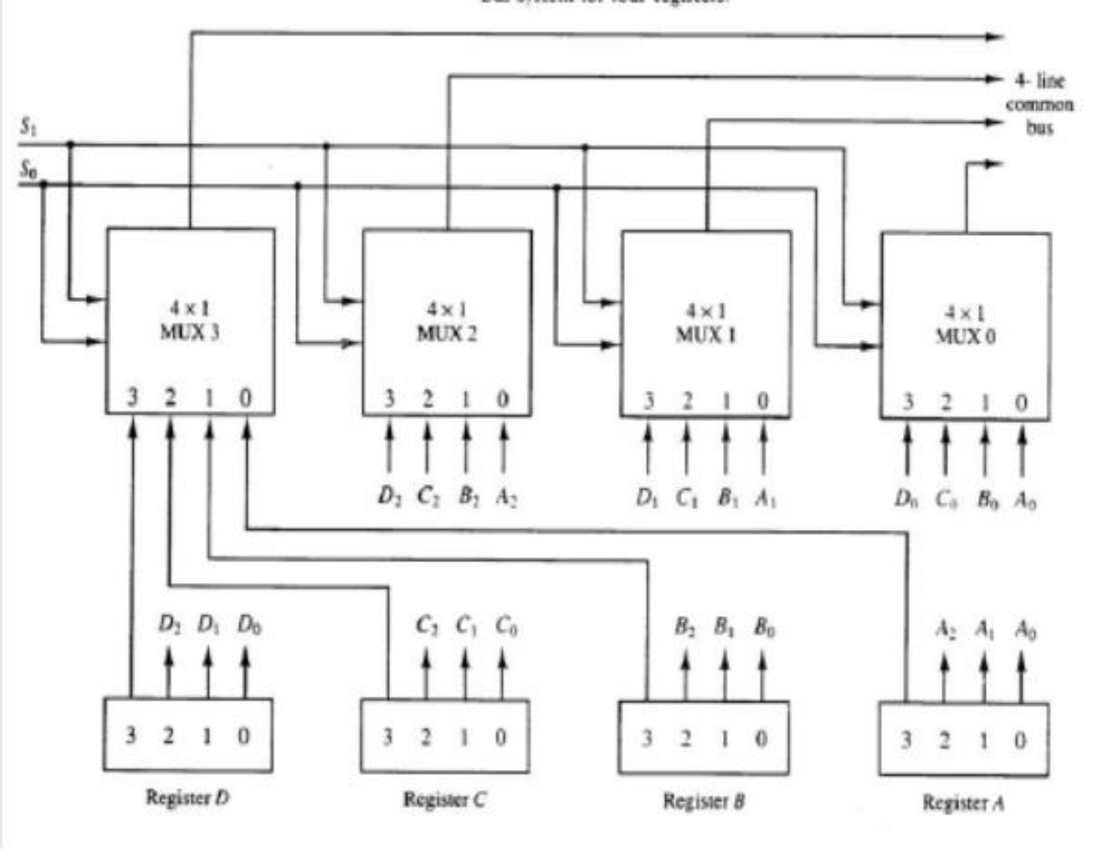
Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	*MAR*, *R*2
Parentheses ( )	Denotes a part of a register	*R*2(0–7), *R*2(*L*)
Arrow ←	Denotes transfer of information	*R*2 ← *R*1
Comma ,	Separates two microoperations	*R*2 ← *R*1, *R*1 ← *R*2

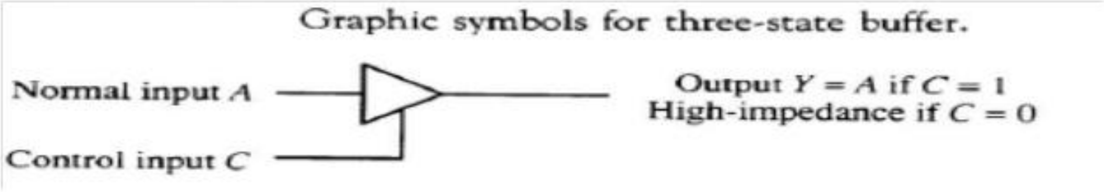
2.3. Bus and Memory Transfers

• Rather than connecting wires between all registers, a common bus is used
• A bus structure consists of a set of common lines, one for each bit of a register
• Control signals determine which register is selected by the bus during each transfer
• Multiplexers can be used to construct a common bus
• Multiplexers select the source register whose binary information is then placed on the bus
• The select lines are connected to the selection inputs of the multiplexers and choose the bits of one register .

Bus system for four registers.

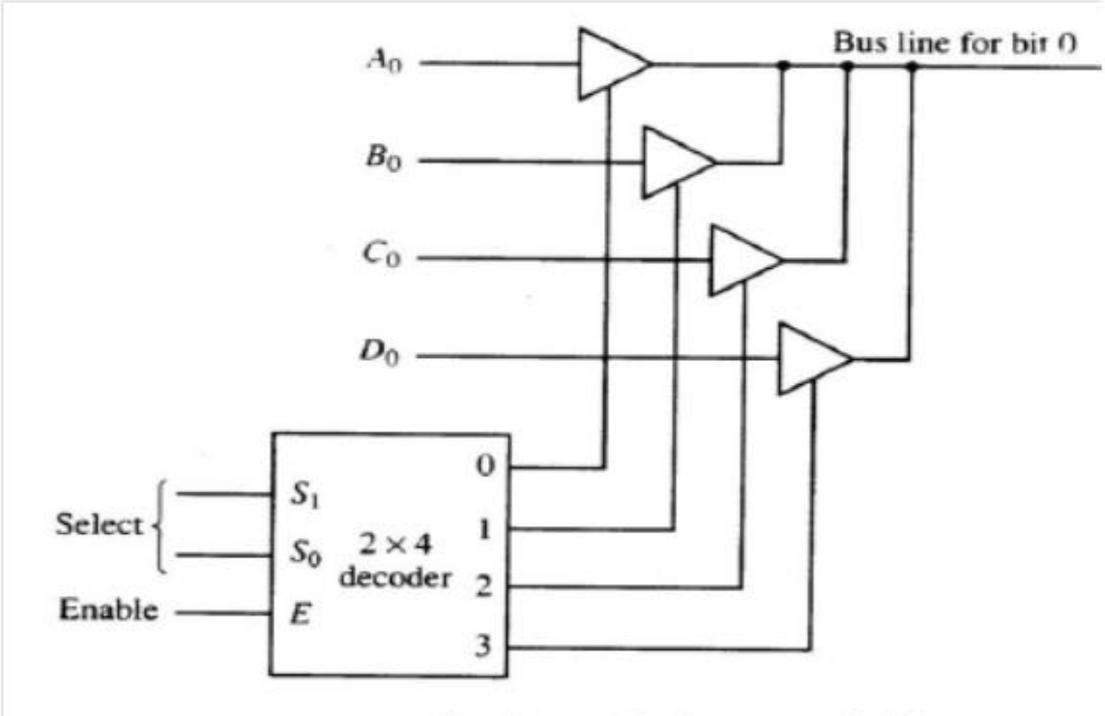


• In general, a system will multiplex *k* registers of *n* bits each to produce an *n*-line common bus
• This requires *n* multiplexers – one for each bit
• The size of each multiplexer must be *k* x 1
• The number of select lines required is *log k* • To transfer information from the bus to a register, the bus lines are connected to the inputs of all destination registers and the corresponding load control line must be activated
• Rather than listing each step as
BUS ← C, R1 ← BUS,
use R1 ← C, since the bus is implied
• Instead of using multiplexers, *three-state gates* can be used to construct the bus system
• A three-state gate is a digital circuit that exhibits three states
• Two of the states are signals equivalent to logic 1 and 0
• The third state is a *high-impedance* state – this behaves like an open circuit, which means the output is disconnected and does not have a logic significance .


Graphic symbols for three-state buffer.
Normal input A
Control input C
Output $Y = A$ if $C = 1$
High-impedance if $C = 0$

- The three-state buffer gate has a normal input and a control input which determines the output state

• With control 1, the output equals the normal input
• With control 0, the gate goes to a high-impedance state
• This enables a large number of three-state gate outputs to be connected with wires to form a common bus line without endangering loading effects.



Bus line with three state-buffers.

• Decoders are used to ensure that no more than one control input is active at any given time
• This circuit can replace the multiplexer in Figure 4.3
• To construct a common bus for four registers of *n* bits each using three-state buffers, we need *n* circuits with four buffers in each
• Only one decoder is necessary to select between the four registers
• Designate a memory word by the letter M
• It is necessary to specify the address of M when writing memory transfer operations
• Designate the address register by AR and the data register by DR
• The read operation can be stated as:
Read: DR ← M[AR]
• The write operation can be stated as:
Write: M[AR] ← R1

2.4. Arithmetic Micro operations

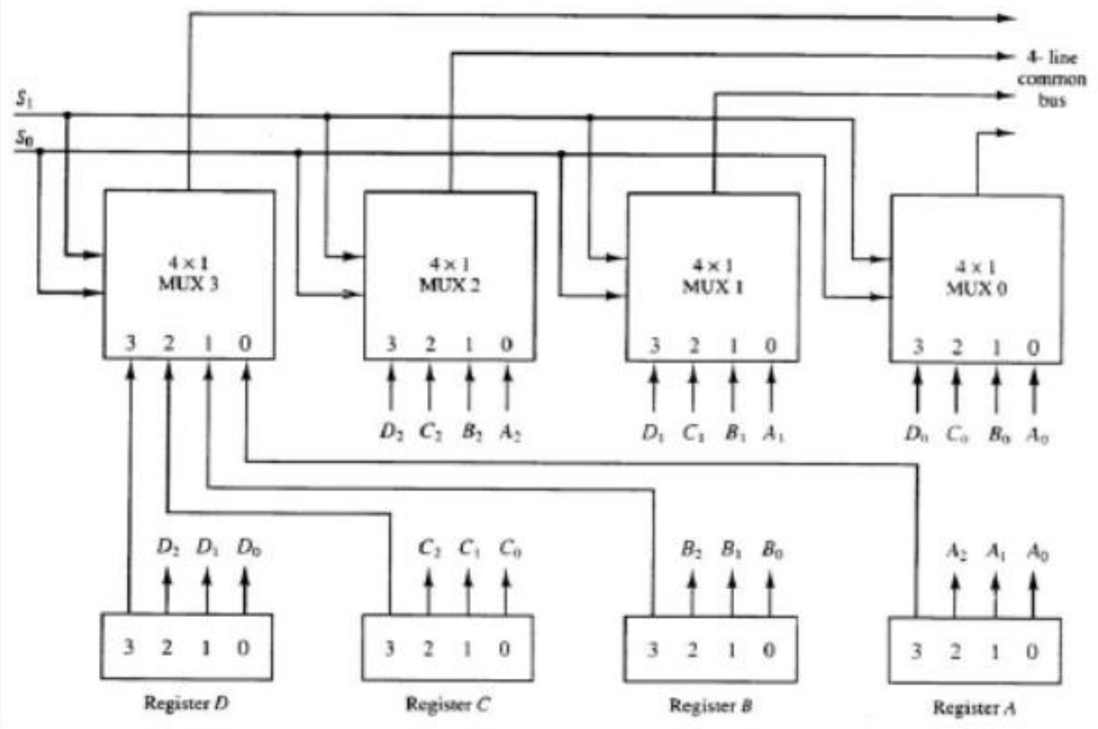
• There are four categories of the most common micro operations:

- Register transfer: transfer binary information from one register to another
- Arithmetic: perform arithmetic operations on numeric data stored in registers
- Logic: perform bit manipulation operations on non-numeric data stored in registers
- Shift: perform shift operations on data stored in registers

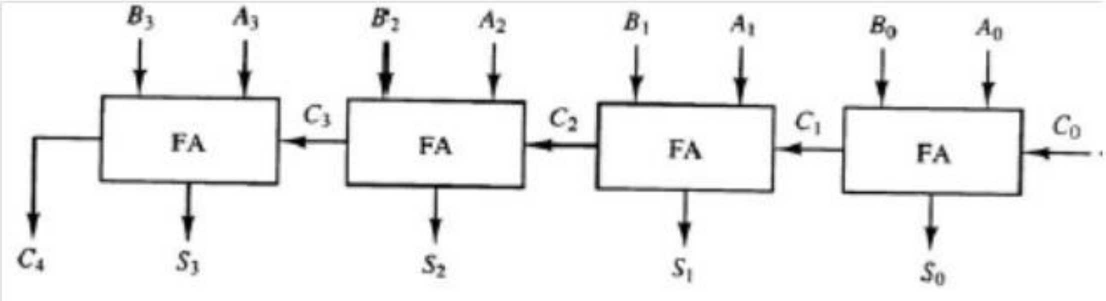
• The basic arithmetic micro operations are addition, subtraction, increment, decrement, and shift

- Example of addition: R3 ← R1 +R2
- Subtraction is most often implemented through complementation and addition
- Example of subtraction: R3 ← R1 + R2 + 1 (strikethrough denotes bar on top – 1’s complement of R2)
- Adding 1 to the 1’s complement produces the 2’s complement .
- Adding the contents of R1 to the 2’s complement of R2 is equivalent to subtracting.

Bus system for four registers.

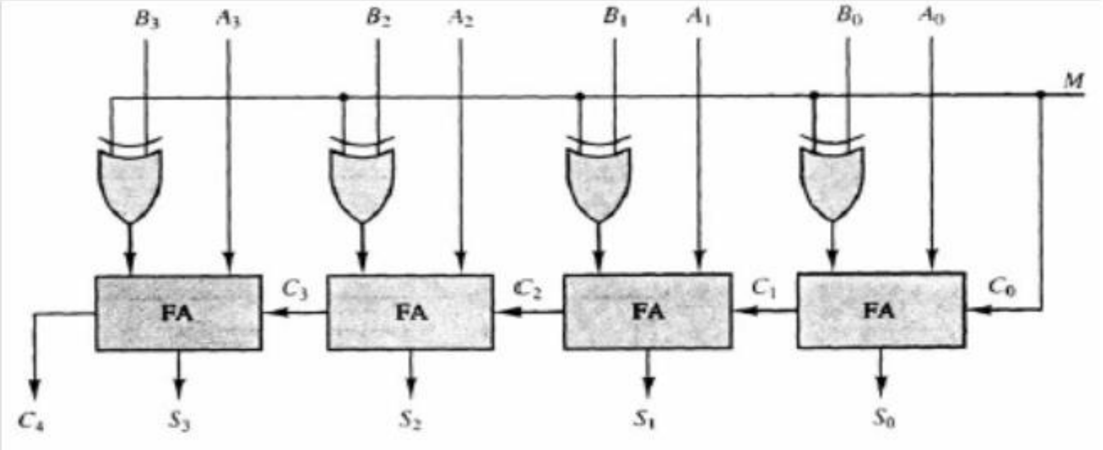


- Multiply and divide are not included as micro operations
- A micro operation is one that can be executed by one clock pulse
- Multiply (divide) is implemented by a sequence of add and shift micro operations (subtract and shift)
- To implement the add micro operation with hardware, we need the registers that hold the data and the digital component that performs the addition
- A full-adder adds two bits and a previous carry.
- A *binary adder* is a digital circuit that generates the arithmetic sum of two binary numbers of any lengths
- A binary added is constructed with full-adder circuits connected in cascade
- An *n*-bit binary adder requires *n* full-adders



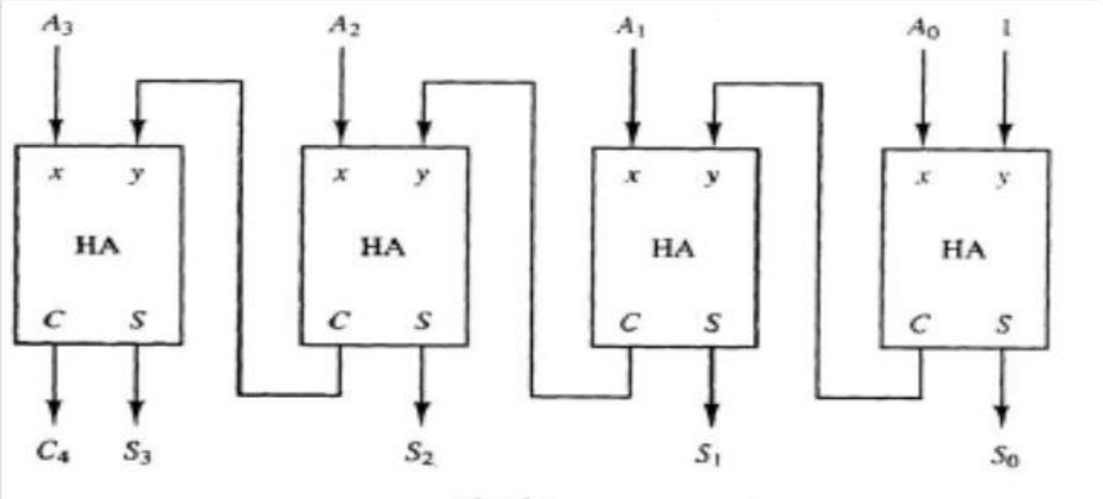
4-bit binary adder.

- The subtraction A-B can be carried out by the following steps
  - Take the 1's complement of B (invert each bit)
  - Get the 2's complement by adding 1
  - Add the result to A
- The addition and subtraction operations can be combined into one common circuit by including an XOR gate with each full-adder.



4-bit adder-subtractor.

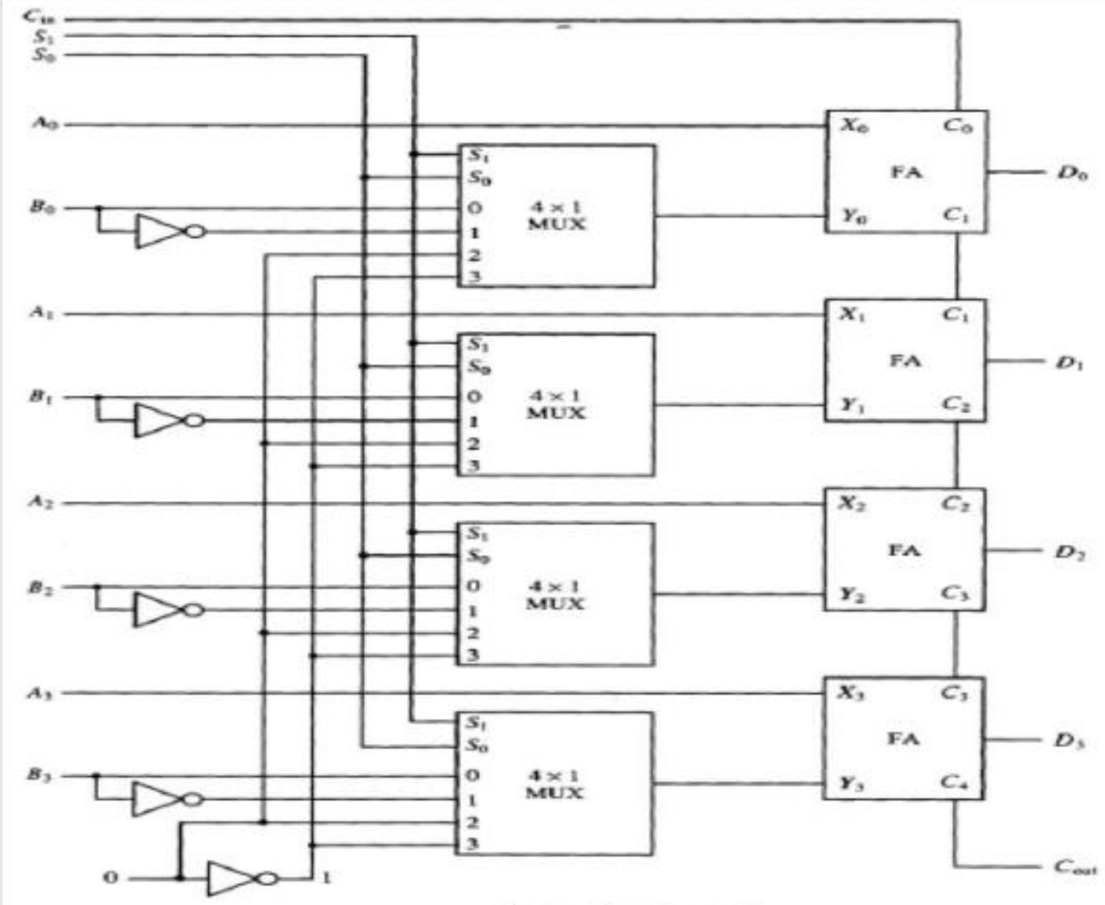
- The increment micro operation adds one to a number in a register
- This can be implemented by using a binary counter – every time the count enable is active, the count is incremented by one
- If the increment is to be performed independent of a particular register, then use half-adders connected in cascade .
- An *n*-bit binary incrementer requires *n* half-adders



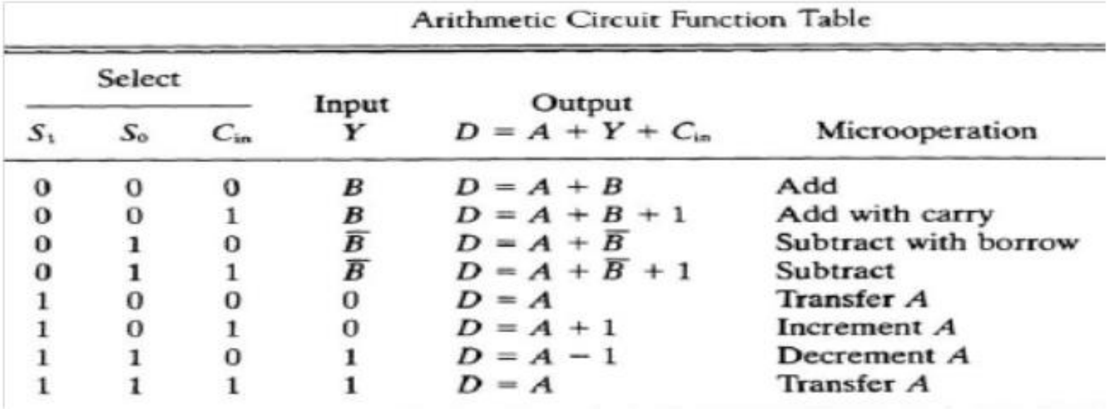
4-bit binary incrementer.

- Each of the arithmetic micro operations can be implemented in one composite arithmetic circuit
- The basic component is the parallel adder
- Multiplexers are used to choose between the different operations
- The output of the binary adder is calculated from the following sum:

$$D = A + Y + C_{in}$$



4-bit arithmetic circuit.



Arithmetic Circuit Function Table

Select			Input	Output	
S_1	S_0	C_{in}	Y	$D = A + Y + C_{in}$	Microoperation
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add with carry
0	1	0	$\overline{B}$	$D = A + \overline{B}$	Subtract with borrow
0	1	1	$\overline{B}$	$D = A + \overline{B} + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

2.5. Logic Micro operations

• Logic operations specify binary operations for strings of bits stored in registers and treat each bit separately

• Example: the XOR of R1 and R2 is symbolized by P: R1 ← R1 ⊕ R2

• Example: R1 = 1010 and R2 = 1100

1010 Content of R1

1100 Content of R2

0110 Content of R1 after P = 1

• Symbols used for logical micro operations:

- OR: ∨
- AND: ∧
- XOR: ⊕

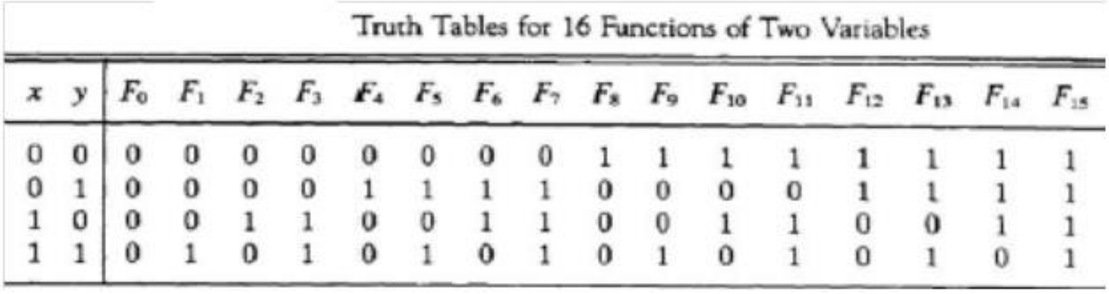
• The + sign has two different meanings: logical OR and summation

• When + is in a micro operation, then summation

• When + is in a control function, then OR

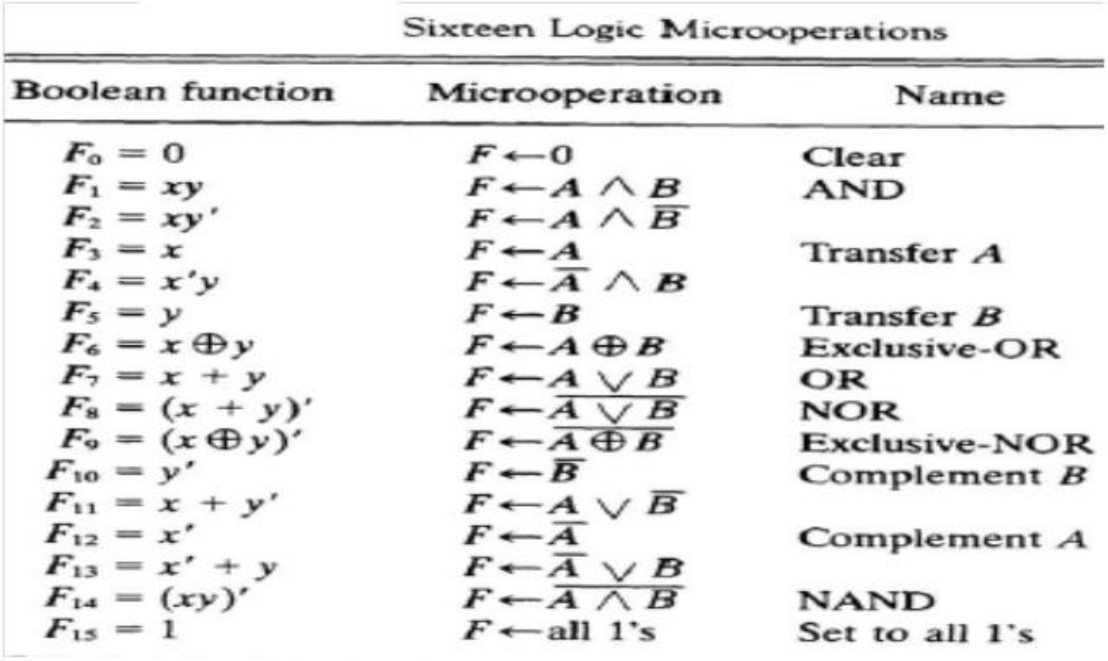
• Example: P + Q: R1 ← R2 + R3, R4 ← R5 ∨ R6

• There are 16 different logic operations that can be performed with two binary variables.



Truth Tables for 16 Functions of Two Variables

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1



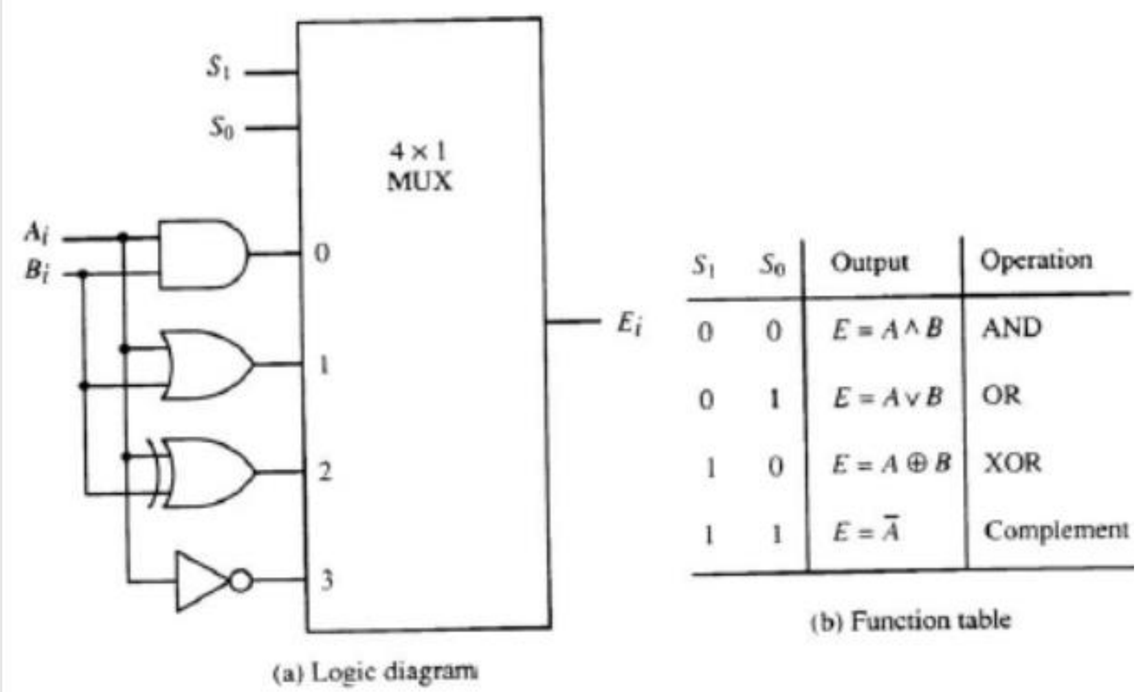
Sixteen Logic Microoperations

Boolean function	Microoperation	Name
$F_0 = 0$	$F \leftarrow 0$	Clear
$F_1 = xy$	$F \leftarrow A \wedge B$	AND
$F_2 = xy'$	$F \leftarrow A \wedge \overline{B}$	
$F_3 = x$	$F \leftarrow A$	Transfer A
$F_4 = x'y$	$F \leftarrow \overline{A} \wedge B$	
$F_5 = y$	$F \leftarrow B$	Transfer B
$F_6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
$F_7 = x + y$	$F \leftarrow A \vee B$	OR
$F_8 = (x + y)'$	$F \leftarrow \overline{A \vee B}$	NOR
$F_9 = (x \oplus y)'$	$F \leftarrow \overline{A \oplus B}$	Exclusive-NOR
$F_{10} = y'$	$F \leftarrow \overline{B}$	Complement B
$F_{11} = x + y'$	$F \leftarrow A \vee \overline{B}$	
$F_{12} = x'$	$F \leftarrow \overline{A}$	Complement A
$F_{13} = x' + y$	$F \leftarrow \overline{A} \vee B$	
$F_{14} = (xy)'$	$F \leftarrow \overline{A \wedge B}$	NAND
$F_{15} = 1$	$F \leftarrow$ all 1's	Set to all 1's

- The hardware implementation of logic micro operations requires that logic gates be inserted for each bit or pair of bits in the registers.

- All 16 micro operations can be derived from using four logic gates.

One stage of logic circuit.



(a) Logic diagram

S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \overline{A}$	Complement

(b) Function table

• Logic micro operations can be used to change bit values, delete a group of bits, or insert new bit values into a register

• The *selective-set* operation sets to 1 the bits in A where there are corresponding 1's in B

1010 A before
1100 B (logic operand)
1110 A after
A ← A ∨ B

• The *selective-complement* operation complements bits in A where there are corresponding 1's in B

1010 A before
1100 B (logic operand)
0110 A after
A ← A ⊕ B

• The *selective-clear* operation clears to 0 the bits in A only where there are corresponding 1's in B

1010 A before
1100 B (logic operand)
0010 A after

A ← A ∧ B

The *mask* operation is similar to the selective-clear operation, except that the bits of A are cleared only where there are corresponding 0's in B

1010 A before
1100 B (logic operand)
1000 A after
A ← A ∧ B

• The *insert* operation inserts a new value into a group of bits

• This is done by first masking the bits to be replaced and then Oring them with the bits to be inserted 0110 1010 A before
<u>0000 1111 </u>B (mask)
0000 1010 A after masking
0000 1010 A before
<u>1001 0000 </u>B (insert)
1001 1010 A after insertion

• The *clear* operation compares the bits in A and B and produces an all 0's result if the two number are equal

1010 A
1010 B
0000 A ← A ⊕ B

<u>2.6. Shift Micro operations</u>

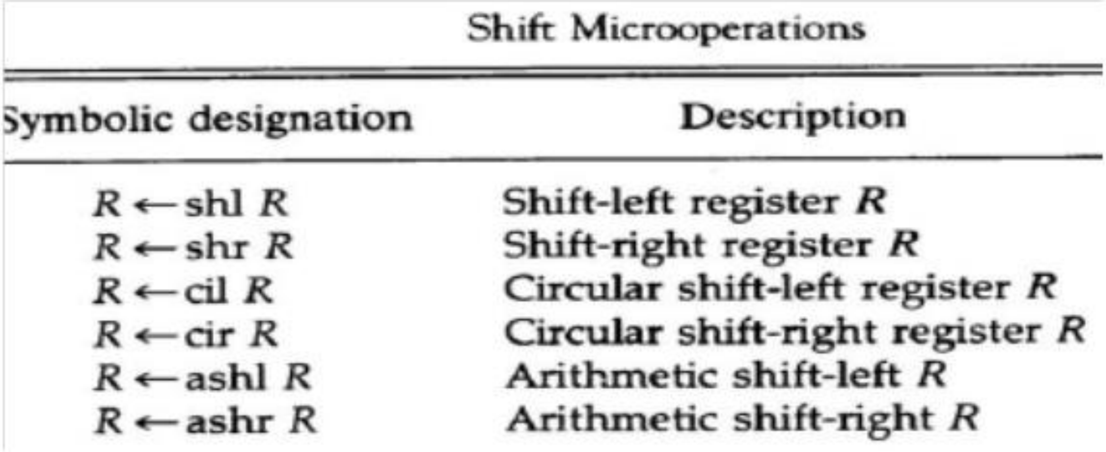
• Shift microoperations are used for serial transfer of data

• They are also used in conjunction with arithmetic, logic, and other data-processing operations

• There are three types of shifts: logical, circular, and arithmetic

• A *logical shift* is one that transfers 0 through the serial input

- The symbols *shl* and *shr* are for logical shift-left and shift-right by one position R1 ← shl R1
- The *circular shift* (aka rotate) circulates the bits of the register around the two ends without loss of information
- The symbols *cil* and *cir* are for circular shift left and right

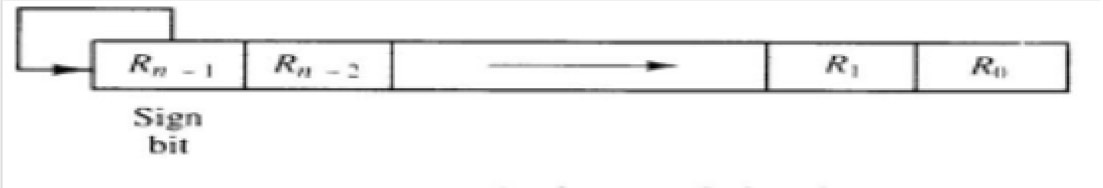


Shift Microoperations

Symbolic designation	Description
$R \leftarrow \text{shl } R$	Shift-left register R
$R \leftarrow \text{shr } R$	Shift-right register R
$R \leftarrow \text{cil } R$	Circular shift-left register R
$R \leftarrow \text{cir } R$	Circular shift-right register R
$R \leftarrow \text{ashl } R$	Arithmetic shift-left R
$R \leftarrow \text{ashr } R$	Arithmetic shift-right R

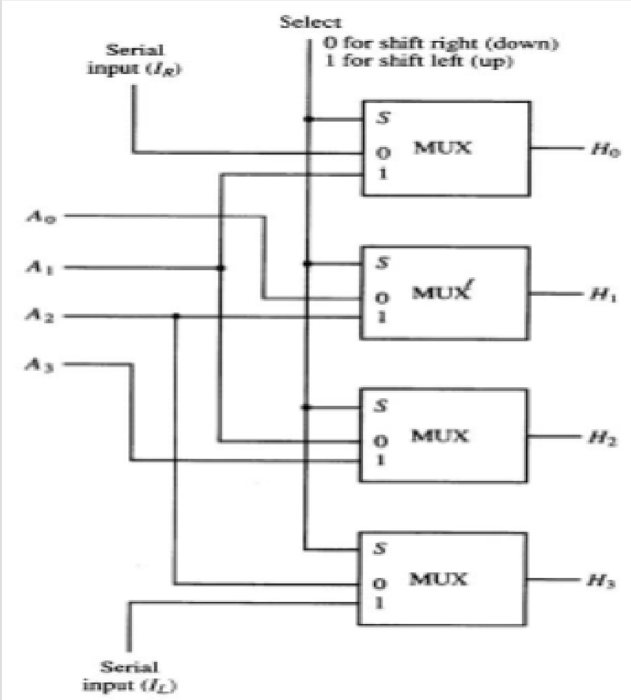
- The *arithmetic shift* shifts a signed binary number to the left or right
- To the left is multiplying by 2, to the right is dividing by 2
- Arithmetic shifts must leave the sign bit unchanged
- A sign reversal occurs if the bit in R_{n-1} changes in value after the shift
- This happens if the multiplication causes an overflow
- An overflow flip-flop V_s can be used to detect the overflow

$$V_s = R_{n-1} \oplus R_{n-2}$$



Arithmetic shift right.

- A bi-directional shift unit with parallel load could be used to implement this
- Two clock pulses are necessary with this configuration: one to load the value and another to shift
- In a processor unit with many registers it is more efficient to implement the shift operation with a combinational circuit
- The content of a register to be shifted is first placed onto a common bus and the output is connected to the combinational shifter, the shifted number is then loaded back into the register
- This can be constructed with multiplexers



Function table

Select	Output			
S	H_0	H_1	H_2	H_3
0	I_R	A_0	A_1	A_2
1	A_1	A_2	A_3	I_L

4-bit combinational circuit shifter.

Arithmetic Logic Shift Unit

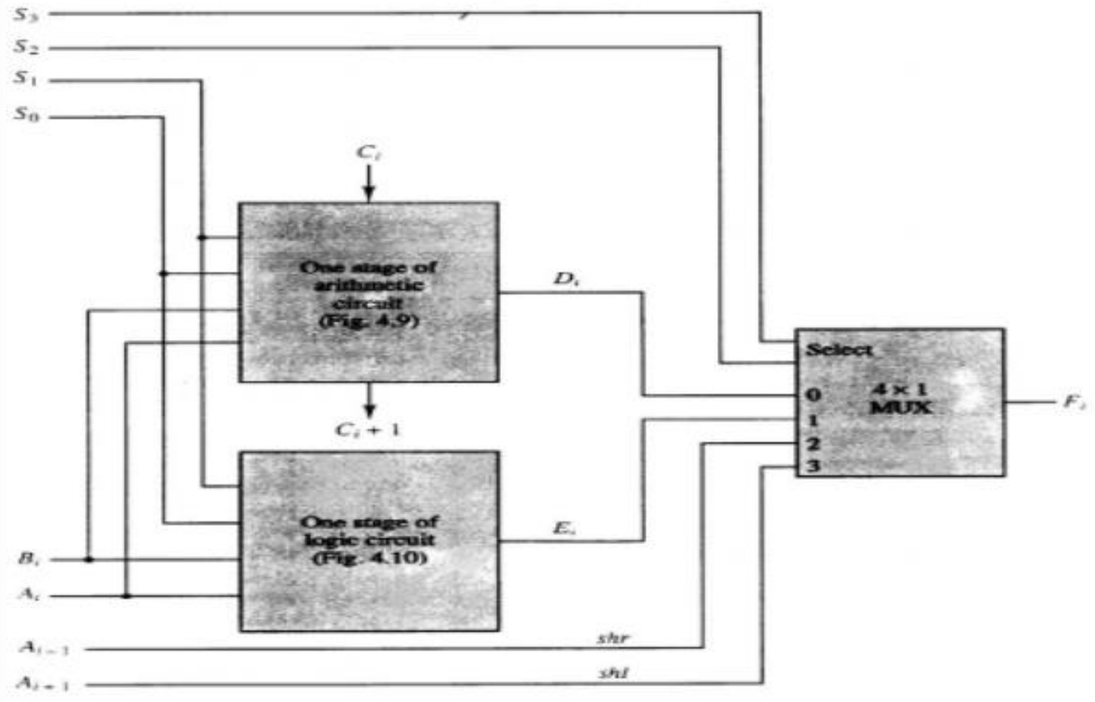
• The *arithmetic logic unit (ALU)* is a common operational unit connected to a number of storage registers

• To perform a microoperation, the contents of specified registers are placed in the inputs of the ALU

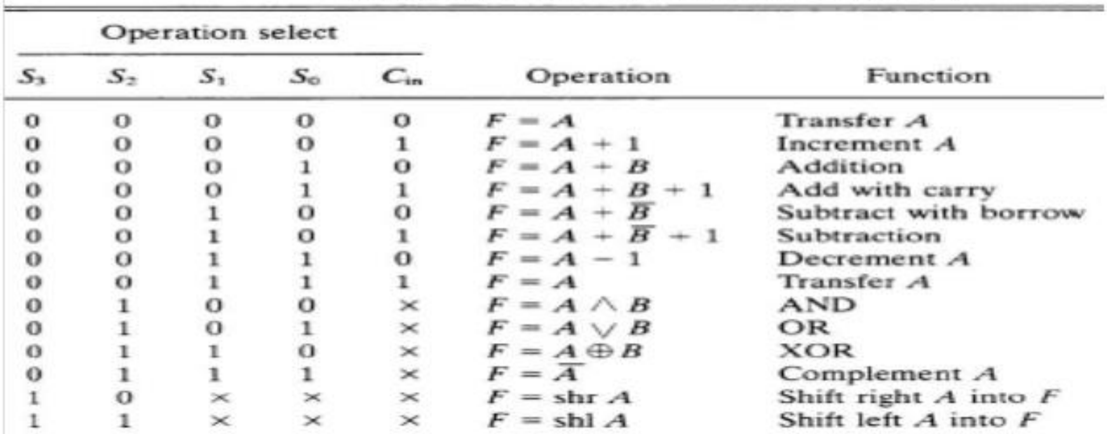
• The ALU performs an operation and the result is then transferred to a destination register

• The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period

One stage of arithmetic logic shift unit.

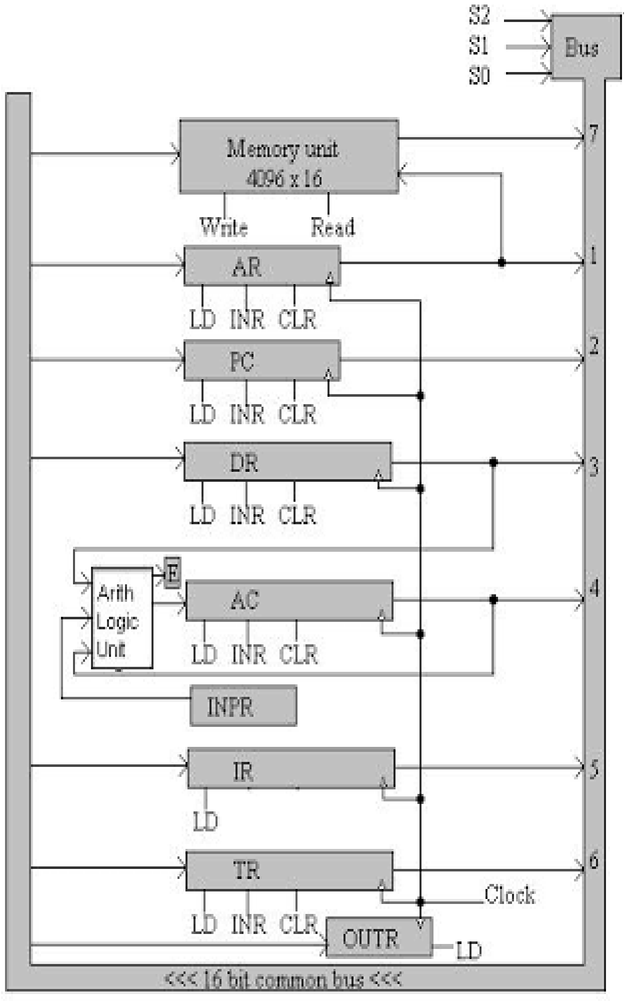


Function Table for Arithmetic Logic Shift Unit



Operation select						
S_3	S_2	S_1	S_0	C_{in}	Operation	Function
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	×	$F = A \wedge B$	AND
0	1	0	1	×	$F = A \vee B$	OR
0	1	1	0	×	$F = A \oplus B$	XOR
0	1	1	1	×	$F = \overline{A}$	Complement A
1	0	×	×	×	$F = \text{shr } A$	Shift right A into F
1	1	×	×	×	$F = \text{shl } A$	Shift left A into F

COMMON BUS SYSTEM



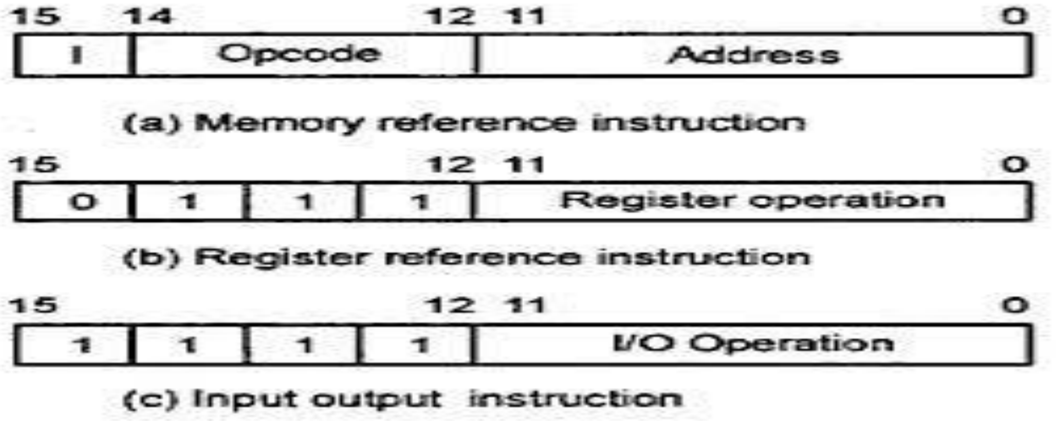
The connection of the registers and memory of the basic computer to a common bus system is shown in the figure. The oup8u of seven registers and memory are connected to the common bus. The specific output that is selected for the bus lines at any given time is determined from the binary value of the selection variables S2, S1 and S0. The number along each output shows the decimal equivalent of the required binary selection. The particular register whose LD (load) input is enabled receives the data from the bus during the next clock pulse transition.

The memory receives the contents of the bus when its write input is activated. The memory places its 16-bit output onto the bus when the read input is activated and S2S1S0=111.

The input register INPR and the output register OUTR has 8 bits each and communicate with the eight least significant bits in the bus. INPR is connected to provide information to the bus but OUTR can only receive information from the bus. Five registers have three control inputs: LD (load), INR (increment), and CLR (clear). The memory address is connected to AR. Thus, AR must always be used to specify a memory address.

2.1.1. COMPUTER INSTRUCTIONS

The basic computer has three instructions code formats, as shown in the following figures.



(a) Memory reference instruction

(b) Register reference instruction

(c) Input output instruction

Each format has 16 bits. 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 111 with a 0 in the leftmost bit of the instruction. A register-reference instruction specifies an operation on AC register. 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.

2.1.2. Instruction Set Completeness

A computer should have a set of instructions so that the use can construct machine language programs to evaluate any function that is known to be computable. The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

1. Arithmetic, logical, and shift instructions
2. Instructions for moving information to and from memory and processor registers
3. Program control instructions together with instructions that check status conditions
4. Input and output instructions

2.1.3. INSTRUCTION CYCLE

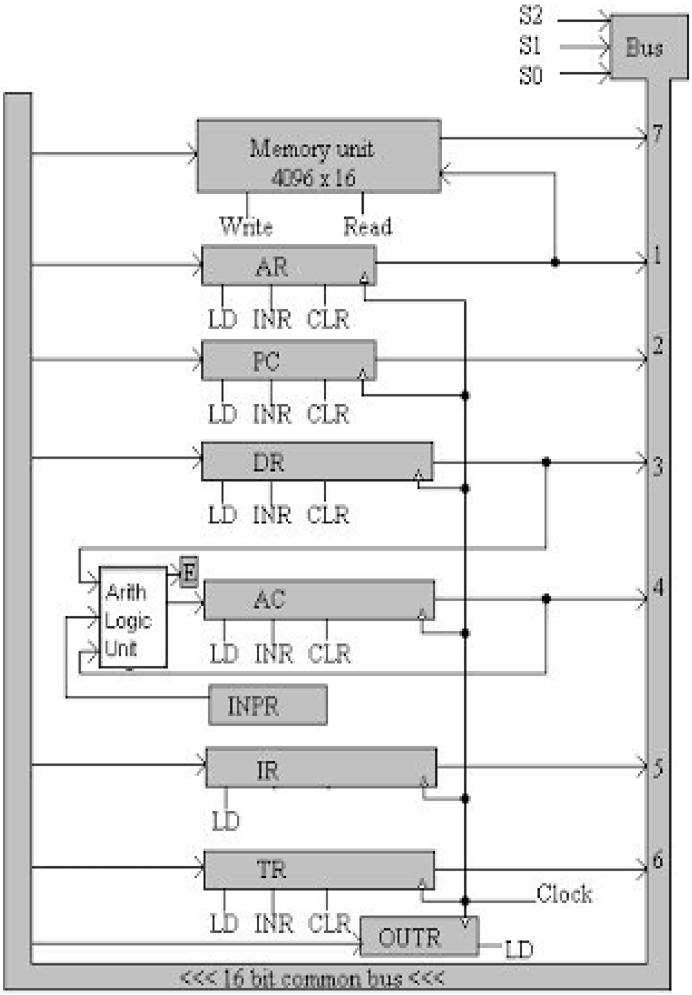
A program residing in the memory unit of the computer consists of a sequence of instructions. The program is executed in the computer by going through a cycle for each instruction. Each instruction cycle in turn is subdivided into a sequence of sub cycles or phases. In the basic computer each instruction 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.

2.1.4. 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 T0. After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T0, T1, T2 and so on. The micro operations for fetch and decode phases can be specified by the following register transfer statements.

T_0 : AR← PC

T_1 : IR ← M[AR], PC ← PC + 1

T_2 : D_0, D_1, ……, D_7 ← Decode IR(12-14), AR ← IR(0-11), I← IR(15)

Since only AR is connected to the address inputs of memory, it is necessary to transfer the address from PC to AR during the clock transition associated with timing signal T0. The instruction read from memory is then placed in the instruction register IR with the clock transition associated with timing signal T1. At the same time, PC is incremented by one to prepare it for the address of the next instruction in the program. At time T2, the operation code in IR is decoded, the indirect bit is transferred to flip-flop I, and the address part of the instruction is transferred to AR. SC is incremented after each clock pulse to produce the sequence T0,T1 and T2. The above figure shows the implementation of the first two register transfer statements.

To provide the data path for the transfer of PC to AR we must apply timing signal T_0 to achieve the following connection:

1. Place the content of PC onto the bus by making the bus selection inputs S2S1S0 equal to 010.
2. Transfer the content of the bust to AR by enabling the LD input of AR.

The next clock transition initiates the transfer from PC to AR since T_0 =1. In order to implement the second statement it is necessary to use timing signal T1 to provide the following connections in the bus system.

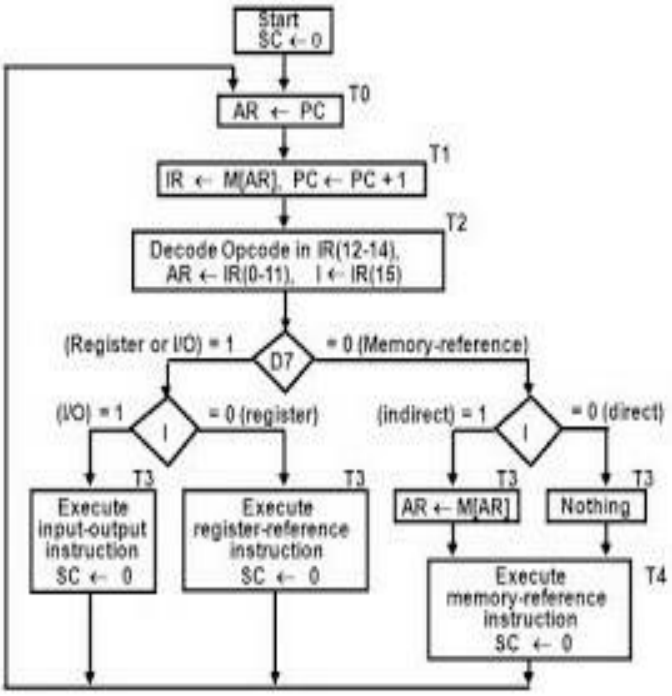
1. Enable the read input of memory.
2. Place the content of memory onto the bus by making S2S1S0 = 111.
3. Transfer the content of the bus to IR by enabling the LD input of IR.
4. Increment PC by enabling the INR input of PC.

The next clock transition initiates the read and increment operations since T1=1.

2.1.5. Determining 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 just read from memory. The following flowchart determines the instruction type after the decoding.

Decoding output D7 is equal to 1 if the operation code is equal to binary 111. From the above figure we can determine that if D7=1, the instruction must be a register-reference or input-output type. If D7=0, the operation code must be one of the other sever 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, we have a memory-reference instruction with an indirect address. It is then necessary to read the effective address from memory. The micro operation for the indirect address condition can be symbolized by the register transfer statement,


Start
SC ← 0
T0
AR ← PC
T1
IR ← M[AR], PC ← PC +1
T2
Decode Opcode in IR(12-14),
AR ← IR(0-11), I ← IR(15)
(Register or I/O) = 1
D7
= 0 (Memory-reference)
(I/O) = 1
I
= 0 (register)
(indirect) = 1
I
= 0 (direct)
T3
Execute input-output instruction SC ← 0
T3
Execute register-reference instruction SC ← 0
T3
AR ← M[AR]
T3
Nothing
T4
Execute memory-reference instruction SC ← 0

The three instruction types are subdivided into four separate paths. The selected operation is activated with the clock transition associated with timing signal T3. This can be symbolized as,

D_7' I T3: AR ← M[AR]
D_7' I' T3: Nothing
D_7 I' T3: Execute a register-reference instruction
D_7 I T3: Execute an input-output instruction

2.1.6.REGISTER-REFERENCE INSTRUCTIONS

Register-reference instructions are recognized by the control when D_7 =1 and I=0. These instructions use bits 0 through 11 of the instruction code to specify one of the 12 instructions. These 12 bits are available in IR(0-11). They are transferred to AR during time T2. Let D_7I'T3= r, which is the Boolean relation for the control function. The control function is distinguished by one of the bits in IR(0-11). By assigning the symbol B_i to bit I of IR, all control functions can be simply denoted by rB_i. The control functions and micro operations for the register-reference instructions are listed in the following table.

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

2.2. MEMORY-REFERENCE INSTRUCTIONS

1. AND to AC: This is an instruction that performs the AND logic operation on pairs of bits in AC and memory word specified by the effective address. The result of the operation is transferred to AC. The micro operations that execute this instruction are:

D_0T_4 : DR ← M[AR]

D_0T_5 : AC ← AC $\wedge$ DR, SC ←0

2. ADD to AC: This instruction adds the content of the memory word specified by the effective address to the value of AC. The sum is transferred into AC and the output carry C_{out} is transferred to the E (extended accumulator) flip-flop. The microoperations needed to execute this instruction are,

D_1T_4: DR ← M[AR]

D_1T_5: AC ← AC + DR, E ← C_{out}, SC ←0

3. LDA: Load to AC: This instruction transfers the memory word specified by the effective address to AC. The microoperations needed to execute this instruction are

D_2T_4: DR ← M[AR]

D_2T_5: AC ← DR, SC ←0

4. STA: Store AC: This instruction stores the content of AC into the memory word specified by the effective address. Since the output of AC is applied to the bus and the data input of memory is connected to the bus, we can execute this instruction with one micro operation:

D_3T_4: M[AR] ← AC, SC ←0

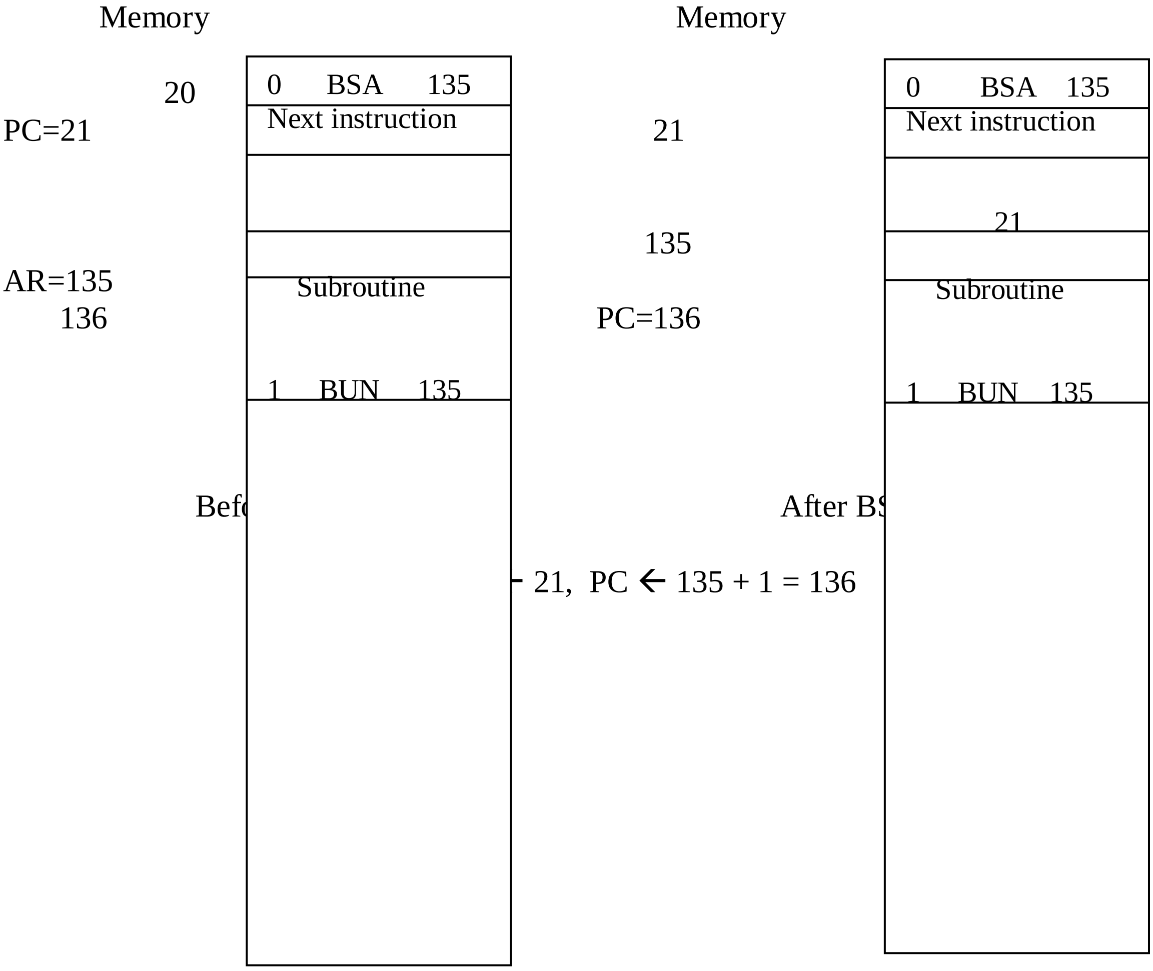
5. BUN: Branch Unconditionally: This instruction transfers the program to the instruction specified by the effective address. PC holds the address of the instruction to be read from memory in the next instruction cycle. PC is incremented at time T1 to prepare it for the address of the next instruction in the program sequence. The BUN instruction allows the programmer to specify an instruction out of the sequence, and the program branches (or jumps) unconditionally. The instruction is executed with one micro operation:

D_4T_4: PC ← AR, SC ← 0

6. BSA: Branch and Save Return Address: This instruction is useful for branching to a portion of the program called a subroutine or procedure. When executed, the BSA instruction stores the address of the next instruction in sequence into a memory location specified by the effective address. The effective address plus one is then transferred to PC to serve as the address of the first instruction in the subroutine. This operation was specified with the following register transfer:

M[AR] ← PC, PC ← AR + 1

A numerical example that demonstrates how this instruction is used with a subroutine is shown in the following figure.



It is not possible to perform the operation of the BSA instruction in one clock cycle when we use the bus system of the basic computer. To use the memory and the bus properly, the BSA instruction must be executed with a sequence of two micro operations:

$D_5 T_4$: M[AR] ← PC, AR ← AR + 1

D_5 T_5: PC ← AR, SC ← 0

7. ISZ: Increment and Skip if Zero: This instruction increments the word specified by the effective address, and if the increment value is equal to 0, PC is incremented by 1. The programmer usually stores a negative numbers (in 2's complement) in the memory word. As this negative number is repeatedly incremented by one, it eventually reaches the value of zero. At that time PC is increment by one in order to skip the next instruction in the program.

Since it is not possible to increment a word inside the memory, it is necessary to read the word into DR, increment DR, and store the word back into memory. This is done with the following sequence of microoperations

D_6T_4: DR ← M[AR]

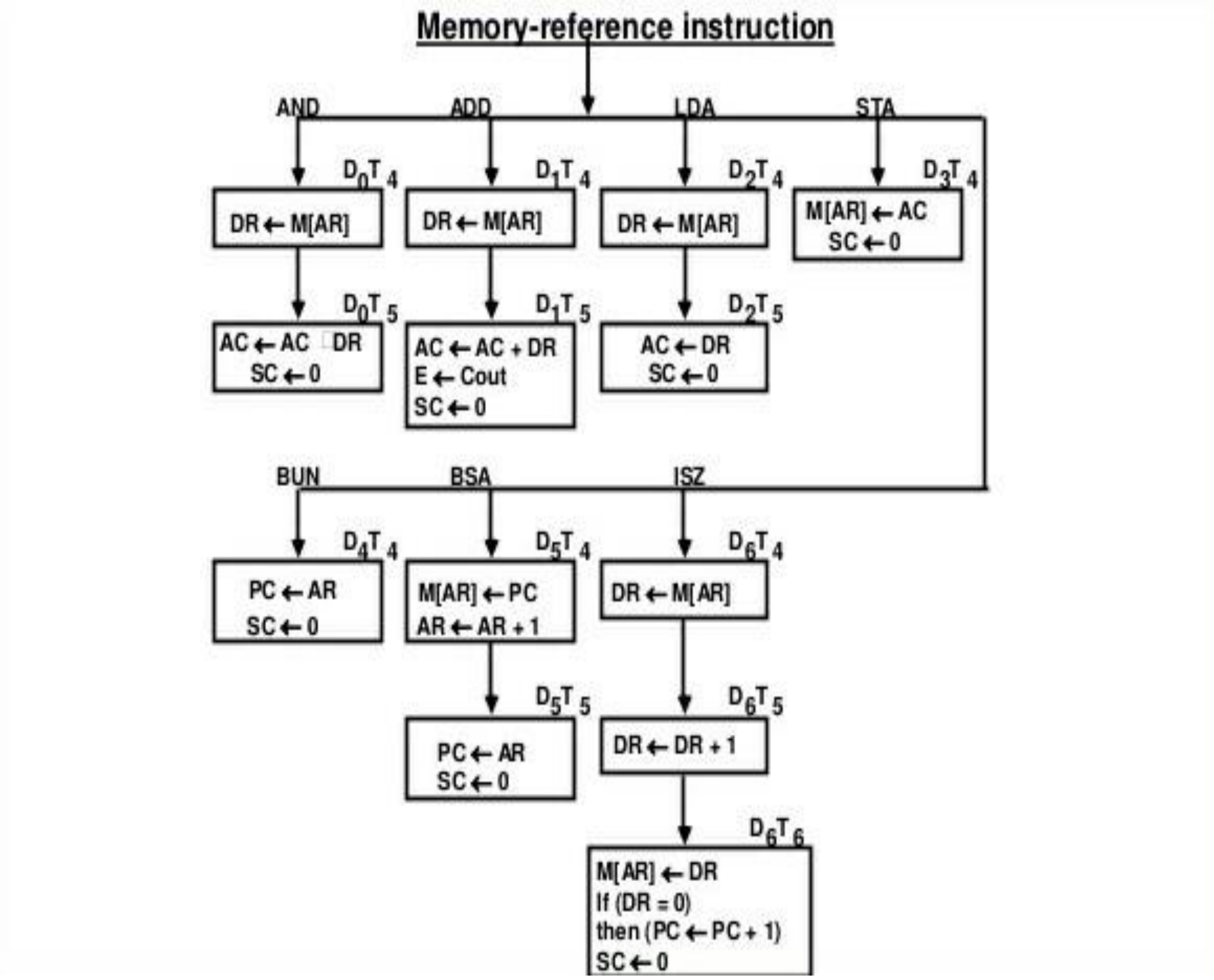
D_6T_5: DR ← DR + 1

D_6T_6: M[AR] ← DR, if (DR = 0) then (PC ← PC + 1), SC ← 0

2.2.1. Control Flowchart

The following flowchart shows all micro operations for execution of the seven memory-reference instructions.

Memory-reference instruction



2.2.2.INPUT-OUTPUT INSTRUCTIONS

Input-output instructions are needed for transferring information to and from AC register, for checking the flag bits, and for controlling the interrupt facility. Input-output instructions have an operation code 1111 and are recognized by the control when $D_7 = 1$ and I=1. The remaining bits of the instruction specify the particular operation. Let $D_7IT_3 = p$, which is the control function for the Boolean relation. The control function is distinguished by one of the bits in IR(6-11). By assigning the symbol B_i to bit I of IR, all control functions can be denoted by pB_i, for I=6 through 11. The control functions and micro operations for the input-output instructions are listed in the following table.

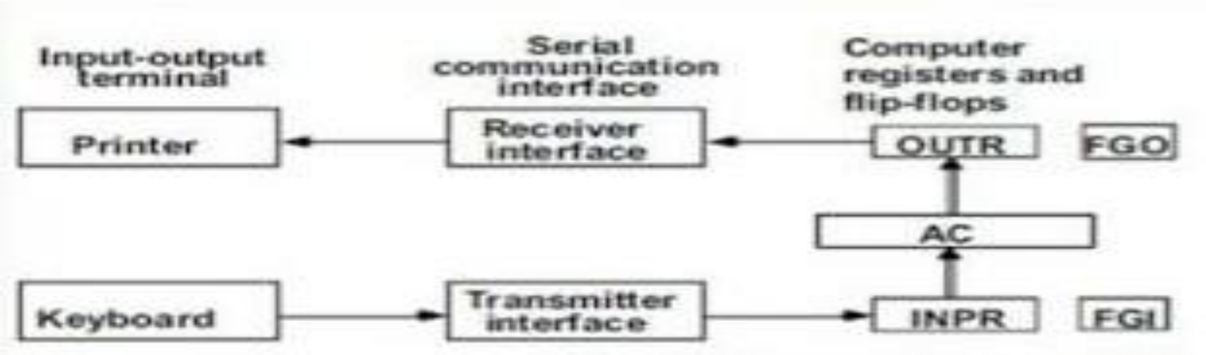
Symbol	Control Function	Symbolic Description	Meaning
INP	pB_{11}	AC(0-7)←INPR, FGI←0	Input character
OUT	pB_{10}	OUTR←AC(0-7), FGO←0	Output character
SKI	pB_9	If(FGI=1) then (PC←PC +1)	Skip on input flag
SKO	pB_8	If(FGO=1) then (PC←PC +1)	Skip on output flag

ION	pB_7	IEN←1	Interrupt enable on
IOF	pB_6	IEN←0	Interrupt enable off

2.3. INPUT-OUTPUT AND INTERRUPT

2.3.1. Input-Output Configuration

The terminal sends and receives serial information. Each quality of information has eight bits of an alphanumeric code. The serial information from the keyboard is shifted into the input register INPR. The serial information for the printer is stored in the output register OUTR. These two registers communicate with a communication interface serially and with the AC in parallel. The input-output configuration is shown in the following figure.



The input register INPR consists of eight bits and holds alphanumeric input information. The 1-bit input flag FGI is a control flip-flop. The flag bit is set to 1 when new information is available in the input device and is cleared to 0 when the information is accepted by the computer. The flag is needed to synchronize the timing rate difference between the input device and the computer.

The process of information transfer is as follows. Initially, the input flag FGI is cleared to 0. when a key is stuck in the keyboard, an 8-bit alphanumeric code is shifted into INPR and the input flag FGI is set to 1. as long as the flag is set, the information in INPR cannot be changed by striking another key.

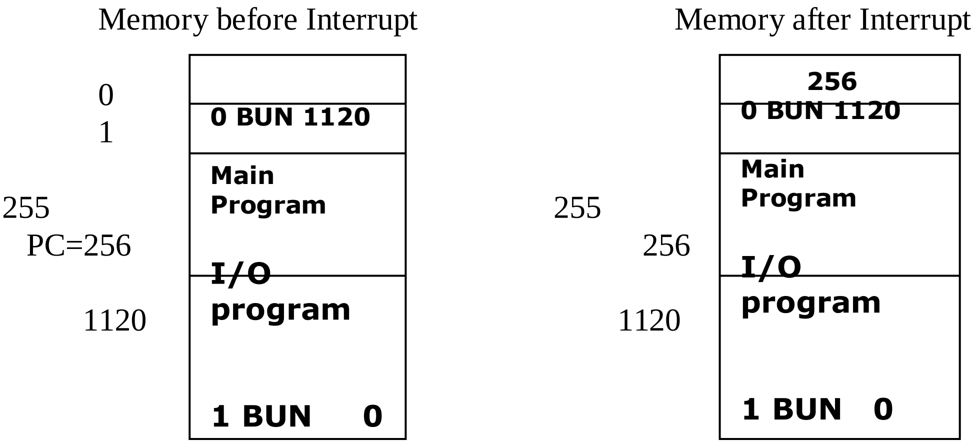
The OUTR works similarly but the direction of information flow is reversed. Initially, the output flag FGO is set to 1. Thus, the computer transfers the information from AC to OUTR and FGO is cleared to 0. The printer accepts the coded information, prints the corresponding character, and when the operation is completed, it sets FGO to 1.

2.3.2. I/O Interrupt

Assume that a computer can execute an instruction cycle in 1 μs. Assume that the input-output device can transfer information at a maximum rate of 10 characters per second. This is equivalent to 1 character every 1,00,000 μs. Two instructions are executed when the computer checks the flag bit and decides not to transfer the information. This means that at the maximum rate, the computer will check the flag 50,000 times between each transfer. The processor is wasting time while checking the flag instead of doing some other useful processing task.

To avoid this, an I/O interrupt can be sent by the external device. While the computer is running a program, it does not check the flags. However, when a flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed that a flag has been set. Then the processor stops its current execution temporarily to take care of the input or output transfer. The following flowchart shows how an interrupt cycle occurs.

An interrupt flip-flop R is included in the computer. When R=0, the computer goes through an instruction cycle. During the execute phase of the instruction cycle IEN is checked by the control. If it is 0, it indicates that the programmer does not want to use the interrupt, so control continues with the next instruction cycle. If IEN is 1, control checks the flag bits. if both flags are 0, it indicates that neither the input nor the output registers are ready for transfer of information. In this case, control continues with the next instruction cycle. If either flag is set to 1 while IEN=1, flip-flop R is set to 1. An example that shows what happens during the interrupt cycle is shown in the following figure.



Memory before Interrupt

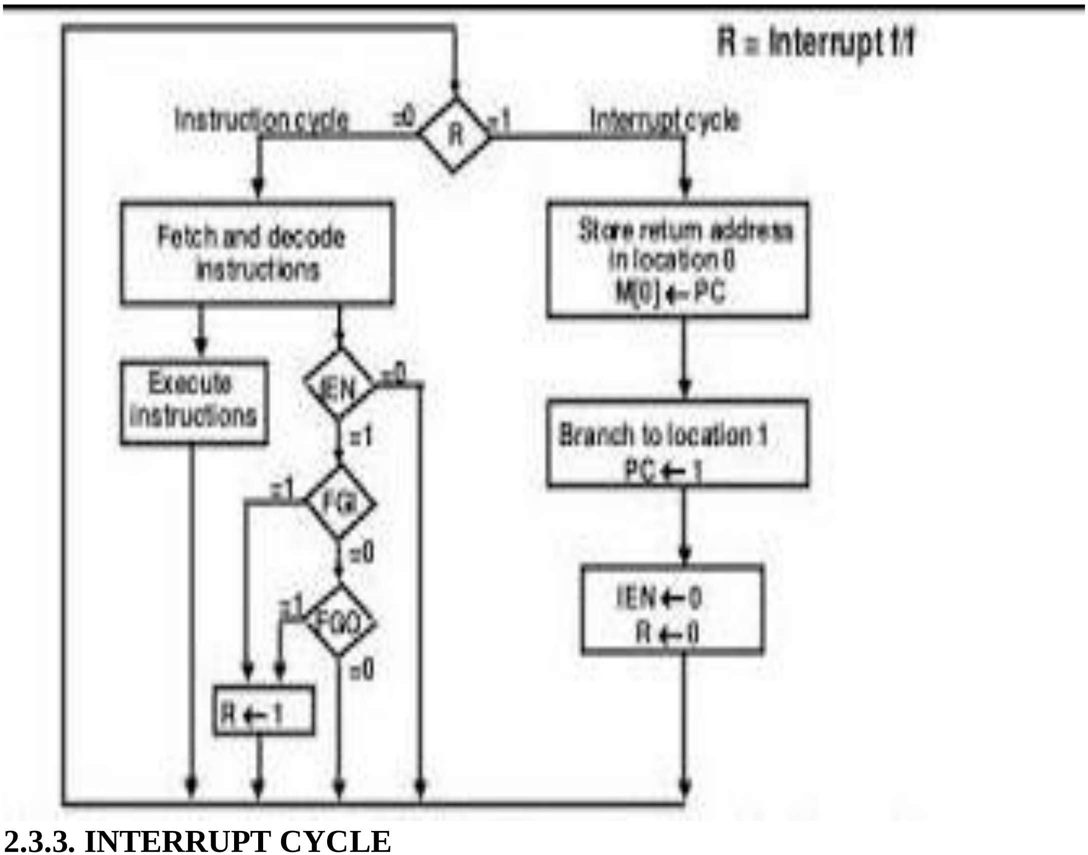
0
1
255
PC=256
1120

0 BUN 1120
Main Program
I/O program
1 BUN 0

Memory after Interrupt

255
256
1120

256
0 BUN 1120
Main Program
I/O program
1 BUN 0



2.3.3. INTERRUPT CYCLE

The interrupt cycle is initiated after the last execute phase if the interrupt flip-flop R is equal to 1. This flip-flop is set to 1 if IEN=1 and either FGI or FGO are equal to 1. This can happen with any clock transition except when timing signals T0, T1, or T2 are active. The condition for setting flip-flop R to 1 can be expressed with the following register transfer statement:

$T_0'T_1'T_2'$ (IEN)(FGI+FGO): $R \leftarrow 1$

The register transfer statements for the interrupt cycle can be expressed as,

RT_0: $AR \leftarrow 0$, $TR \leftarrow PC$
RT_1: $M[AR] \leftarrow TR$, $PC \leftarrow 0$
RT_2: $PC \leftarrow PC + 1$, $IEN \leftarrow 0$, $R \leftarrow 0$, $SC \leftarrow 0$

2.4. ADDRESSING MODES

Computers use addressing mode techniques for the purpose of accommodating one or both of the following provisions:

1. To give programming flexibility to the user by providing such facilities as pointers to memory, counters for loop control, indexing of data, and program relocation.
2. To reduce the number of bits in the addressing field of the instruction.

In some computers the addressing mode of the instruction is specified with a distinct binary code. Other computers use a single binary code that designates both the operation and the mode of the instruction. An example of an instruction format with a distinct addressing mode field is shown in the following figure.

Opcode	Mode	Address

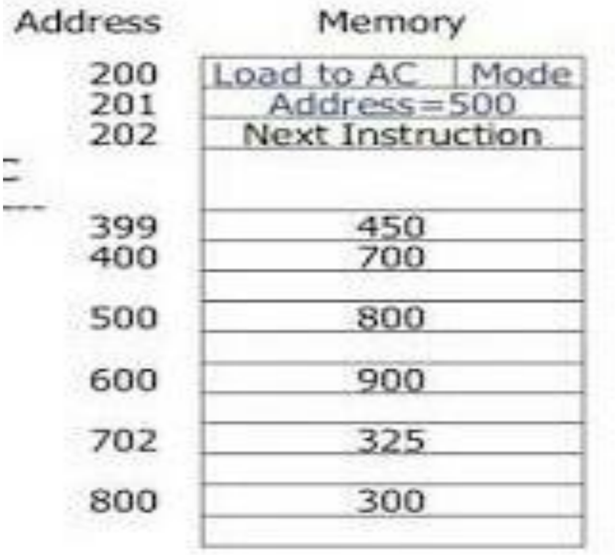
The 'Mode' specifies any one of the different Addressing Modes. They are,

1. Implied Mode: In this mode the operands are specified implicitly in the definition of the instruction. For e.g., the instruction "Complement Accumulator" is an implied mode instruction, because the operand in the Accumulator register because the operand in the Accumulator register is implied in the definition of the instruction. Thus, all register-reference instructions that use an Accumulator are implied-mode instructions. Zero-address instructions in a stack-organized computer are implied-mode instructions, since the operands are implied to be on top of the Stack.
2. Immediate Mode: In this mode the operand is specified in the instruction itself.
3. Register Mode: In this mode the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction. A k-bit field can specify any one of 2^k registers.
4. Register Indirect Mode: In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory. The advantage of a Register Indirect Mode instruction is that the address field of the instruction uses fewer bits to select a register than a memory address directly.
5. Auto-increment or Auto-decrement Mode: This is similar to the register indirect mode except that the register is incremented after its value is used to access memory in case of Auto-increment and the register is decremented before its value is used to access memory in case of Auto-decrement. When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table. The effective address is defined to be the memory address obtained from the computation directed by the given addressing mode
6. Direct Address Mode: In this mode the effective address is equal to the address part of the instruction.

7. Indirect Address Mode: In this mode the address field of the instruction gives the address where the effective address is stored in memory.
8. Relative Address Mode: In this mode the content of the PC is added to the address part of the instruction in order to obtain the effective address.
9. Indexed Addressing Mode: In this mode the content of an Index Register is added to the address part of the instruction to obtain the effective address. The Index Register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory. Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stored in the Index Register. Any operand in the array can be accessed with the same instruction provided that the index register contains the correct index value. The index register can be incremented to facilitate access to consecutive operands.
10. Base Register Addressing Mode: In this mode the content of a base register is added to the address part of the instruction to obtain the effective address. A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address. The Base Register Addressing mode is used in computer to facilitate the relocation of programs in memory. When programs and data are moved form one segment of memory to another, as required in multiprogramming systems, the address values of instructions must reflect this change of position. With a base register, the displacement values of instructions do not have to change. Only the value of the base register requires updating to reflect the beginning of a new memory segment.

2.4.1. Numerical Example

To show the differences between the various modes, let us assume the state of the memory and various processor registers as shown below.



Address	Memory
200	Load to AC | Mode
201	Address=500
202	Next Instruction
399	450
400	700
500	800
600	900
702	325
800	300

Now consider the following table, which specifies the Effective Address Obtained from the different Addressing Modes and the Contents of AC.

Addressing Mode	Effective	Content of AC

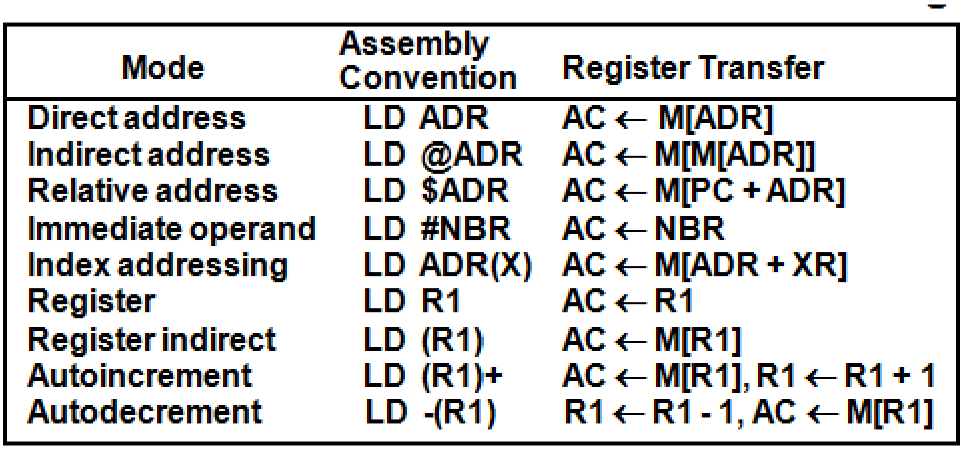
	Address	
Direct Address	500	800
Immediate Operand	200	500
Indirect Address	800	300
Relative Address	702	325
Indexed Address	600	900
Register	--	400
Register Indirect	400	700
Auto increment	400	700
Auto decrement	399	450

2.5. DATA TRANSFER INSTRUCTIONS

The Typical Data Transfer Instructions are shown in the table below.

Name	Mnemonic
Load	**LD**
Store	**ST**
Move	**MOV**
Exchange	**XCH**
Input	**IN**
Output	**OUT**
Push	**PUSH**
Pop	**POP**

The Data Transfer Instructions with Different Addressing Modes are shown in table below.



Mode	Assembly Convention	Register Transfer
Direct address	LD ADR	AC ← M[ADR]
Indirect address	LD @ADR	AC ← M[M[ADR]]
Relative address	LD $ADR	AC ← M[PC + ADR]
Immediate operand	LD #NBR	AC ← NBR
Index addressing	LD ADR(X)	AC ← M[ADR + XR]
Register	LD R1	AC ← R1
Register indirect	LD (R1)	AC ← M[R1]
Autoincrement	LD (R1)+	AC ← M[R1], R1 ← R1 + 1
Autodecrement	LD -(R1)	R1 ← R1 - 1, AC ← M[R1]

2.5.1.DATA MANIPULATION INSTRUCTIONS

The three Basic Types of instructions are

(1) Arithmetic instructions

(2) Logical and bit manipulation instructions

(3) Shift instructions

2.5.2. Arithmetic Instructions

The basic arithmetic operations are Add, Subtract, Increment, and Decrement etc. The following table shows the operation, its representation and description.

Operation	Representation	Description
Add	R3 ← R1 + R2	Contents of R1 and R2 are added and the result is transferred to R3
Subtract	R3 ← R1 – R2	Contents of R2 are subtracted from contents of R1 and the result is transferred to R3
1's Complement	$\overline{R}1$	Complement the content of R1
2's Complement	$\overline{R}1$ +1	Complement the contents of R1 and add 1 in it.
2's Complement subtraction	R3 ← R1 + $\overline{R}2$ + 1	Add R1 and the 2's Complement of R2
Increment	R1 ← R1 +1	Increment the contents of R1 by one
Decrement	R1 ← R1 –1	Decrement the contents of R1 by one

Unit-III

Control Memory

3.7. Control Unit

3.7.1. Control Memory

- The control unit in a digital computer initiates sequences of microoperations
- The complexity of the digital system is derived form the number of sequences that are performed
- When the control signals are generated by hardware, it is *hardwired*
- In a bus-oriented system, the control signals that specify microoperations are groups of bits that select the paths in multiplexers, decoders, and ALUs.
- The control unit initiates a series of sequential steps of microoperations
- The control variables can be represented by a string of 1's and 0's called a *control word*
- A microprogrammed *control unit* is a control unit whose binary control variables are stored in memory
- A sequence of microinstructions constitutes a *microprogram*
- The control memory can be a read-only memory
- *Dynamic* microprogramming permits a microprogram to be loaded and uses a writable control memory
- A computer with a microprogrammed control unit will have two separate memories: a main memory and a control memory
- The microprogram consists of microinstructions that specify various internal control signals for execution of register microoperations
- These microinstructions generate the microoperations to:
  - o fetch the instruction from main memory
  - o evaluate the effective address
  - o execute the operation
  - o return control to the fetch phase for the next instruction
- The control memory address register specifies the address of the microinstruction
- The control data register holds the microinstruction read from memory
- The microinstruction contains a control word that specifies one or more

microoperations for the data processor

- The location for the next microinstruction may, or may not be the next in sequence
- Some bits of the present microinstruction control the generation of the address of the next microinstruction
- The next address may also be a function of external input conditions
- While the microoperations are being executed, the next address is computed in the next address generator circuit (sequencer) and then transferred into the CAR to read the next microinstructions
- Typical functions of a sequencer are:

  incrementing the CAR by one

  o loading into the CAR and address from control memory

  o transferring an external address

  o loading an initial address to start the control operations

- A clock is applied to the CAR and the control word and next-address information are taken directly from the control memory
- The address value is the input for the ROM and the control work is the output
- No read signal is required for the ROM as in a RAM
- The main advantage of the micro programmed control is that once the hardware configuration is established, there should be no need for h/w or wiring changes
- To establish a different control sequence, specify a different set of microinstructions for control memory

3.8. Address Sequencing

- Microinstructions are stored in control memory in groups, with each group specifying a *routine*
- Each computer instruction has its own microprogram routine to generate the microoperations
- The hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another
- Steps the control must undergo during the execution of a single

computer instruction:

o Load an initial address into the CAR when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction fetch routine – IR holds instruction

o The control memory then goes through the routine to determine the effective address of the operand – AR holds operand address

o The next step is to generate the microoperations that execute the instruction by considering the opcode and applying a *mapping*

o After execution, control must return to the fetch routine by executing an unconditional branch

• The microinstruction in control memory contains a set of bits to initiate microoperations in computer registers and other bits to specify the method by which the next address is obtained

• Conditional branching is obtained by using part of the microinstruction to select a specific status bit in order to determine its condition

• The status conditions are special bits in the system that provide parameter information such as the carry-out of an adder, the sign bit of a number, the mode bits of an instruction, and i/o status conditions

• The status bits, together with the field in the microinstruction that specifies a branch address, control the branch logic

• The branch logic tests the condition, if met then branches, otherwise, increments the CAR

• If there are 8 status bit conditions, then 3 bits in the microinstruction are used to specify the condition and provide the selection variables for the multiplexer

• For unconditional branching, fix the value of one status bit to be one load the branch address from control memory into the CAR

• A special type of branch exists when a microinstruction specifies a branch to the first word in control memory where a microprogram routine is located

• The status bits for this type of branch are the bits in the opcode

• Assume an opcode of four bits and a control memory of 128 locations

• The mapping process converts the 4-bit opcode to a 7-bit address for control memory

- This provides for each computer instruction a microprogram routine with a capacity of four microinstructions
- Subroutines are programs that are used by other routines to accomplish a particular task and can be called from any point within the main body of the microprogram
- Frequently many microprograms contain identical section of code
- Microinstructions can be saved by employing subroutines that use common sections of microcode
- Microprograms that use subroutines must have a provisions for storing the return address during a subroutine call and restoring the address during a subroutine return
- A subroutine register is used as the source and destination for the addresses

3.9. Micro program Example

- The process of code generation for the control memory is called *microprogramming*
- Two memory units:
    - o Main memory – stores instructions and data
    - o Control memory – stores microprogram
- Four processor registers
    - o Program counter – PC
    - o Address register – AR
    - o Data register – DR
    - o Accumulator register - AC
- Two control unit registers
    - o Control address register – CAR
    - o Subroutine register – SBR
- Transfer of information among registers in the processor is through MUXs rather than a bus
- Three fields for an instruction:
    - o 1-bit field for indirect addressing
    - o 4-bit opcode
    - o 11-bit address field
- The example will only consider the following 4 of the possible 16 memory

instructions

ADD 0000 AC ← AC + M[EA]

BRANCH 0001 If (AC < 0) then (PC ← EA)

STORE 0010 M[EA] ← AC

EXCHANGE 0011 AC ← M[EA], M[EA] ← AC

• The microinstruction format is composed of 20 bits with four parts to it

o Three fields F1, F2, and F3 specify microoperations for the computer [3 bits each]

o The CD field selects status bit conditions [2 bits]

o The BR field specifies the type of branch to be used [2 bits]

o The AD field contains a branch address [7 bits]

• Each of the three microoperation fields can specify one of seven possibilities

• No more than three microoperations can be chosen for a microinstruction

• If fewer than three are needed, the code 000 = NOP

DR ← M[AR], PC ← PC +1 F2 = 100 and F3 = 101

F1 F2 F3 = 000 100 101

• Five letters to specify a transfer-type microoperation

o First two designate the source register

o Third is a 'T'

o Last two designate the destination register

AC ← DR F1 = 100 = DRTAC

• The condition field is two bits to specify four status bit conditions

00 Always = 1 U Unconditional branch

01 DR(15) I Indirect address bit

10 AC(15) S Sign bit of AC

11 AC = 0 Z Zero value in AC

• The branch field is two bits and is used with the address field to choose the address of the next microinstruction

00 JMP CAR ← AD if condition =1

CAR ← CAR + 1 else

01 CALL CAR ← AD, SBR ← CAR + 1 if cond. =1

CAR ← CAR +1 else

10 RET CAR ← SBR

11 MAP CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0

- Each line of an assembly language microprogram defines a symbolic microinstruction and is divided into five parts

    1. The label field may be empty or it may specify a symbolic address. Terminate with a colon
    2. The microoperations field consists of 1-3 symbols, separated by commas. Only one symbol from each F field. If NOP, then translated to 9 zeros
    3. The condition field specifies one of the four conditions
    4. The branch field has one of the four branch symbols
    5. The address field has three formats

        a. A symbolic address – must also be a label

        b. The symbol NEXT to designate the next address in sequence

        c. Empty if the branch field is RET or MAP and is converted to 7 zeros

- The symbol ORG defines the first address of a microprogram routine

ORG 64 – places first microinstruction at control memory 1000000

- The control memory has 128 locations, each one is 20 bits
- The first 64 locations are occupied by the routines for the 16 instructions, addresses 0-63
- Can start the fetch routine at address 64
- The fetch routine requires the following three microinstructions (locations 64-66)

AR ← PC
DR ← M[AR], PC ← PC +1
AR ← DR(0-10), CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0
ORG 64
Fetch: PCTAR U JMP NEXT
READ, INCPC U JMP NEXT
DRTAR U MAP

Address	F1	F2	F3	CD	BR	AD
1000000	110	000	000	00	00	1000001
1000001	000	100	101	00	00	1000010
1000010	101	000	000	00	11	0000000

3.10. Design of Control Unit

The bits of the microinstruction are usually divided into fields, with each field defining a distinct, separate function. The various fields encountered in instruction formats provide control bits to initiate micro operations in the system, special bits to specify the way that the next address is to be evaluated, and an address field for branching. The number of control bits that initiate micro operation can be reduced by grouping mutually exclusive variables into fields and encoding the k bits in each field to provide 2 micro operations. Each field requires a decoder to produce the corresponding control signals. This method reduces the size of the microinstruction bits but requires additional hardware external to the control memory. It also increases the delay time of the control signals because they must propagate through the decoding circuits.

The encoding of control bits was demonstrated in the programming example of the preceding section. The nine bits of the microoperation field are divided into three subfields of three bits each. The control memory output of each subfield must be decoded to provide the distinct microoperations. The outputs of the decoders are connected to the appropriate inputs in the processor unit.

Below figure shows the three decoders and some of the connections that must be made from their outputs. Each of the three fields of the microinstruction presently available in the output of control memory are decoded with a 3 □ 8 decoder to provide eight outputs. Each of these outputs must be connected to the proper circuit to initiate the corresponding microoperation as specified in Table 7-1. For example, when FI = 101 (binary 5), the next clock pulse transition transfers the content of DR (0-10) to AR (symbolized by DRTAR in Table 7-1). Similarly, when F1 = 101 (binary 6) there is a transfer from PC to AR (symbolized by PCTAR). As shown in Fig. 4-7, outputs 5 and 6 of decoder F1 are connected to the load input of AR so that when either one of these outputs is active, information from the multiplexers is transferred to AR. The multiplexers select the information from DR when output 5 is active and from PC when output 5 in inactive. The transfer into AR occurs with a clock pulse transition only when output 5 or output 6 of the decoder are active. The other outputs of the decoders that initiate transfers between registers must be connected in a similar fashion.

The arithmetic logic shift unit can be designed, instead of using gates to generate the control signals marked by the symbols AND, ADD, and DR in Fig , these inputs will now come from the outputs of the decoders associated with the symbols AND, ADD, and DRTAC, respectively

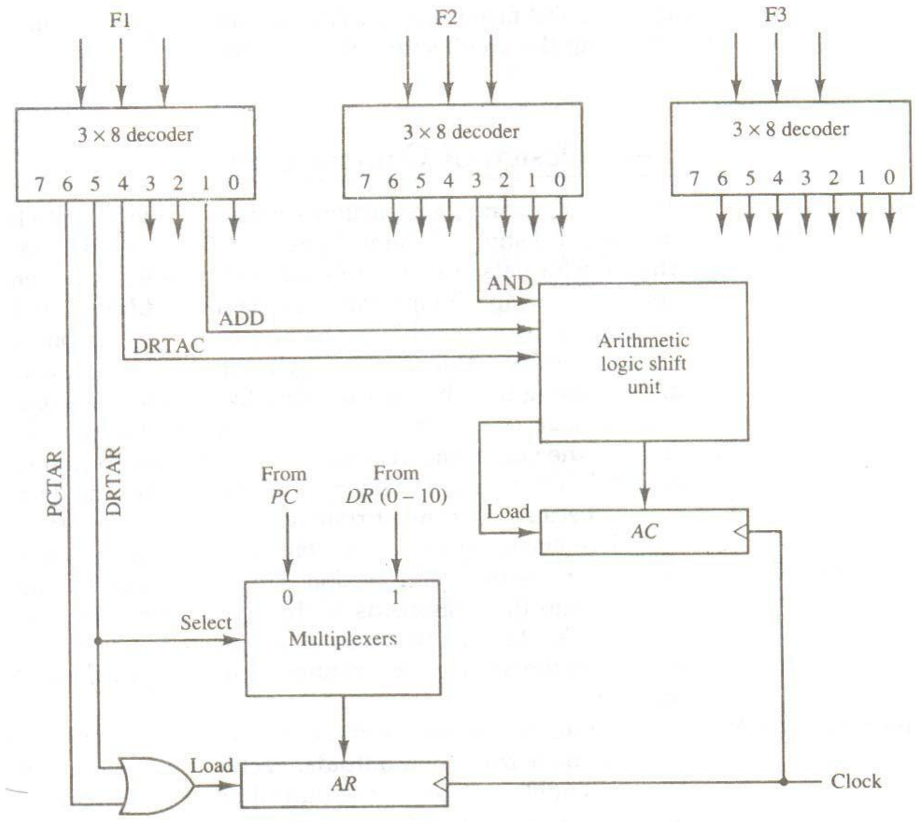


Figure : Decoding of micro operation fields

as shown in Fig. The other output of the decoders that are associated with an AC operation must also be connected to the arithmetic logic shift unit in a similar fashion.

2.7. COMPUTER ARITHMETIC

Data is manipulated by using the arithmetic instructions in digital computers. Data is manipulated to produce results necessary to give solution for the computation problems. The Addition, subtraction, multiplication and division are the four basic arithmetic operations. If we want then we can derive other operations by using these four operations.

To execute arithmetic operations there is a separate section called arithmetic processing unit in central processing unit. The arithmetic instructions are performed generally on binary or decimal data. Fixed-point numbers are used to represent integers or fractions. We can have signed or unsigned negative numbers. Fixed-point addition is the simplest arithmetic operation.

If we want to solve a problem then we use a sequence of well-defined steps. These steps are collectively called algorithm. To solve various problems we give algorithms. In order to solve the computational problems, arithmetic instructions are used in digital computers that manipulate data. These instructions perform arithmetic calculations.

And these instructions perform a great activity in processing data in a digital computer. As we already stated that with the four basic arithmetic operations addition, subtraction, multiplication and division, it is possible to derive other arithmetic operations and solve scientific problems by means of numerical analysis methods.

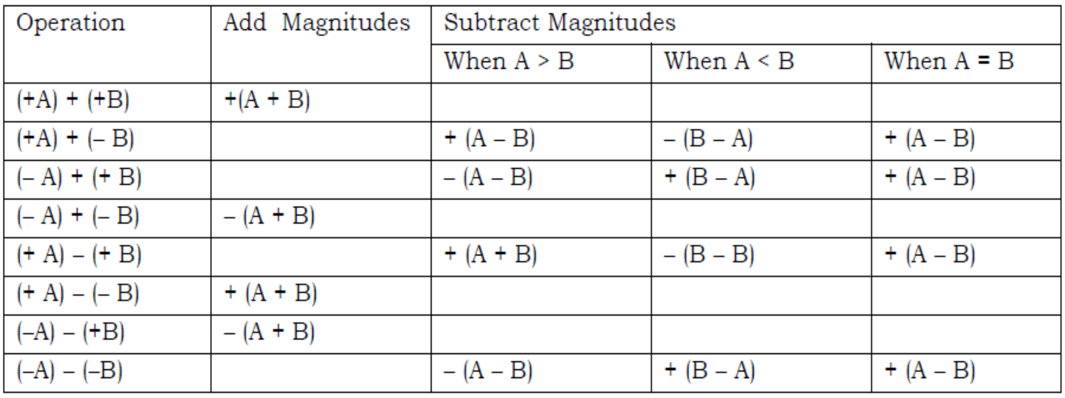
A processor has an arithmetic processor(as a sub part of it) that executes arithmetic operations. The data type, assumed to reside in processor, registers during the execution of an arithmetic instruction. Negative numbers may be in a signed magnitude or signed complement representation. There are three ways of representing negative fixed point - binary numbers signed magnitude, signed 1's complement or signed 2's complement. Most computers use the signed magnitude representation for the mantissa.

2.7.1. Addition and Subtraction with Signed –Magnitude Data

We designate the magnitude of the two numbers by A and B. Where the signed numbers are added or subtracted, we find that there are eight different conditions to consider, depending on the sign of the numbers and the operation performed. These conditions are listed in the first column of

Table 4.1. The other columns in the table show the actual operation to be performed with the magnitude of the numbers. The last column is needed to present a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0. The algorithms for addition and subtraction are derived from the table and can be stated as follows (the words parentheses should be used for the subtraction algorithm).

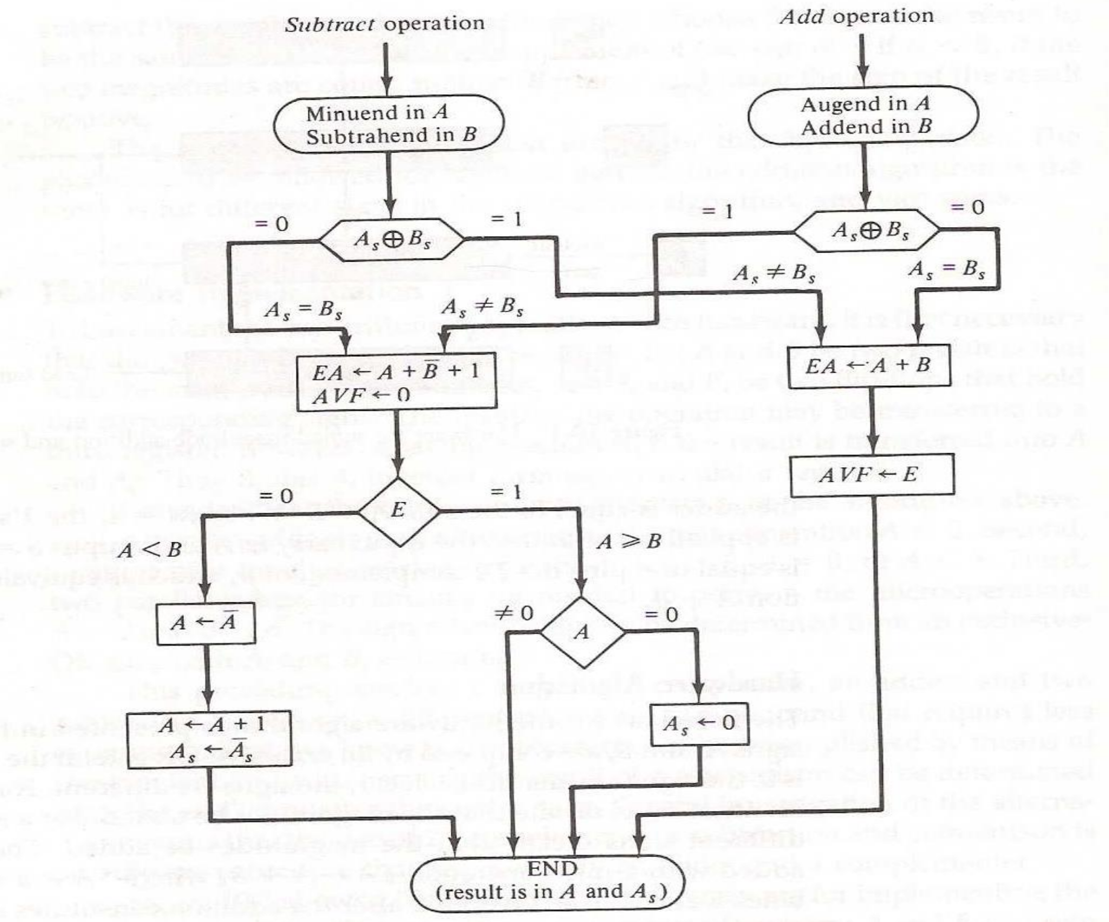
Table 4.1: Addition and Subtraction of Signed-Magnitude Numbers

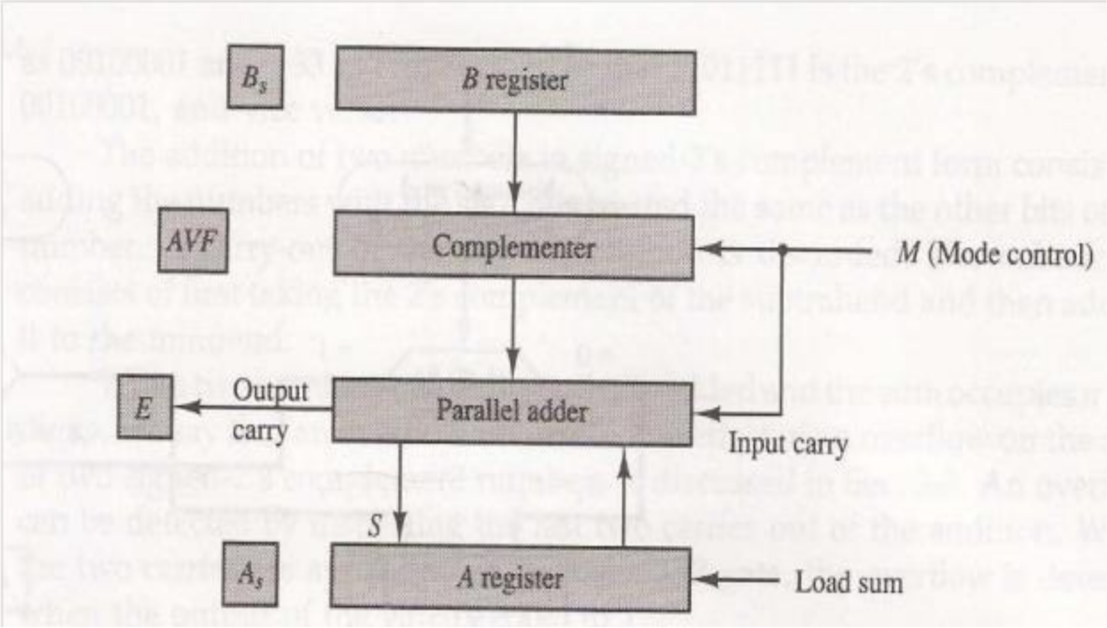


Operation	Add Magnitudes	Subtract Magnitudes		
		When A > B	When A < B	When A = B
(+A) + (+B)	+(A + B)			
(+A) + (– B)		+ (A – B)	– (B – A)	+ (A – B)
(– A) + (+ B)		– (A – B)	+ (B – A)	+ (A – B)
(– A) + (– B)	– (A + B)			
(+ A) – (+ B)		+ (A + B)	– (B – B)	+ (A – B)
(+ A) – (– B)	+ (A + B)			
(–A) – (+B)	– (A + B)			
(–A) – (–B)		– (A – B)	+ (B – A)	+ (A – B)

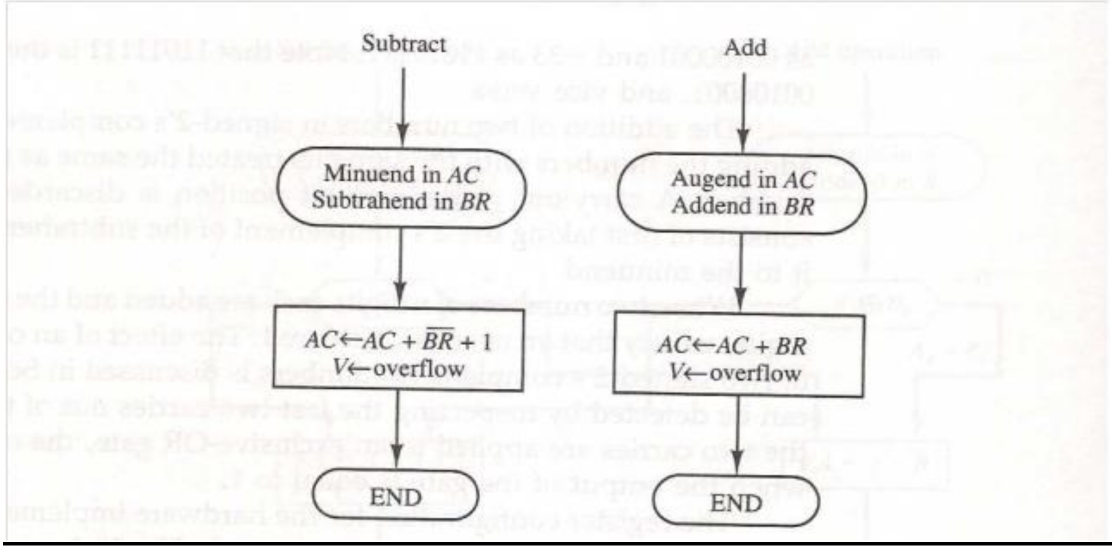
When the signs of A and B are same, add the two magnitudes and attach the sign of result is that of A. When the signs of A and B are not same, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A, if A > B or the complement of the sign of A if A < B. If the two magnitudes are equal, subtract B from A and make the sign of the result will be positive.

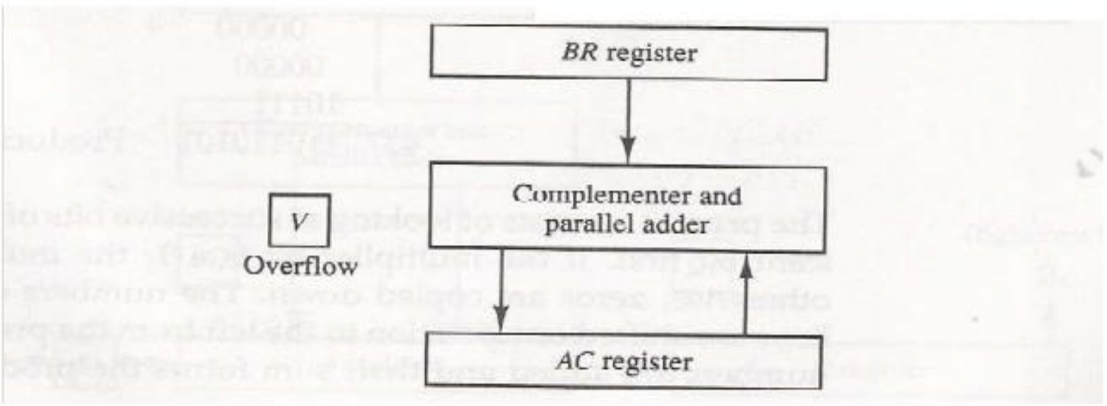
2.7.2. Hardware Implementation for Signed-Magnitude Data:





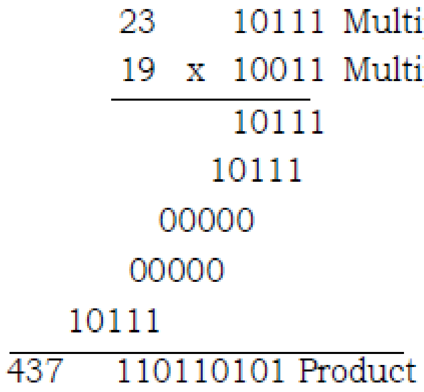
2.7.3. Addition and Subtraction with signed 2's Complement Data:





2.8. MULTIPLICATION ALGORITHMS:

Multiplication of two fixed-point binary numbers in signed magnitude representation is done with paper and pencil by a process of successive shift and add operations. This process is best illustrated with a numerical example:



```
        23      10111  Multiplicand
        19   x  10011  Multiplier
       ----------------
                10111
               10111
             00000
            00000
         10111
   --------------------
   437     110110101 Product
```

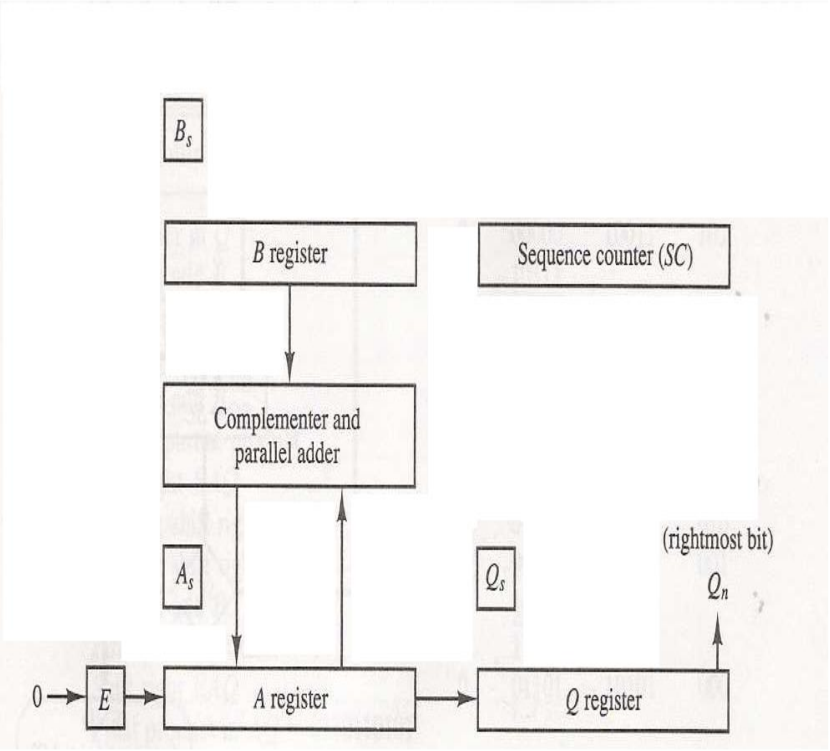
This process looks at successive bits of the multiplier, least significant bit first. If the multiplier bit is 1, the multiplicand is copied as it is; otherwise, we copy zeros. Now we shift numbers copied down one position to the left from the previous numbers. Finally, the numbers are added and their sum produces the product.

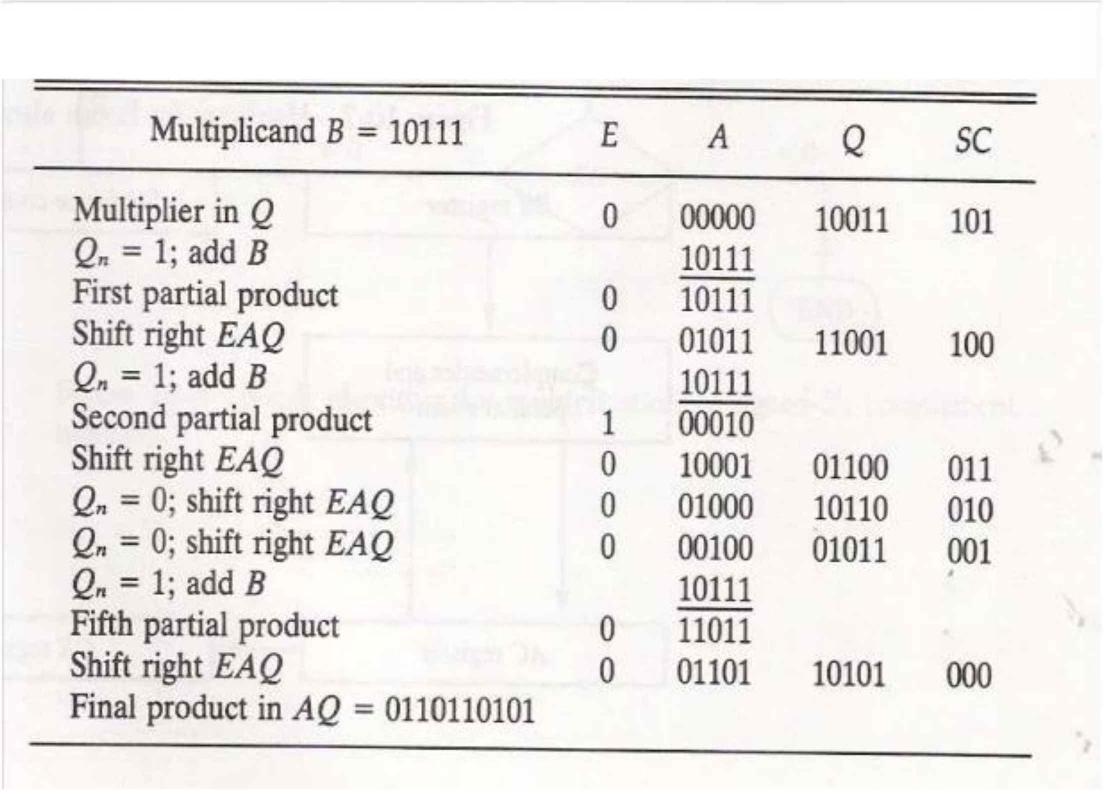
2.8.1. Hardware Implementation for Signed-Magnitude Data:

When multiplication is implemented in a digital computer, we change the process slightly. Here, instead of providing registers to store and add simultaneously as many binary numbers as there are bits in the multiplier, it is convenient to provide an adder for the summation of only two binary numbers, and successively accumulate the partial

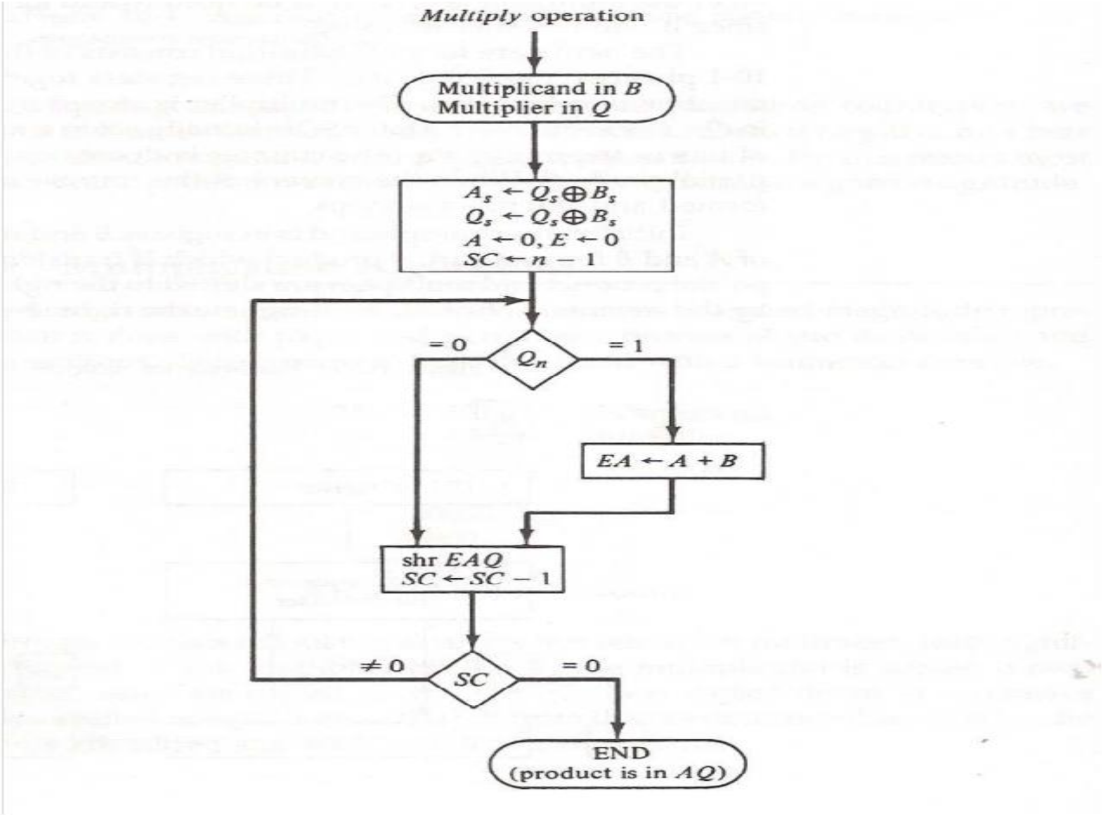
products in a register. Second, instead of shifting the multiplicand to left, the partial product is shifted to the right, which results in leaving the partial product and the multiplicand in the required relative positions. Now, when the corresponding bit of the multiplier is 0, there is no need to add all zeros to the partial product since it will not alter its value.

The hardware for multiplication consists of the equipment given in Figure .The multiplier is stored in the register and its sign in Qs. The sequence counter SC is initially set bits in the multiplier. After forming each partial product the counter is decremented. When the content of the counter reaches zero, the product is complete and we stop the process.





Multiplicand B = 10111	E	A	Q	SC
Multiplier in Q	0	00000	10011	101
$Q_n = 1$; add B		10111		
First partial product	0	10111		
Shift right EAQ	0	01011	11001	100
$Q_n = 1$; add B		10111		
Second partial product	1	00010		
Shift right EAQ	0	10001	01100	011
$Q_n = 0$; shift right EAQ	0	01000	10110	010
$Q_n = 0$; shift right EAQ	0	00100	01011	001
$Q_n = 1$; add B		10111		
Fifth partial product	0	11011		
Shift right EAQ	0	01101	10101	000
Final product in AQ = 0110110101				



2.8.2. Booth Multiplication Algorithm:

If the numbers are represented in signed 2’s complement then we can multiply them by using Booth algorithm. In fact the strings of 0's in the multiplier need no addition but just shifting, and a string of l's in the multiplier from bit weight 2k to weight 2m can be treated as 2k+1 - 2m. For example, the binary number 001111 (+15) has a string of 1's from 23 to 20(k = 3, m = 0).

The number can be represented as 2k+1 – 2m = 24- 20= 16 - 1 = 15. Therefore, the multiplication M x 14, where M is the multiplicand and 14 the multiplier may be computed as M x 24 - M x 21. That is, the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

Booth algorithm needs examination of the multiplier bits and shifting of the partial product. Prior to the shifting, the multiplicand added to the partial product, subtracted from the partial product, or left unchanged by the following rules:

1. The multiplicand is subtracted from the partial product when we get the first least significant 1 in a string of 1's in the multiplier.

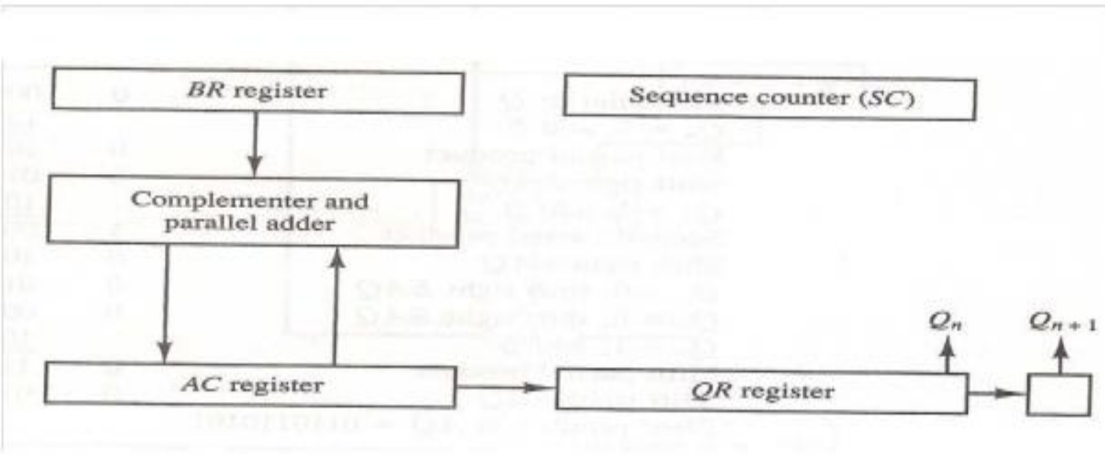
2. The multiplicand is added to the partial product when we get the first Q (provided that there was a previous 1) in a string of 0's in the multiplier.

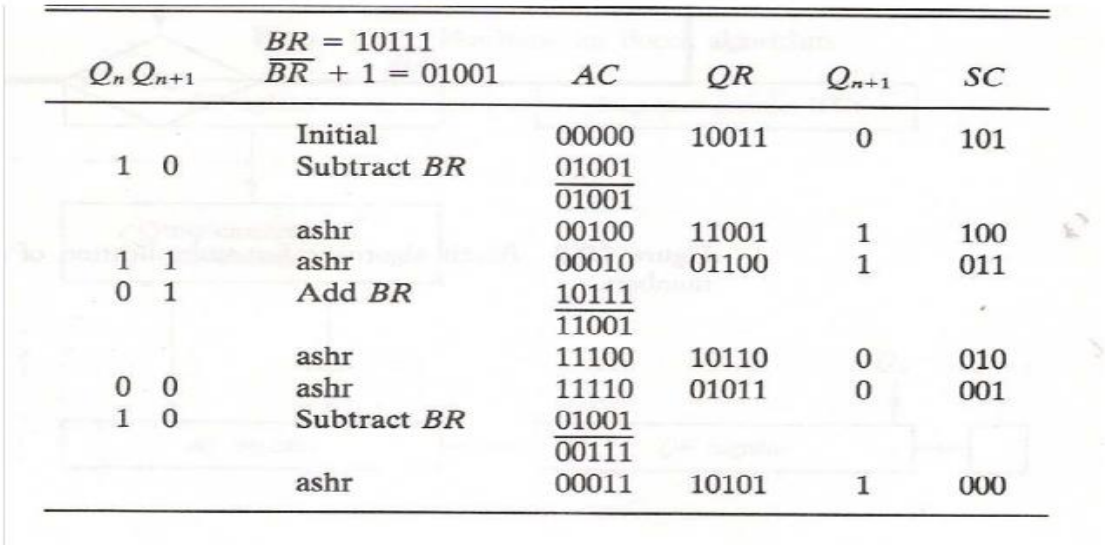
3. The partial product does not change when the multiplier bit is the same as the previous multiplier bit.

The algorithm applies to both positive and negative multipliers in 2's complement representation. This is because a negative multiplier ends with a string of l's and the last operation will be a subtraction of the appropriate weight. For example, a multiplier equal to -14 is represented in 2's complement as 110010 and is treated as -24 + 22 – 21 = -14.

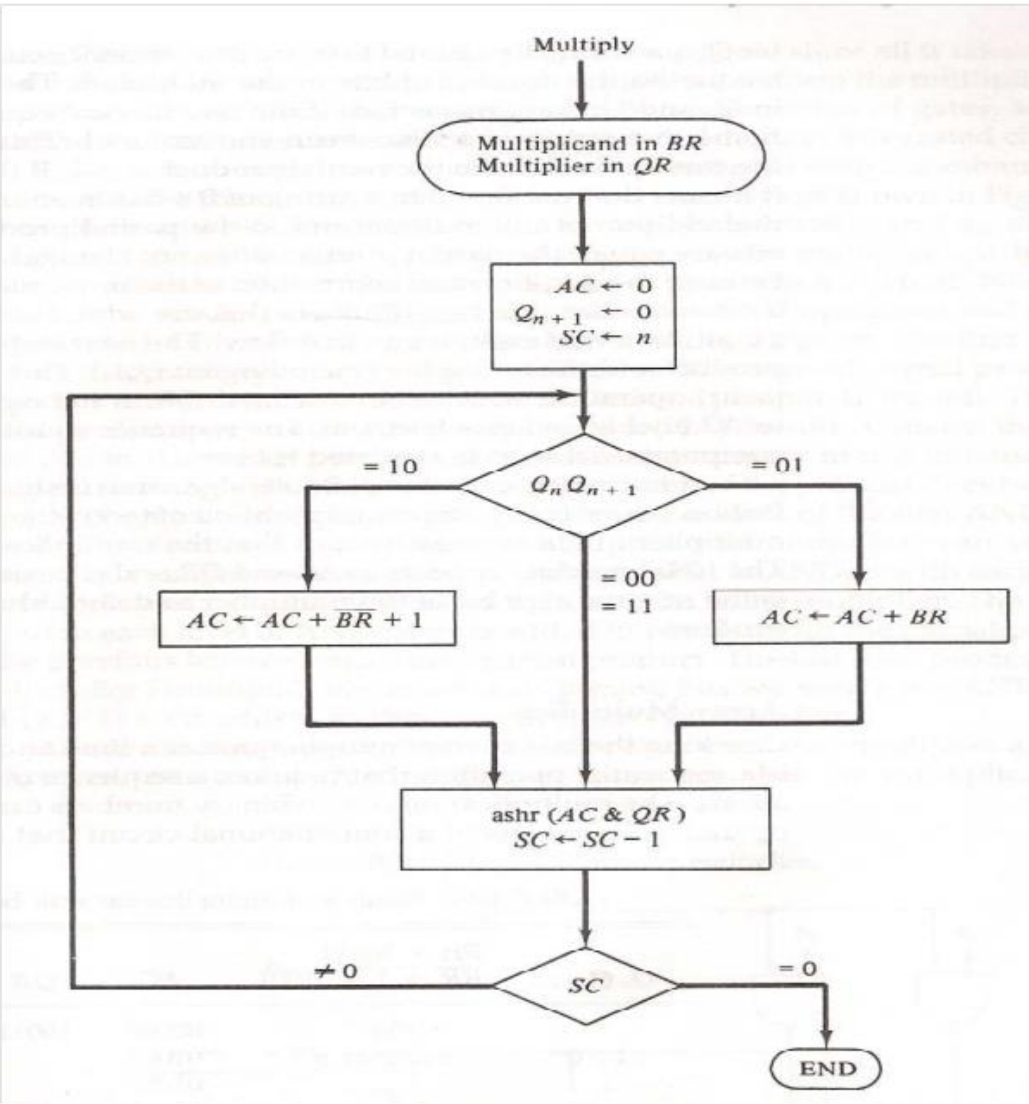
The hardware implementation of Booth algorithm requires the register configuration shown in Fig. Qn represents the least significant bit of the multiplier in register QR. An extra flip-flop Qn+1 is appended to QR to provide a double bit inspection of the multiplier. The flowchart for Booth algorithm is shown in Figure 4.7(b). AC and the appended bit Qn+1 are initially set to 0 and the sequence counter SC is set to a number n equal to the number of bits in the multiplier. The two bits of the multiplier in Qn and Qn+1 are inspected. If the two bits are 10, it means that the first 1 in a string of 1's has been encountered. This needs a subtraction of the multiplicand from the partial product in AC. If the two bits are equal to 01. It means that the first 0 in a string of 0's has been encountered. This needs the addition of the multiplicand to the partial product in AC. When the two bits are equal, the partial product does not change. An overflow cannot occur because the addition and subtraction of the multiplicand follow each other. Hence, the two numbers that are added always have opposite sign, a condition that excludes an overflow. Next step is to shift right the partial product and the multiplier (including bit Qn+1). This is an arithmetic shift right (ashr) operation which shifts AC and QR to the right and leaves the sign bit in AC same The sequence counter decrements and the computational loop is repeated n times.

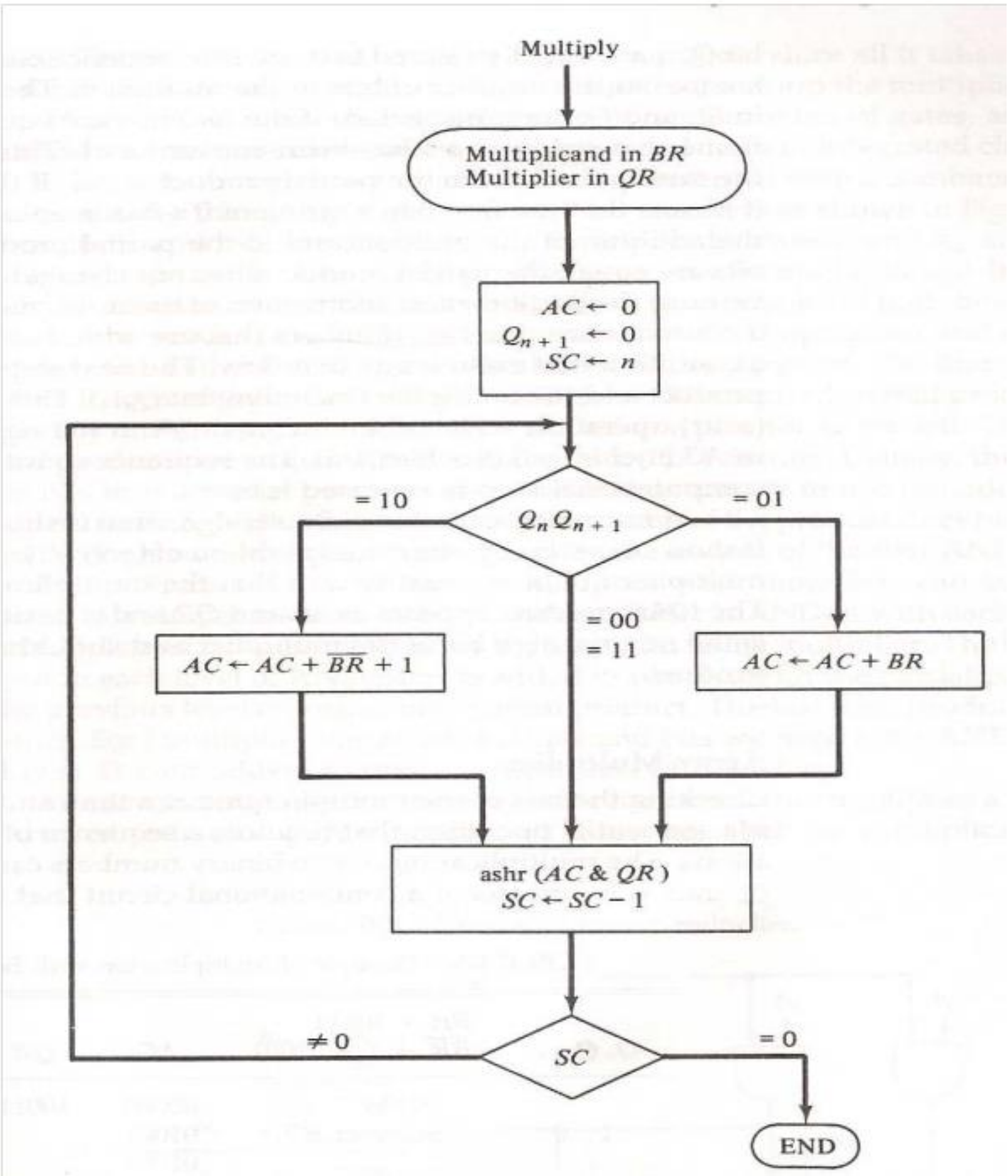
A numerical example of Booth algorithm is given in Table 4.3 for n = 5. It gives the multiplication of (-9) x (-13) = +117. Note that the multiplier in QR is negative and that the multiplicand in BR is also negative. The 10-bit product appears in AC. The final value of Qn+1 is the original sign bit of the multiplier and should not be taken as part of the product.





$Q_n\ Q_{n+1}$	BR = 10111 $\overline{BR}$ + 1 = 01001	AC	QR	Q_{n+1}	SC
	Initial	00000	10011	0	101
1 0	Subtract BR	01001			
		01001			
	ashr	00100	11001	1	100
1 1	ashr	00010	01100	1	011
0 1	Add BR	10111			
		11001			
	ashr	11100	10110	0	010
0 0	ashr	11110	01011	0	001
1 0	Subtract BR	01001			
		00111			
	ashr	00011	10101	1	000


Multiply
Multiplicand in BR
Multiplier in QR
$AC \leftarrow 0$
$Q_{n+1} \leftarrow 0$
$SC \leftarrow n$
$Q_n Q_{n+1}$
= 10
= 01
= 00
= 11
$AC \leftarrow AC + \overline{BR} + 1$
$AC \leftarrow AC + BR$
ashr (AC & QR)
$SC \leftarrow SC - 1$
SC
≠ 0
= 0
END



2.8.3. Array Multiplier

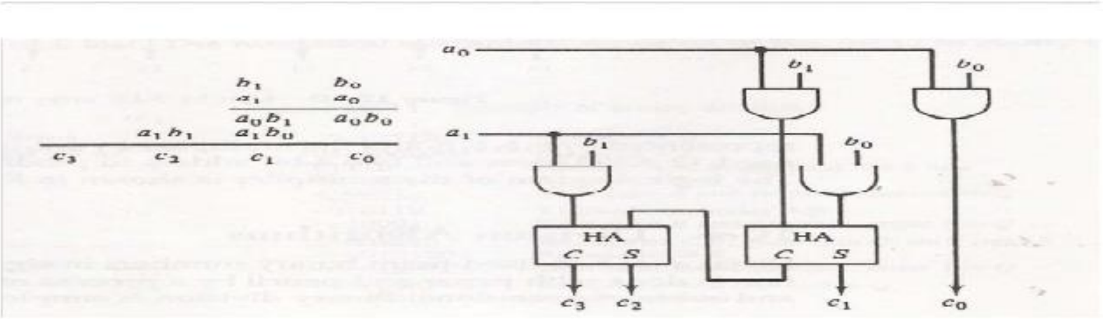
To check the bits of the multiplier one at a time and forming partial products is a sequential operation requiring a sequence of add and shift micro-operations. The multiplication of two binary numbers can be done with one micro-operation by using combinational circuit that forms the product bits all at once.

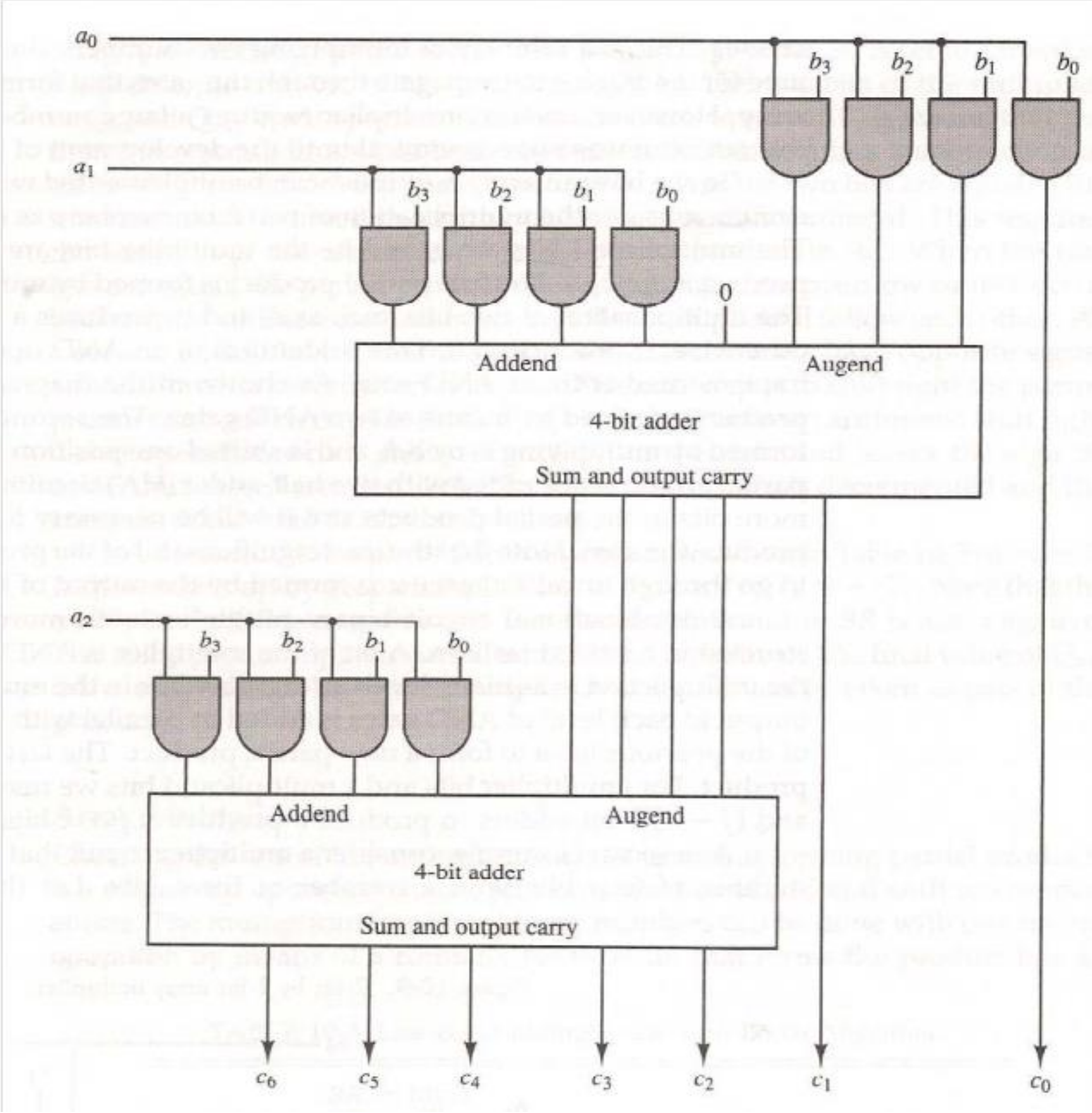
This is a fast way since all it takes is the time for the signals to propagate through the gates that form the multiplication array. However, an array multiplier requires a large number of gates, and so it is not an economical unit for the development of ICs. Now we see how an array multiplier is implemented with a combinational circuit. Consider the multiplication of two 2-bit numbers as shown in Fig. 4.8.

The multiplicand bits are b1 and b0, the multiplier bits are a1 and a0, and the product is c3 c2 c1 c0. The first partial product is obtained by multiplying a0 by b1b0. The multiplication of two bits gives a 1 if both bits are 1; otherwise, it produces a 0. This is identical to an AND operation and can we implement it with an AND gate. As shown in the diagram, the first partial product is formed by means of two AND gates. The second partial product is formed by multiplying a1 by b1b0 and is shifted one position to the left. The two partial products are added with two half-adder (HA) circuits. Usually, there are more bits in the partial products and it will be necessary to use full-adders to produce the sum. Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the first AND gate.

A combinational circuit binary multiplier with more bits can be constructed in a similar fashion. A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier. The binary output in each level AND gates is added in parallel with the partial product of the previous level to form a new partial product. The last level produces the product. For j multiplier bits and k multiplicand bits we need j * k AND gates and (j – 1) k-bit adders to produce a product of j + k bits.

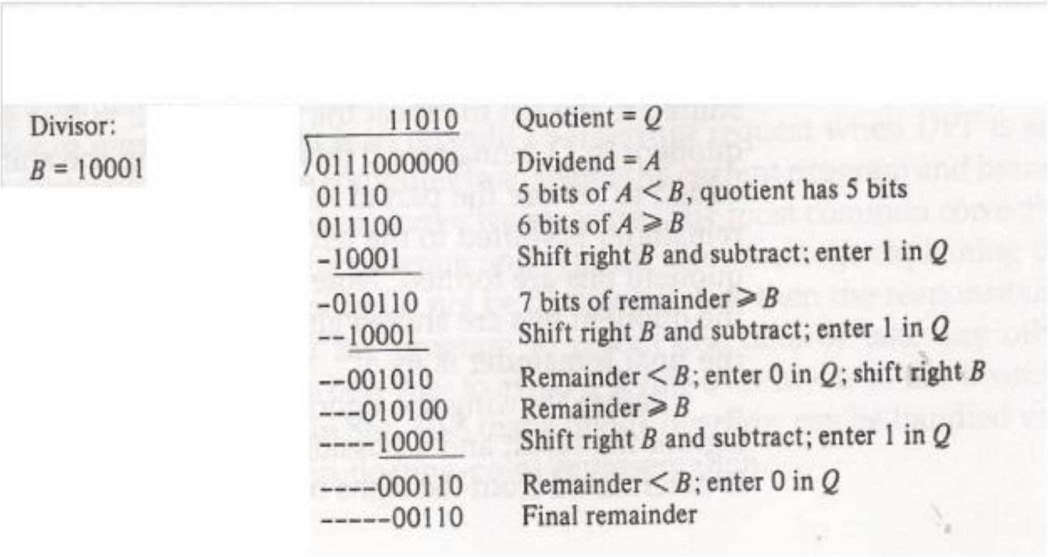
As a second example, consider a multiplier circuit that multiplies a binary number of four bits with a number of three bits. Let the multiplicand be represented by b3b2b1b0 and the multiplier by a2a1a0. Since k=4 and j=3, we need 12 AND gates and two 4-bit adders to produce a product of seven bits. The logic diagram of the multiplier is





2.9. DIVISION ALGORITHMS

Division of two fixed-point binary numbers in signed magnitude representation is performed with paper and pencil by a process of successive compare, shift and subtract operations. Binary division is much simpler than decimal division because here the quotient digits are either 0 or 1 and there is no need to estimate how many times the dividend or partial remainder fits into the divisor. The division process is described in Figure 4.10. The divisor B has five bits and the dividend A has ten.



```
Divisor:            11010       Quotient = Q
B = 10001     )0111000000       Dividend = A
               01110            5 bits of A < B, quotient has 5 bits
               011100           6 bits of A ⩾ B
              -10001            Shift right B and subtract; enter 1 in Q
              -010110           7 bits of remainder ⩾ B
              --10001           Shift right B and subtract; enter 1 in Q
              --001010          Remainder < B; enter 0 in Q; shift right B
              ---010100         Remainder ⩾ B
              ----10001         Shift right B and subtract; enter 1 in Q
              ----000110        Remainder < B; enter 0 in Q
              -----00110        Final remainder
```

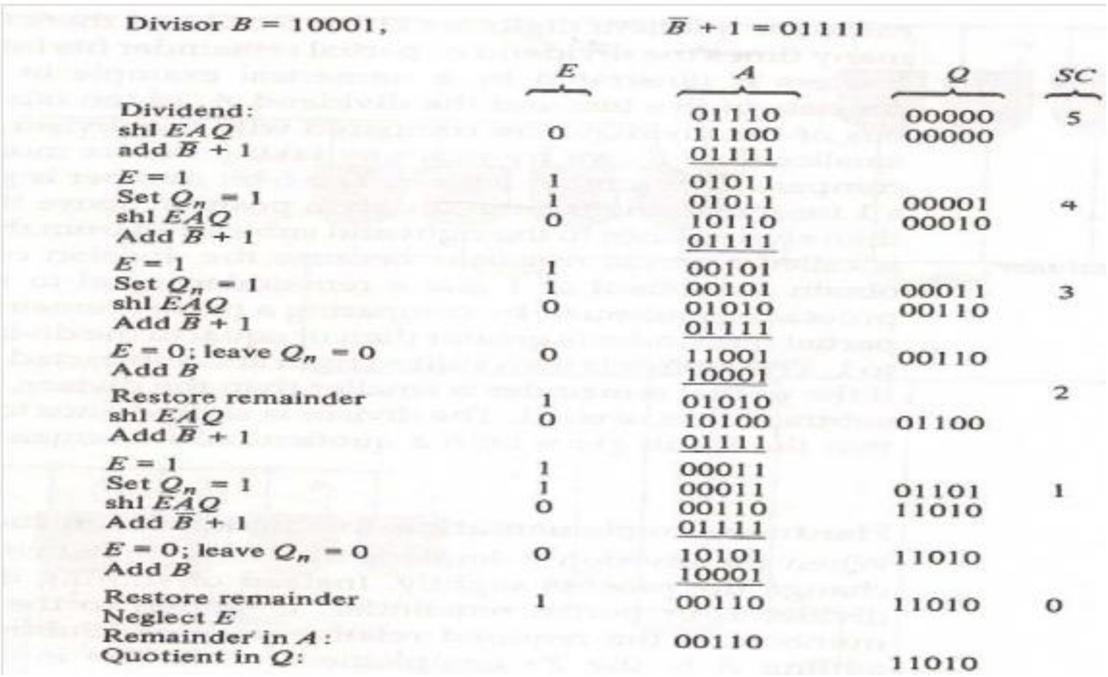
The devisor is compared with the five most significant bits of the dividend. Since the 5- bit number is smaller than B, we again repeat the same process. Now the 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. Now we shift the divisor once to the right and subtract it from the dividend. The difference is known as a partial remainder because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. Comparing a partial remainder with the divisor continues the process. If the partial remainder is greater than or equal to the divisor, the quotient bit is equal to 1. The divisor is then shifted right and subtracted from the partial remainder. If the partial remainder is smaller than the divisor, the quotient bit is 0 and no subtraction is needed. The divisor is shifted once to the right in any case. Obviously the result gives both a quotient and a remainder.

2.9.1.Hardware Implementation for Signed-Magnitude Data:

In hardware implementation for signed-magnitude data in a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, two dividends, or partial remainders, are shifted to the left, thus leaving the two numbers in the required relative position. Subtraction is achieved by adding A to the 2's complement of B. End carry gives the information about the relative magnitudes.

The hardware required is identical to that of multiplication. Register EAQ is now shifted to the left with 0 inserted into Qn and the previous value of E is lost. The example is given in Figure 4.10 to clear the proposed division process. The divisor is stored in the B register and the double-length dividend is stored in registers A and Q. The dividend is shifted to the left and the divisor is subtracted by adding its 2's complement value.

The register E keeps the information about the relative magnitude. A quotient bit 1 is inserted into Qn and the partial remainder is shifted to the left to repeat the process when E = 1. If E =0, it signifies that A < B so the quotient in Qn remains a 0 (inserted during the shift). To restore the partial remainder in A the value of B is then added to its previous value. The partial remainder is shifted to the left and the process is repeated again until we get all five quotient-bits. Note that while the partial remainder is shifted left, the quotient bits are shifted also and after five shifts, the quotient is in Q and A has the final remainder. Before showing the algorithm in flowchart form, we have to consider the sign of the result and a possible overflow condition. The sign of the quotient is obtained from the signs of the dividend and the divisor. If the two signs are same, the sign of the quotient is plus. If they are not identical, the sign is minus. The sign of the remainder is the same as that of the dividend.



Divisor $B = 10001$, $\overline{B} + 1 = 01111$

	E	A	Q	SC
Dividend:		01110	00000	5
shl EAQ	0	11100	00000	
add $\overline{B} + 1$		01111		
$E = 1$	1	01011		
Set $Q_n = 1$	1	01011	00001	4
shl EAQ	0	10110	00010	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00101		
Set $Q_n = 1$	1	00101	00011	3
shl EAQ	0	01010	00110	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	11001	00110	
Add B		10001		
Restore remainder	1	01010		2
shl EAQ	0	10100	01100	
Add $\overline{B} + 1$		01111		
$E = 1$	1	00011		
Set $Q_n = 1$	1	00011	01101	1
shl EAQ	0	00110	11010	
Add $\overline{B} + 1$		01111		
$E = 0$; leave $Q_n = 0$	0	10101	11010	
Add B		10001		
Restore remainder	1	00110	11010	0
Neglect E				
Remainder in A:		00110		
Quotient in Q:			11010	

2.9.2. Hardware Algorithm:

Figure 4.6 is a flowchart of the hardware multiplication algorithm. In the beginning, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in Bs and Qs respectively. We compare the signs of both A and Q and set to corresponding sign of the product since a double-length product will be stored in registers A and Q. Registers A and E are cleared and the sequence counter SC is set to the number of bits of the multiplier. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n-1 bits.

Now, the low order bit of the multiplier in Qn is tested. If it is 1, the multiplicand (B) is added to present partial product (A), 0 otherwise. Register EAQ is then shifted once to the right to form the new partial product. The sequence counter is decremented by 1 and its new value checked. If it is not equal to zero, the process is repeated and a new partial product is formed. When SC = 0 we stops the process.

The hardware divide algorithm is given in Figure A and Q contain the dividend and B has the divisor. The sign of the result is transferred into Q. A constant is set into the sequence counter SC to specify the number of bits in the quotient. As in multiplication, we assume that operands are transferred to registers from a memory unit that has words of n bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will have n-1 bits. We can check a divide-overflow condition by subtracting the divisor (B) from half of the bits of the dividend stored (A). If A<B, the divide-overflow occur and the operation is terminated. If A≥ B, no divide overflow occurs and so the value of the dividend is restored by adding B to A.

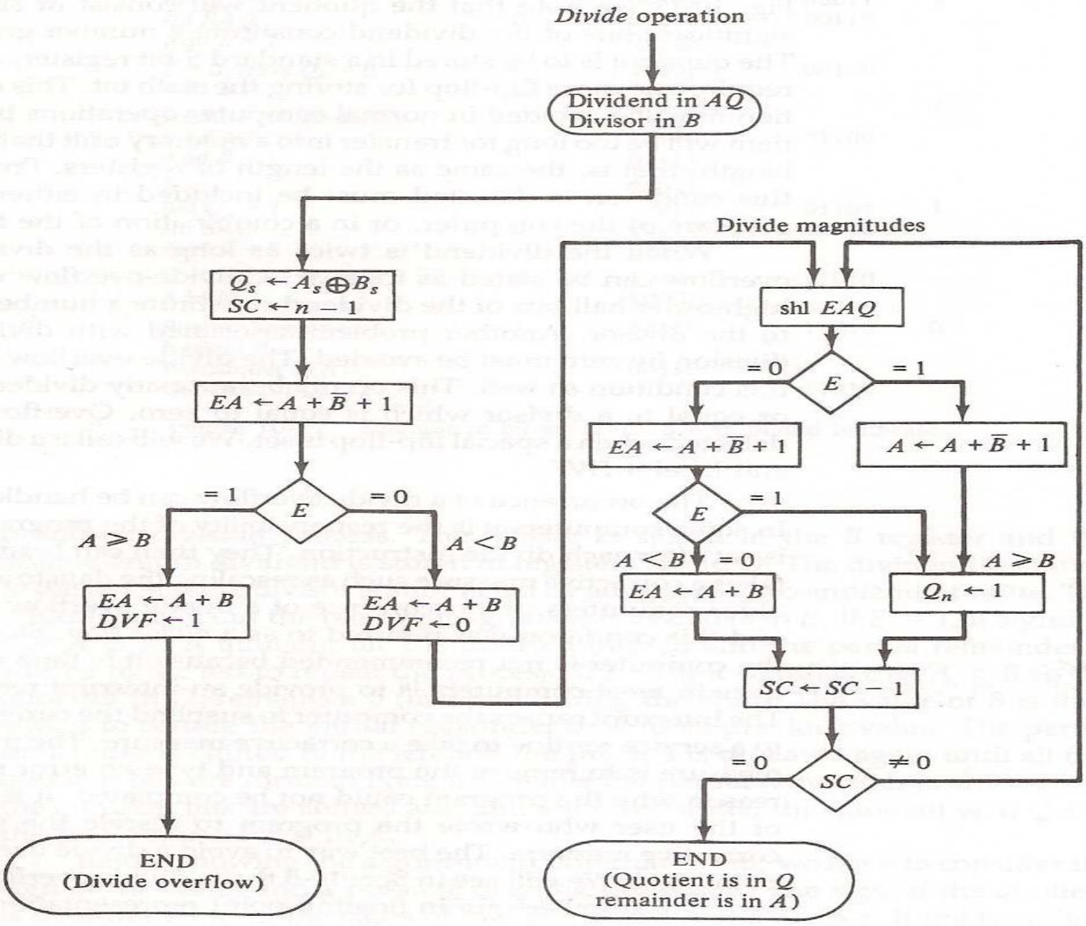
The division of the magnitudes begins by shifting the dividend in AQ to the left with the high-order bit shifted into E. If the bit shifted into E is 1, we know that EA>B because EA consists of 1 followed by n-1 bits while B consists of only n-1 bits. In this case, B must be subtracted from EA and 1 inserted into Qn for the quotient bit. Since in register A, the high-order bit of the dividend (which is in E) is missing, its value is EA – 2n-1. Adding to this value the 2’s complement of B results in:

$$(EA - 2n\text{-}1) + (2n\text{-}1 - B) = EA - B$$

If we want E to remain a 1, the carry from this addition is not transferred to E. If the shift-left operation inserts a 0 into E, we subtract the divisor by adding its 2’s complement value and the carry is transferred into E. If E=1, it shows that A< B,

therefore Qn is set. If E = 0, it signifies that A < B and the original number is restored by B + A. In the latter case we leave a 0 in Qn. We repeat this process with register A holding the partial remainder. After n-1 loops, the quotient magnitude is stored in register Q and the remainder is found in register A. The quotient sign is in Qs and the sign of the remainder is in As.

Figure 10-13 Flowchart for divide operation.



2.9.3. Divide Overflow

An overflow may occur in the division operation, which may be easy to handle if we are using paper and pencil but is not easy when are using hardware. This is because the length of registers is finite and will not hold a number that exceeds the standard length. To see this, let us consider a system that has 5-bit registers. We use one register to hold

the divisor and two registers to hold the dividend. From the example of Figure the quotient will consist of six bits if the five most significant bits of the dividend constitute a number greater than the divisor. The quotient is to be stored in a standard 5-bit register, so the overflow bit will require one more flip-flop for storing the sixth bit. This divide-overflow condition must be avoided in normal computer operations because the entire quotient will be too long for transfer into a memory unit that has words of standard length, that is, the same as the length of registers. Provisions to ensure that this condition is detected must be included in either the hardware or the software of the computer, or in a combination of the two.

When the dividend is twice as long as the divisor, we can understand the condition for overflow as follows:

A divide-overflow occurs if the high-order half bits of the dividend makes a number greater than or equal to the divisor. Another problem associated with division is the fact that a division by zero must be avoided. The divide-overflow condition takes care of this condition as well. This occurs because any dividend will be greater than or equal to a divisor, which is equal to zero. Overflow condition is usually detected when a special flip-flop is set. We will call it a divide-overflow flip-flop and label it DVF.

2.10. FLOATING-POINT ARITHMETIC OPERATIONS

In many high-level programming languages we have a facility for specifying floating point numbers. The most common way is by a real declaration statement. High level programming languages must have a provision for handling floating-point arithmetic operations. The operations are generally built in the internal hardware. If no hardware is available, the compiler must be designed with a package of floating-point software subroutine. Although the hardware method is more expensive, it is much more efficient than the software method. Therefore, floating- point hardware is included in most computers and is omitted only in very small ones.

2.10.1. Basic Considerations

There are two part of a floating-point number in a computer - a mantissa m and an exponent e. The two parts represent a number generated from multiplying m times a radix r raised to the value of e. Thus

m x re

The mantissa may be a fraction or an integer. The position of the radix point and the value of the radix r are not included in the registers. For example, assume a fraction representation and a radix 10. The decimal number 537.25 is represented in a register with m = 53725 and e = 3 and is interpreted to represent the floating-point number

.53725 x 103

A floating-point number is said to be normalized if the most significant digit of the mantissa in nonzero. So the mantissa contains the maximum possible number of significant digits. We cannot normalize a zero because it does not have a nonzero digit. It is represented in floating-point by all 0's in the mantissa and exponent. Floating-point representation increases the range of numbers for a given register.

Consider a computer with 48-bit words. Since one bit must be reserved for the sign, the range of fixed-point integer numbers will be + (247 – 1), which is approximately + 1014. The 48 bits can be used to represent a floating-point number with 36 bits for the mantissa and 12 bits for the exponent. Assuming fraction representation for the mantissa and taking the two sign bits into consideration, the range of numbers that can be represented is

+ (1 – 2-35) x 22047

This number is derived from a fraction that contains 35 1's, an exponent of 11 bits(excluding its sign), and because 211–1 = 2047. The largest number that can be accommodated is approximately 10615. The mantissa that can accommodated is 35 bits (excluding the sign) and if considered as an integer it can store a number as large as
(235 –1). This is approximately equal to 1010, which is equivalent to a decimal number of 10 digits.

Computers with shorter word lengths use two or more words to represent a floating point number. An 8-bit microcomputer uses four words to represent one floating-point number. One word of 8 bits are reserved for the exponent and the 24 bits of the other three words are used in the mantissa.

Arithmetic operations with floating-point numbers are more complicated than with fixed-point numbers. Their execution also takes longer time and requires more complex hardware. Adding or subtracting two numbers requires first an alignment of the radix point since the exponent parts must be made equal before adding or subtracting the mantissas. We do this alignment by shifting one mantissa while its exponent is adjusted until it becomes equal to the other exponent. Consider the sum of the following floating-point numbers:

.5372400 x 102
+ .1580000 x 10-1

It is necessary to make two exponents be equal before the mantissas can be added. We can either shift the first number three positions to the left, or shift the second number three positions to the right. When we store the mantissas in registers, shifting to the left causes a loss of most significant digits. Shifting to the right causes a loss of least significant digits. The second method is preferable because it only reduces the accuracy, while the first method may cause an error. The usual alignment procedure is to shift the mantissa that has the smaller exponent to the right by a number of places equal to the difference between the exponents. Now, the mantissas can be added.

. 5372400 x 102

+. 0001580 x 102

. 5373980 x 102

When two normalized mantissas are added, the sum may contain an overflow digit. An overflow can be corrected easily by shifting the sum once to the right and incrementing the exponent. When two numbers are subtracted, the result may contain most significant zeros as shown in the following example:

.56780 x 105

- .56430 x 105

.00350 x 105

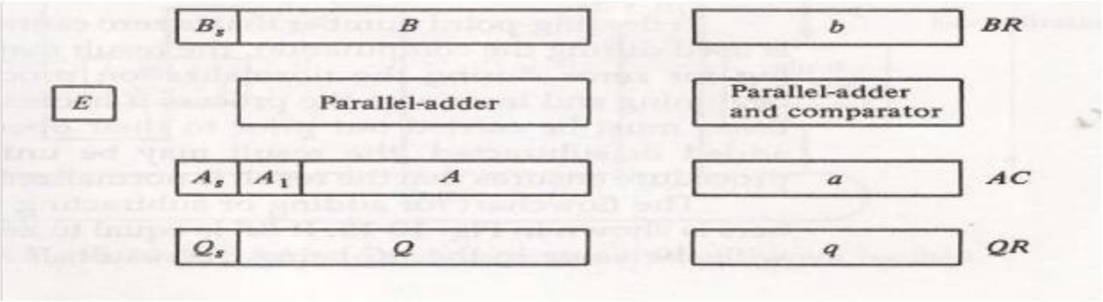
An underflow occurs if a floating-point number that has a 0 in the most significant position of the mantissa. To normalize a number that contains an underflow, we shift the mantissa to the left and decrement the exponent until a nonzero digit appears in the first position. Here, it is necessary to shift left twice to obtain .35000 x 103. In most computers a normalization procedure is performed after each operation to ensure that all results are in a normalized form. Floating-point multiplication and division need not do an alignment of the mantissas. Multiplying the two mantissas and adding the exponents can form the product. Dividing the mantissas and subtracting the exponents perform division.

The operations done with the mantissas are the same as in fixed-point numbers, so the two can share the same registers and circuits. The operations performed with the exponents are compared and incremented (for aligning the mantissas), added and subtracted (for multiplication) and division), and decremented (to normalize the result). We can represent the exponent in any one of the three representations – signed magnitude, signed 2's complement or signed 1's complement. A is a fourth representation also, known as a biased exponent.

In this representation, the sign bit is removed from beginning to form a separate entity. The bias is a positive number that is added to each exponent as the floating-point number is formed, so that internally all exponents are positive. The following example may clarify this type of representation. Consider an exponent that ranges from –50 to 49. Internally, it is represented by two digits (without a sign) by adding to it a bias of 50. The exponent register contains the number e + 50, where e is the actual exponent. This way, the exponents are represented in registers as positive numbers in the range of 00 to 99. Positive exponents in registers have the range of numbers from 99 to 50. The subtraction of 50 gives the positive values from 49 to 0. Negative exponents are represented in registers in the range of –1 to –50. Biased exponents have the advantage that they contain only positive numbers. Now it becomes simpler to compare their relative magnitude without bothering about their signs. Another advantage is that the smallest possible biased exponent contains all zeros. The floating-point representation of zero is then a zero mantissa and the smallest possible exponent.

2.10.2. Register Configuration

The register configuration for floating-point operations is shown in figure 4.13. As a rule, the same registers and adder used for fixed-point arithmetic are used for processing the mantissas. The difference lies in the way the exponents are handled. The register organization for floating-point operations is shown in Fig. 4.13. Three registers are there, BR, AC, and QR. Each register is subdivided into two parts. The mantissa part has the same uppercase letter symbols as in fixed-point representation. The exponent part may use corresponding lower-case letter symbol.



Assuming that each floating-point number has a mantissa in signed-magnitude representation and a biased exponent. Thus the AC has a mantissa whose sign is in As, and a magnitude that is in A. The diagram shows the most significant bit of A, labeled by A1. The bit in his position must be a 1 to normalize the number. Note that the symbol AC represents the entire register, that is, the concatenation of As, A and a. In the similar way, register BR is subdivided into Bs, B, and b and QR into Qs, Q and q.

A parallel-adder adds the two mantissas and loads the sum into A and the carry into E. A separate parallel adder can be used for the exponents. The exponents do not have a district sign bit because they are biased but are represented as a biased positive quantity. It is assumed that the floating-point number is so large that the chance of an exponent overflow is very remote and so the exponent overflow will be neglected. The exponents are also connected to magnitude comparator that provides three binary outputs to indicate their relative magnitude.

The number in the mantissa will be taken as a fraction, so they binary point is assumed to reside to the left of the magnitude part. Integer representation for floating point causes certain scaling problems during multiplication and division. To avoid these problems, we adopt a fraction representation. The numbers in the registers should initially be normalized. After each arithmetic operation, the result will be normalized. Thus all floating-point operands are always normalized.

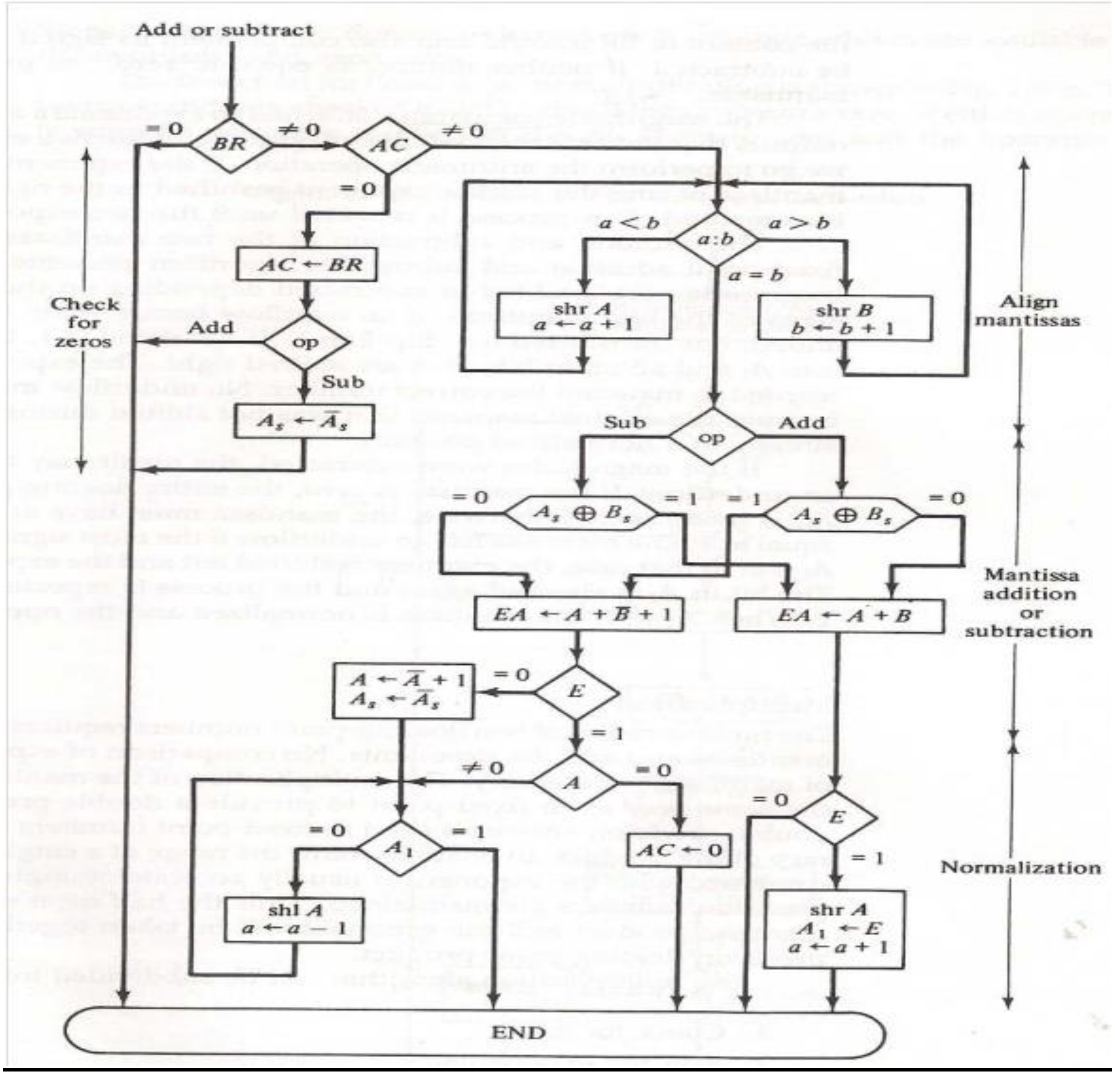
2.10.3. Addition and Subtraction of Floating Point Numbers

During addition or subtraction, the two floating-point operands are kept in AC and BR. The sum or difference is formed in the AC. The algorithm can be divided into four consecutive parts:

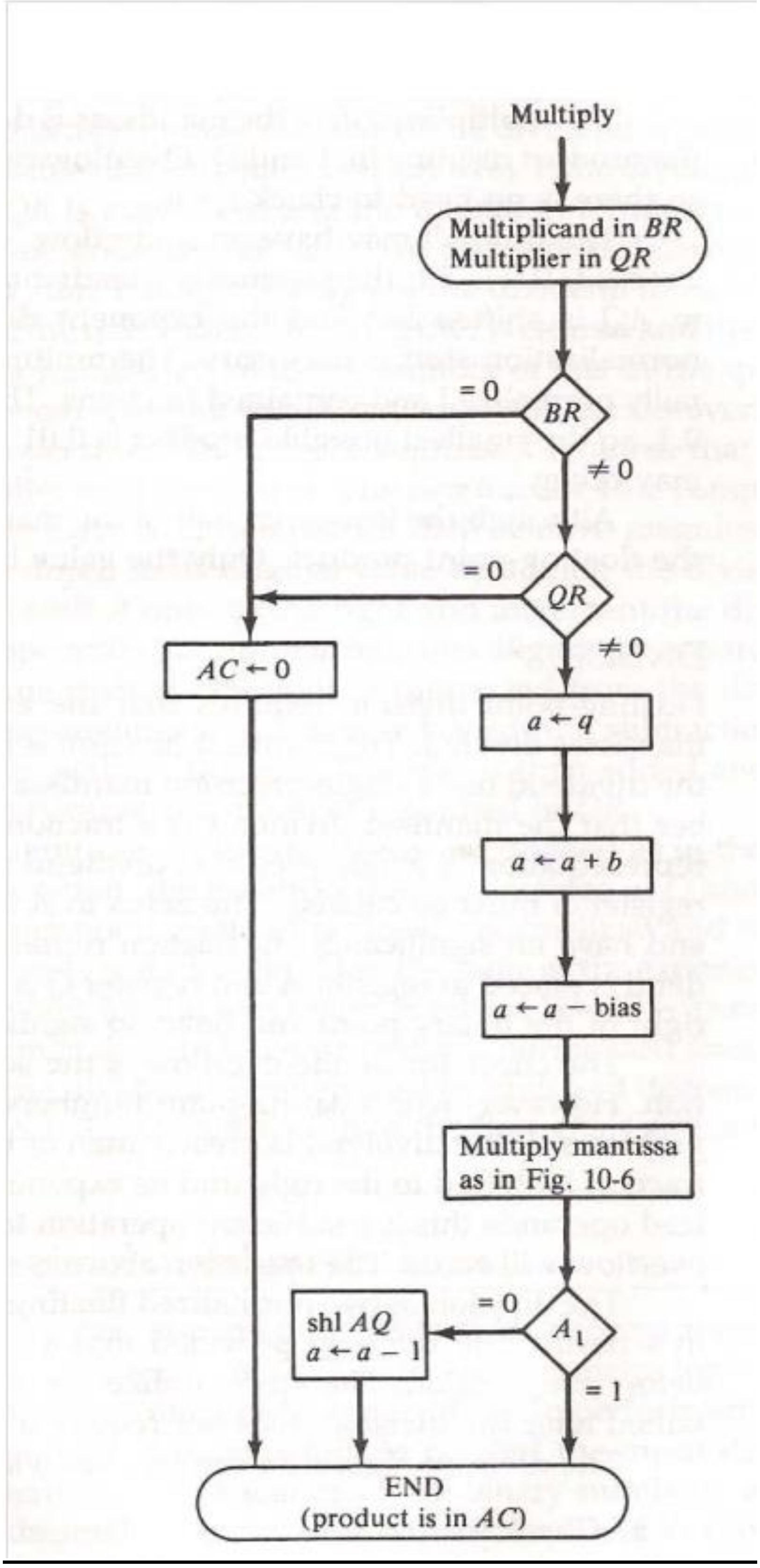
1. Check for zeros.
2. Align the mantissas.
3. Add or subtract the mantissas
4. Normalize the result

A floating-point number cannot be normalized, if it is 0. If this number is used for computation, the result may also be zero. Instead of checking for zeros during the normalization process we check for zeros at the beginning and terminate the process if necessary. The alignment of the mantissas must be carried out prior to their operation. After the mantissas are added or subtracted, the result may be un-normalized. The normalization procedure ensures that the result is normalized before it is transferred to memory. For adding or subtracting two floating-point binary numbers, if BR is equal to zero, the operation is stopped, with the value in the AC being the result. If AC = 0, we transfer the content of BR into AC and also complement its sign we have to subtract the numbers. If neither number is equal it to zero, we proceed to align the mantissas. The magnitude comparator attached to exponents a and b gives three outputs, which show their relative magnitudes. If the two exponents are equal, we go to perform the arithmetic operation. If the exponents are not equal, the mantissa having the smaller exponent is shifted to the right and its exponent incremented. This process is repeated until two exponents are equal.

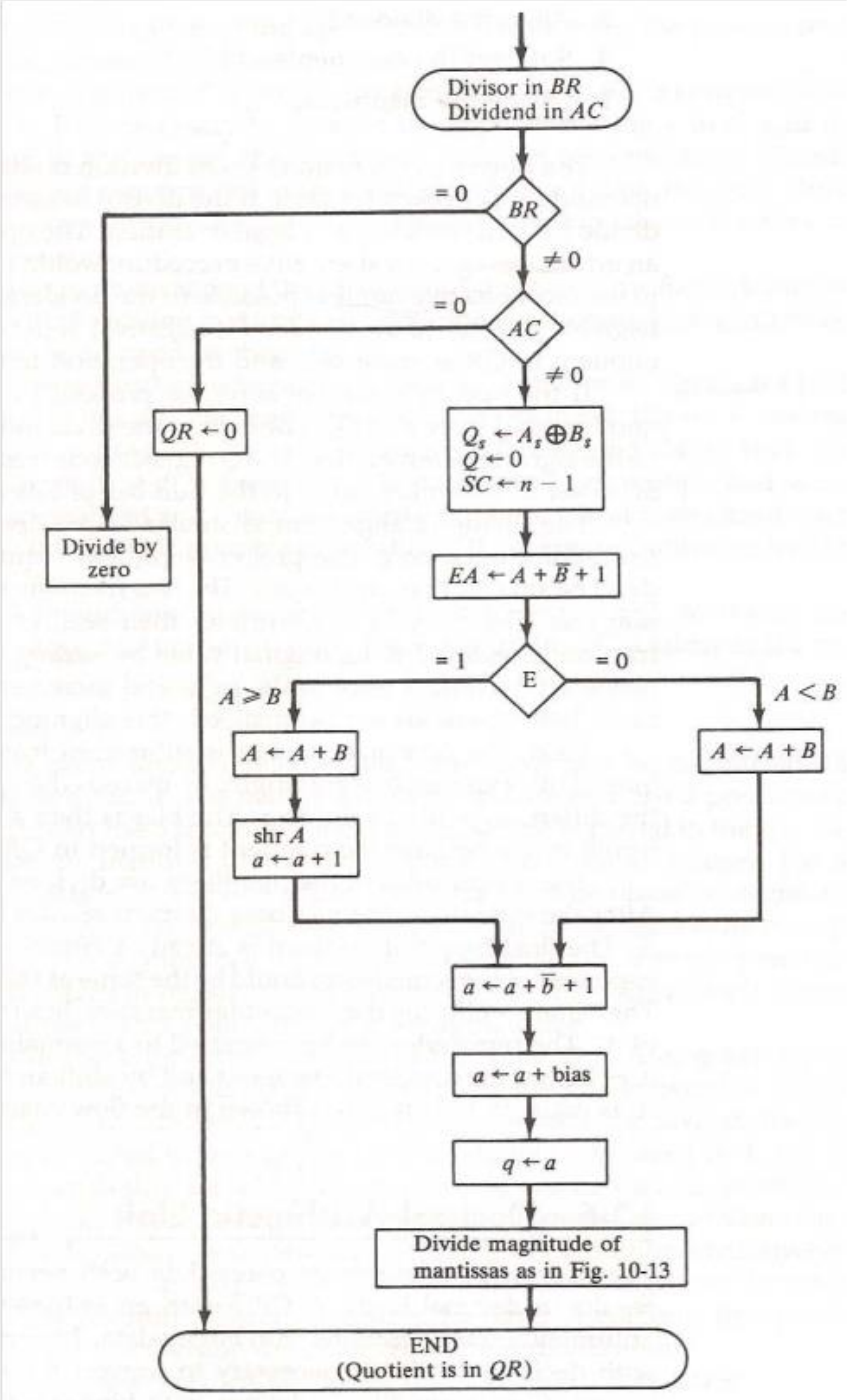
The addition and subtraction of the two mantissas is similar to the fixed-point addition and subtraction algorithm presented in Fig. 4.14. The magnitude part is added or subtracted depends on the operation and the signs of the two mantissas. If an overflow occurs when the magnitudes are added, it is transferred into flip-flop E. If E = 1, the bit is transferred into A1 and all other bits of A are shifted right. The exponent must be incremented so that it can maintain the correct number. No underflow may occur in this case this is because the original mantissa that was not shifted during the alignment was already in a normalized position. If the magnitudes were subtracted, there may be zero or may have an underflow in the result. If the mantissa is equal to zero the entire floating-point number in the AC is cleared to zero. Otherwise, the mantissa must have at least one bit that is equal to 1.The mantissa has an underflow if the most significant bit in position A1, is 0. In that case, the mantissa is shifted left and the exponent decremented. The bit in A1 is checked again and the process is repeated until A1 = 1. When A1 = 1, the mantissa is normalized and the operation is completed.



2.10.4. Multiplication



2.10.5.Division



2.11. DECIMAL ARITHMETIC OPERATIONS:

2.11.1. Decimal Arithmetic Unit:

The user of a computer input data in decimal numbers and receives output in decimal form. But a CPU with an ALU can perform arithmetic micro-operations only on binary data. To perform arithmetic operations with decimal data, it is necessary to convert the input decimal numbers to binary, to perform all calculations with binary numbers, and to convert the results into decimal. This may be an efficient method in applications requiring a large number of calculations and a relatively smaller amount of input and output data. When the application calls for a large amount of input-output and a

relatively smaller number of arithmetic calculations, it becomes convenient to do the internal arithmetic directly with the decimal numbers. Computers that can do decimal arithmetic must store the decimal data in binary coded form. The decimal numbers are then applied to a decimal arithmetic unit, which can execute decimal arithmetic micro-operations.

Electronic calculators invariably use an internal decimal arithmetic unit since inputs and outputs are frequent. There does not seem to be a reason for converting the keyboard input numbers to binary and again converting the displayed results to decimal, this is because this process needs special circuits and also takes a longer time

to execute. Many computers have hardware for arithmetic calculations with both binary and decimal data.

Users can specify by programmed instructions whether they want the computer to does calculations with binary or decimal data. A decimal arithmetic unit is a digital function that does decimal micro-operations. It can add or subtract decimal numbers. The unit needs coded decimal numbers and produces results in the same adopted binary code. A single-stage decimal arithmetic unit has of nine binary input variables and five binary output variables, since a minimum of four bits is required to represent each coded decimal digit. Each stage must have four inputs for the addend digit, four inputs for the addend digit, and an input-carry. The outputs need four terminals for the sum digit and one for the output carry. Of course, there is a wide range of possible circuit configurations dependent on the code used to represent the decimal digits.

2.11.2. BCD Adder:

Now let us see the arithmetic addition of two decimal digits in BCD, with a possible carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than 9 + 9 + 1 = 19, the 1 in the sum being an input-carry. Assume that we apply two BCD digits to a 4-bit binary adder. The adder will form the sum in binary and produce a result that may range from 0 to 19. These binary numbers are listed in Table 4.4 and are labeled by symbols K, Z8, Z4, Z2, and Z1. K is the carry and the subscripts under the letter Z represent the weights 8, 4, 2, and 1 that can be

assigned to the four its in the BCD code. The first column in the table lists the binary sums as they appear in the outputs of a 4-bit binary adder. The output sum of two decimal numbers must be represented in BCD and should appear in the form listed in the second column of the table. The problem is to find a simple rule by which the binary column of the table. The problem is to find a simple rule so that the binary number in the first column can be converted to the correct BCD digit representation of the number in the second column. It is apparent that when the binary sum is equal to or less than 1001, no conversion is needed. When the binary sum is greater than 1001, we need to add of binary 6 (0110) to the binary sum to find the correct BCD representation and to produces output-carry as required.

One way of adding decimal numbers in BCD is to use one 4-bit binary adder and perform the arithmetic operation one digit at a time. The low-order pair of BCD digits is first added to produce a binary sum if the result is equal or greater than 1010, it is corrected by adding 0110 to the binary sum. The second operation produces an output carry for the next pair of significant digits. The next higher-order pair of digits, together with the input-carry, is then added to produce their binary sum. If this result is equal to or greater than 1010, it is corrected by adding 0110. The procedure is repeated until

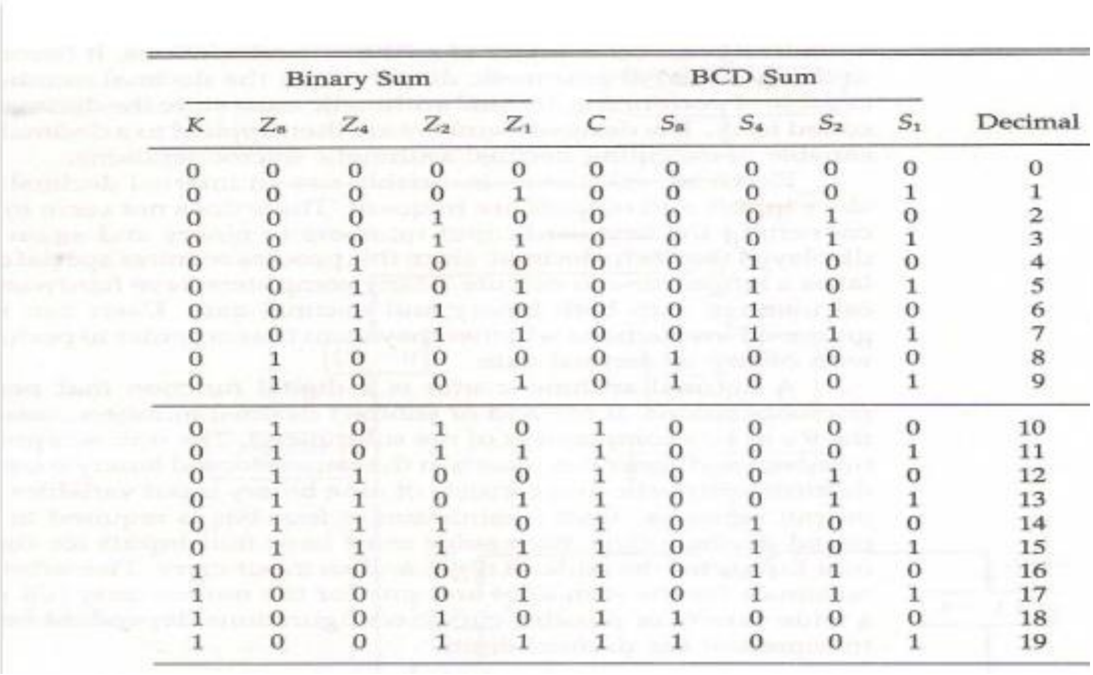
all decimal digits are added.

The logic circuit that detects the necessary correction can be derived from the table entries. It is obvious that a correction is needed when the binary sum has an output carry K = 1. The other six combinations from

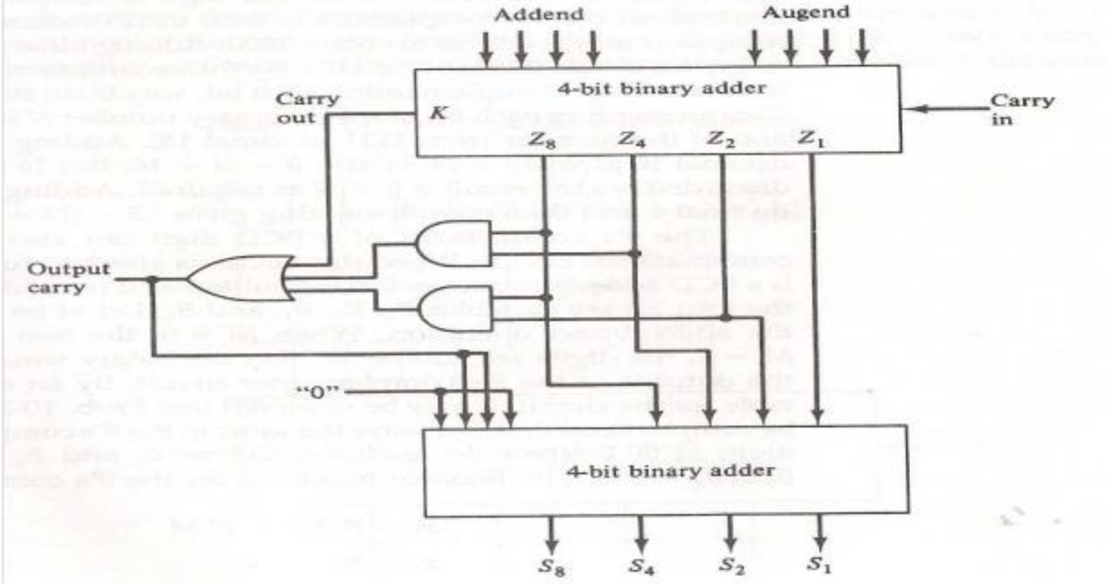
1010 to 1111 that need a correction have a 1 in position Z8. To differentiate them from binary 1000 and 1001, which also have a 1 in position Z8, we specify further that either Z4 or Z2 must have a 1. The condition for a correction and an output-carry can be expressed by the Boolean function

C = K + Z8 Z4 + Z8 Z2

When C = 1, we need to add 0110 to the binary sum and provide an output-carry for the next stage. A BCD adder is circuit that adds two BCD digits in parallel and generates a sum digit also in BCD. ABCD adder must include the correction logic in its internal construction. To add 0110 to the binary sum, we use a second 4-bit binary adder. The two decimal digits, together with the input-carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output-carry is equal to 0, nothing is added to the binary sum through the bottom 4-bit binary adder. The output-carry generated from the bottom binary adder may be ignored, since it supplies information already available in the output-carry terminal.



Binary Sum					BCD Sum					
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	Decimal
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19



2.11.3. BCD Subtraction:

Subtraction of two decimal numbers needs a subtractor circuit that is different from a BCD adder. We perform the subtraction by taking the 9's or 10's complement of the subtrahend and adding it to the minuend. Since the BCD is not a self-complementing code, we cannot obtain the 9's complement by complementing each bit in the code. It must be formed using a circuit that subtracts each BCD digit from 9. The 9's complement of a decimal digit represented in BCD may be obtained by complementing the bits in the coded representation of the digit but we have to include. There are two possible correction methods. In the first method, binary 1010 (decimal 10) is added to each complemented digit then we discard the carry after each addition.

In the second method, binary 0110 (decimal 6) is added before the digit is complemented. As a numerical illustration, the 9's complement of BCD 0111(decimal 7) is computed by first complementing each bit to obtain 1000. Adding binary 1010 and discarding the carry, we obtain 0010 (decimal 2). By the second method, we add 0110
to 0111 to obtain 1101. Complementing each bit, we obtain the required result of 0010. Complementing each bit of 4-bit binary number N is identical to the subtraction of the number from 1111 (decimal 15). Adding

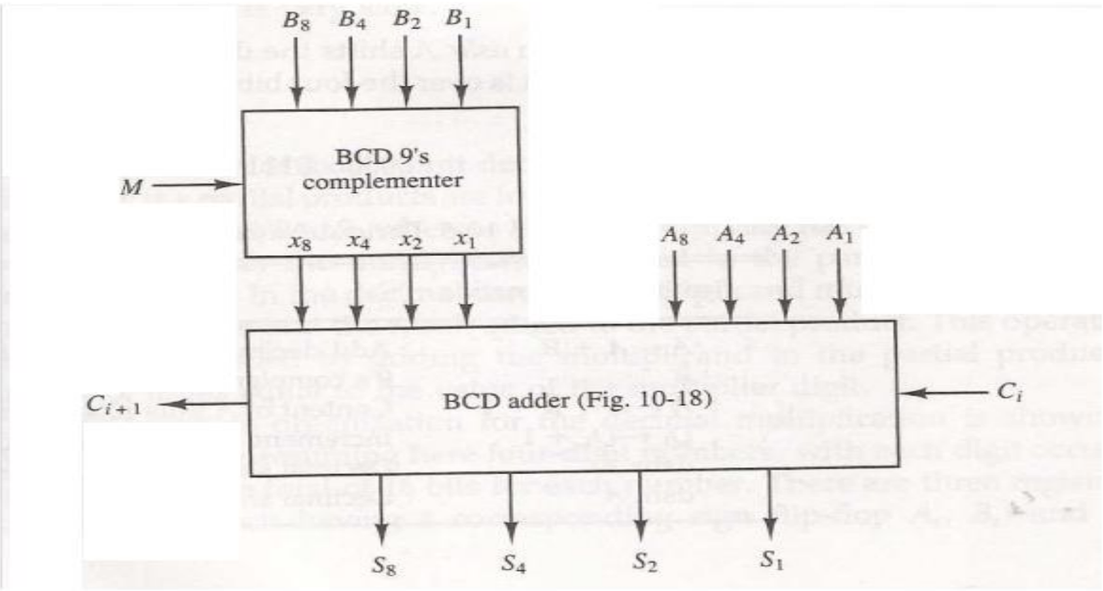
the binary equivalent of decimal 10 gives 15 – N + 10 = 9 + 16. But 16 signifies the carry that is discarded, so the result is 9 – N as required. Adding the binary equivalent of decimal 6 and then complementing gives 15 – (N + 6) = 9 – N as required.

We can also obtain the 9's complement of a BCD digit through a combinational circuit. When this circuit is combined to a BCD adder, we get a BCD adder/subtractor. Let the subtrahend (or addend) digit be denoted by the four binary variables B8, B4, B2, and B1. Let M be a mode bit that controls the add/subtract operation. When M = 0, the two

digits are added; when M = 1, the digits are subtracted. Let the binary variables x8, x4, x2, and x1 be the outputs of the 9's complement circuit. By an examination of the truth table for the circuit, it may be observed that B1 should always be complemented; B2 is always the same in the 9's complement as in the original digit; x4 is 1 when the exclusive OR of B2 and B4 is 1; and x8 is 1 when B8B4B2 = 000. The Boolean functions for the 9's complement circuit are

$$x1 = B1\,M' + B'1\,M$$
$$x2 = B2$$
$$x4 = B4M' + (B'4B2 + B4B'2)M$$
$$x8 = B8M' + B'8B4'B'2M$$

From these equations we see that x = B when M = 0. When M = 1, the x equals to the 9's complement of B. One stage of a decimal arithmetic unit that can be used to add or subtract two BCD digits is given in Fig. 4.18. It has of a BCD adder and a 9's complementer. The mode M controls the operation of the unit. With M = 0, the S outputs form the sum of A and B. With M = 1, the S outputs form the sum of A plus the 9's complement of B. For numbers with n decimal digits we need n such stages. The output carries Ci+1 from one stage. to subtract the two decimal numbers let M = 1 and apply a 1 to the input carry C1 of the first stage. The outputs will form the sum of A plus the 10's complement of B, which is equivalent to a subtraction operation if the carry-out of the last stage is discarded.


B_8 B_4 B_2 B_1
BCD 9's complementer
M
x_8 x_4 x_2 x_1
A_8 A_4 A_2 A_1
C_{i+1}
BCD adder (Fig. 10-18)
C_i
S_8 S_4 S_2 S_1

Unit-IV

Periperal Devices

Input –output Organization

4.7. Input-Output Interface

Input-output interface provides a method for transferring information between internal storage and external I/O devices. Peripherals connected to a computer need special communication links for interfacing them with the central processing unit. The purpose of the communication link is to resolve the differences that exist between the central computer and each peripheral. The major differences are:

1. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory, which are electronic devices. Therefore, a conversion of signal values may be required.

2. The data transfer rate of peripherals is usually slower than the transfer rate of the CPU, and consequently, a synchronization mechanism may be need.

3. Data codes and formats in peripherals differ form the word format in the CPU and memory.

4. The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

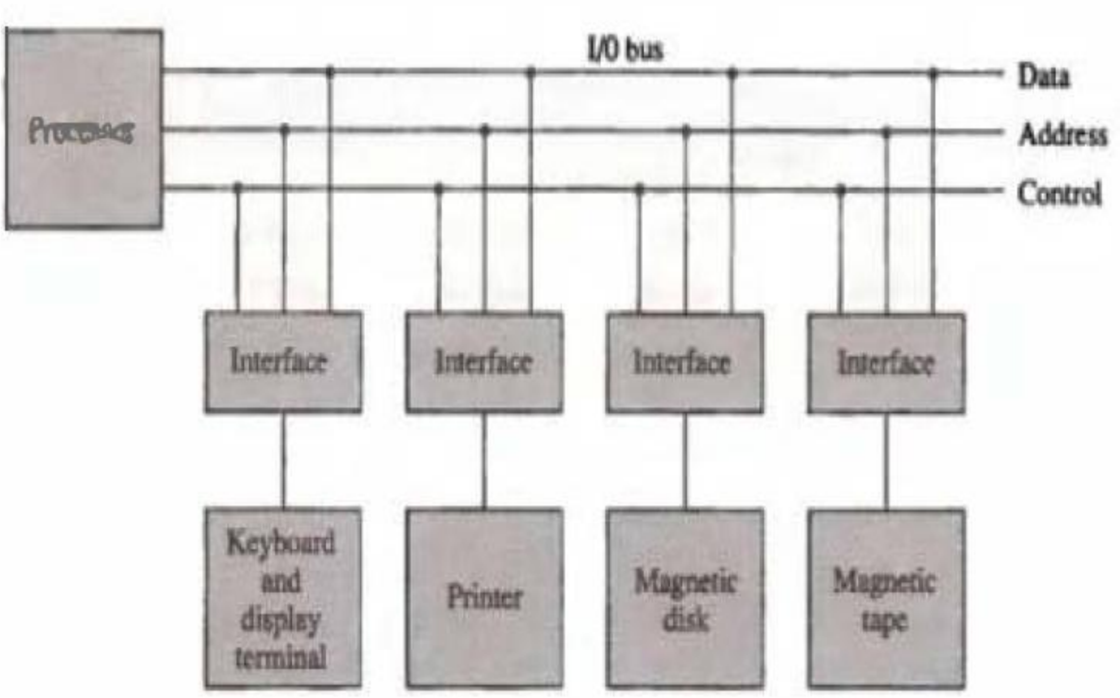
Interface units are used because they interface between the processor bus and the peripheral device. In addition, each device may have its own controller that supervises the operations of the particular mechanism in the peripheral.

Two main types of interface are CPU interface that corresponds to the system bus and input-output interface that depends on the nature of input-output device.

4.7.1. I/O Bus and Interface Modules

A typical communication link between the processor and several peripherals is shown in Fig below. I/O bus consists of data lines, address lines, and control lines. The magnetic disk, printer, and terminal are employed in practically any general-purpose computer. The magnetic tape is used in some computers for backup storage. Each peripheral device has associated with it an interface unit.

Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral, and provides signals for the peripheral controller. Each peripheral has its own controller that operates the particular electromechanical device.



The I/O bus from the processor is attached to all peripheral interfaces. To communicate with a particular device, the processor places a device address on the address lines. Each interface attached to the I/O bus contains an address decoder that monitors the address lines.

All peripherals whose address does not correspond to the address in the bus are disabled their interface.

The interface selected responds to the function code and proceeds to execute it. The function code is referred to as an I/O command and is in essence an instruction that is executed in the interface and its attached peripheral unit.

There are four types of commands that an interface may receive. They are classified as control, status, status, data output, and data input.

A control command is issued to activate the peripheral and to inform it what to do. For example, a magnetic tape unit may be instructed to backspace the tape by one record, to rewind the tape, or to start the tape moving in the forward direction. The particular control command issued depends on the peripheral, and each peripheral receives its own distinguished sequence of control commands, depending on its mode of operation.

A status command is used to test various status conditions in the interface and the peripheral.

A data output command causes the interface to respond by transferring data from the bus into one of its registers.

The data input command is the opposite of the data output. In this case the interface receives an item of data from the peripheral and places it in its buffer register

4.7.2. I/O versus Memory Bus

In addition to communicating with I/O, the processor must communicate with the memory unit. Like the I/O bus, the memory bus contains data, address, and read/write control lines. There are three ways that computer buses can be used to communicate with memory and I/O:

1. Use two separate buses, one for memory and the other for I/O.

2. Use one common bus for both memory and I/O but have separate control lines for each.

3. Use one common bus for memory and I/O with common control lines.

In the first method, the computer has independent sets of data, address, and control buses, one for accessing memory and the other for I/O. This is done in computers that provide a separate I/O processor (IOP) in addition to the central processing unit (CPU).

The purpose of the IOP is to provide an independent pathway for the transfer of information between external devices and internal memory.

4.7.3. Isolated Versus Memory-Mapped I/O

Many computers use one common bus to transfer information between memory or I/O and the CPU. The distinction between a memory transfer and I/O transfer is made through separate read and write lines. The CPU specifies whether the address on the address lines is for a memory word or for an interface register by enabling one of two possible read or write lines. The I/O read and I/O write control lines are enabled during an I/O transfer.

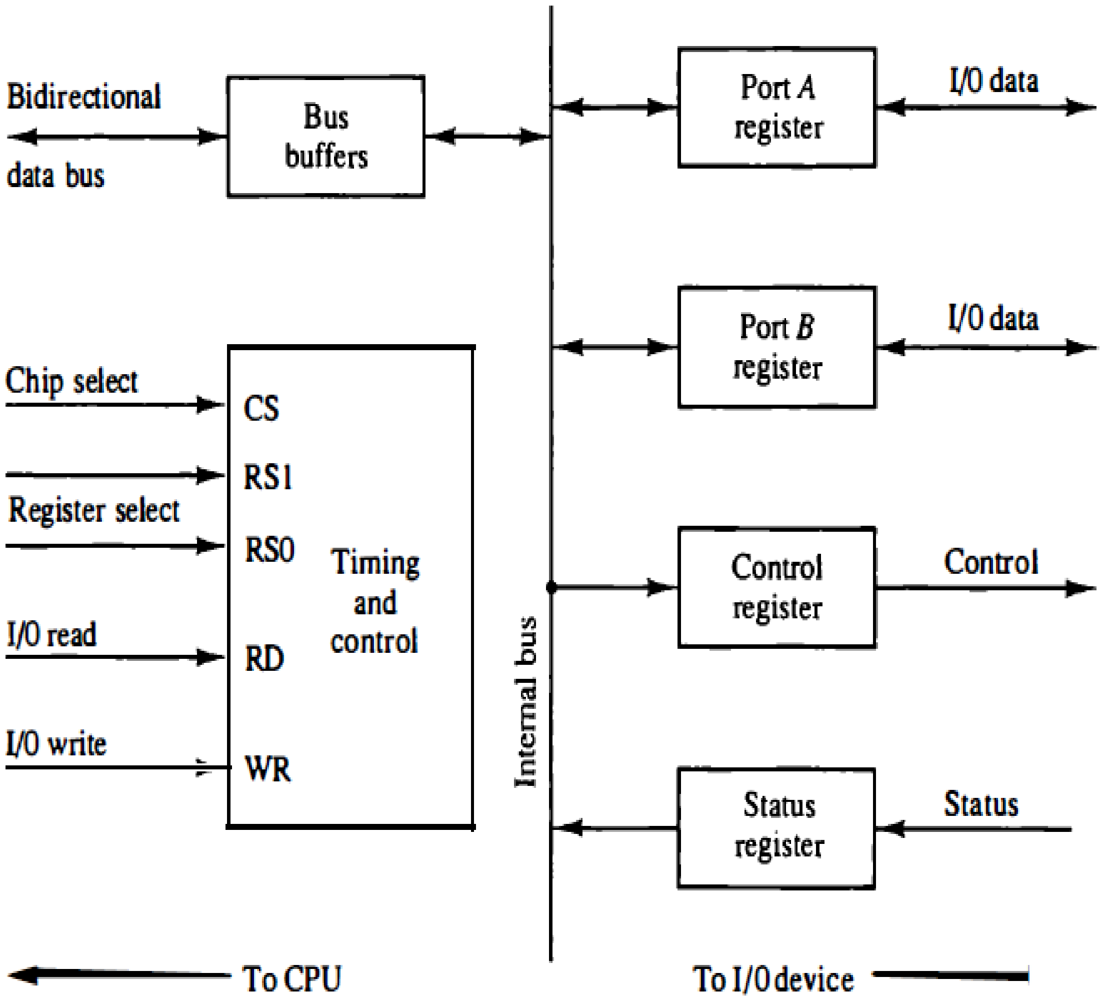
In the isolated I/O configuration, the CPU has distinct input and output instructions, and each of these instructions is associated with the address of an interface register. When the CPU fetches and decodes the operation code of an input or output instruction, it places the address associated with the instruction into the common address lines. At the same time, it enables the I/O read (for input) or I/O write (for output) control line.

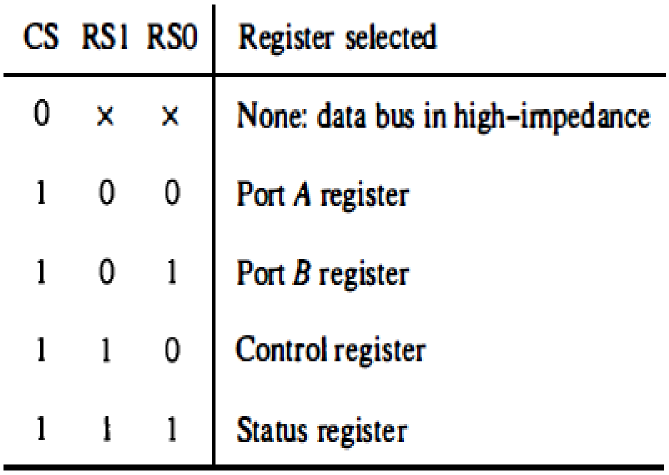
The isolated I/O method isolates memory word and not for an I/O addresses so that memory address values are not affected by interface address assignment since each has its own address space. The other alternative is to use the same address space for both memory and I/O. This is the case in computers that employ only one set of read and write signals and do not distinguish between memory and I/O addresses. This configuration is referred to as memory-mapped I/O.

In a memory-mapped I/O organization there are no specific input or output instructions. The CPU can manipulate I/O data residing in interface registers with the same instructions that are used to manipulate memory words. Each interface is organized as a set of registers that respond to read and write requests in the normal address space.

4.7.4. Example of I/O Interface

An example of an I/O interface unit is shown in block diagram form in Fig below.





CS	RS1	RS0	Register selected
0	×	×	None: data bus in high-impedance
1	0	0	Port *A* register
1	0	1	Port *B* register
1	1	0	Control register
1	1	1	Status register

It consists of two data registers called ports, a control register, a status register, bus buffers, and timing and control circuits. The interface communicates with the CPU through the data bus. The chip select and register select inputs determine the address assigned to the interface. The I/O read and write are two control lines that specify an input or output, respectively. The four registers communicate directly with the I/O device attached to the interface.

The I/O data to and from the device can be transferred into either port A or Port B. The interface may operate with an output device or with an input device, or with a device that requires both input and output. If the interface is connected to a printer, it will only output data, and if it services a character reader, it will only input data. A magnetic disk unit transfers data in both directions but not at the same time, so the interface can use bidirectional lines. A command is passed to the I/O device by sending a word to the appropriate interface register. In a system like this, the function code in the I/O bus is not needed because control is sent to the control register, status information is received from the status register, and data are transferred to and from ports A and B registers. Thus the transfer of data, control, and status information is always via the common data bus. The distinction between data, control, or status information is determined from the particular register with which the CPU communicates.

The interface registers communicate with the CPU through the bidirectional data bus. The address bus selects the interface unit through the chip select and the two register select inputs. A circuit must be provided externally (usually, a decoder) to detect the address assigned to the interface registers. This circuit enables the chip select (CS) input when the interface is selected by the address bus. The two register select inputs RS1 and RS0 are usually connected to the two least significant lines of the lines address bus. These two inputs select one of the four registers in the interface as specified in the table accompanying the diagram. The content of the selected register is transfer into the CPU via the data bus when the I/O read signal is enabled. The CPU transfers binary information into the selected register via the data bus when the I/O write input is enabled.

4.8. Asynchronous Data Transfer

The internal operations in a digital system are synchronized by means of clock pulses supplied by a common pulse generator. Clock pulses are applied to all registers within a unit and all data transfers among internal registers occur simultaneously during the occurrence of a clock pulse. Two units, such as a CPU and an I/O interface, are designed independently of each other. If the registers in the interface share a common clock with the CPU registers, the transfer between the two units is said to be synchronous. In most cases, the internal timing in each unit is independent from the other in that each uses its own private clock for internal registers. In that case, the two units are said to be asynchronous to each other. This approach is widely used in most computer systems.

Asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indicate the time at which data is being transmitted. One way of achieving this is by means of a strobe pulse supplied by one of the units to indicate to the other unit when the transfer has to occur. Another method commonly used is to accompany each data item being transferred with a control signal that indicates the presence of data in the bus. The unit receiving the data item responds with another control signal to acknowledge receipt of the data. This type of agreement between two independent units is referred to as handshaking.

The strobe pulse method and the handshaking method of asynchronous data transfer are not restricted to I/O transfers. In fact, they are used extensively on numerous occasions requiring the transfer of data between two independent units. In the general case we consider the transmitting unit as the source and the receiving unit as the destination.

For example, the CPU is the source unit during an output or a write transfer and it is the destination unit during an input or a read transfer. It is customary to specify the asynchronous transfer between two independent units by means of a timing diagram that shows the timing relationship that must exist between the control signals and the data in buses. The sequence of control during an asynchronous transfer depends on whether the transfer is initiated by the source or by the destination unit.

There are two types of asynchronous data transmission methods

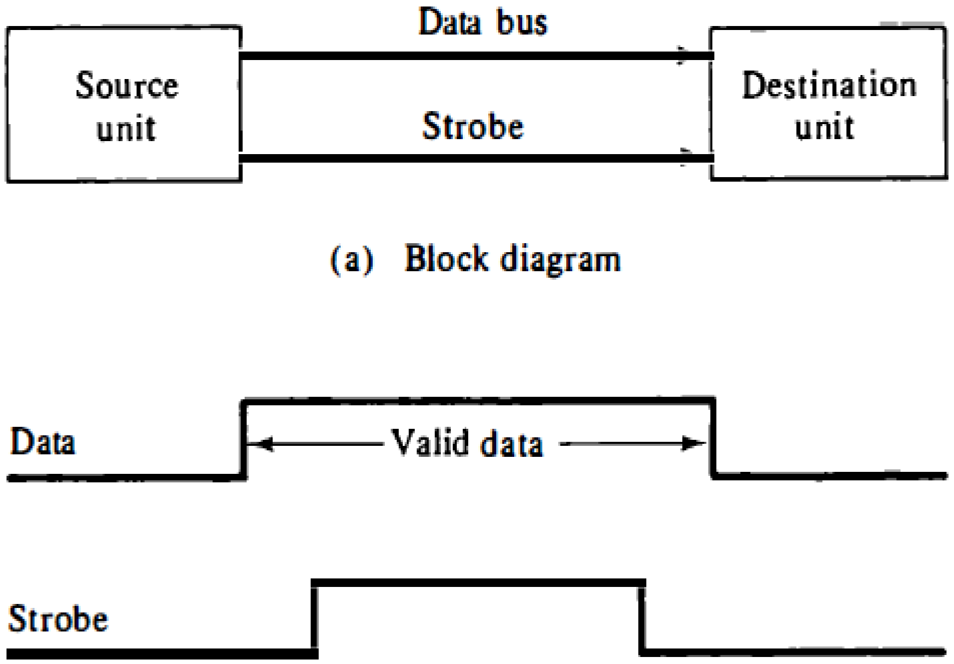
1. Strobe control
2. Handshaking

4.8.1. Strobe Control

The strobe control method of asynchronous data transfer employs a single control line to time each transfer.

The strobe may be activated by either the source or the destination unit.

i. source-initiated transfer.



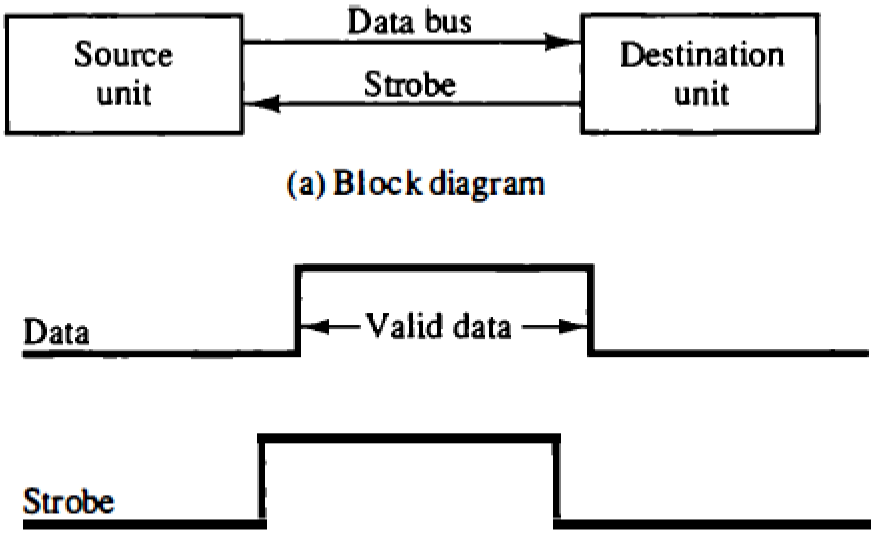
(a) Block diagram

(b) Timing diagram

The data bus carries the binary information from source unit to the destination unit. Typically, the bus has multiple lines to transfer an entire byte or word. The strobe is a single line that informs the destination unit when a valid data word is available in the bus.

The strode signal is given after a brief daly, after placing the data on the data bus. A brief period after the strobe pulse is disabled the source stops sending the data.

ii. Destination-initiated strobe for data transfer



(a) Block diagram

(b) Timing diagram

In this case the destination unit activates the strobe pulse, informing the source to provide the data. The source places the data on the data bus. The transmission id stopped briefly after the strobe pulse is removed.

The disadvantage of the strobe method is that the source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus. Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit has actually placed the data on the bus. This difficulty is solved by using hand shaking method of data transfer.

4.8.2. HANDSHAKING

The disadvantage of the strobe method is that the source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus. Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit has actually placed the data on the bus,.

The handshake method solves this problem by introducing a second control signal that provides a reply to the unit that initiates the transfer.

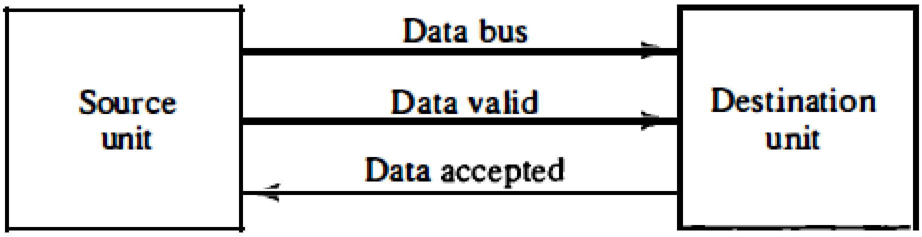
The basic principle of the two-write handshaking method of data transfer is as follows.

One control line is in the same direction as the data flow in the bus from the source to the destination. It is used by the source unit to inform the destination unit whether there are valued data in the bus.

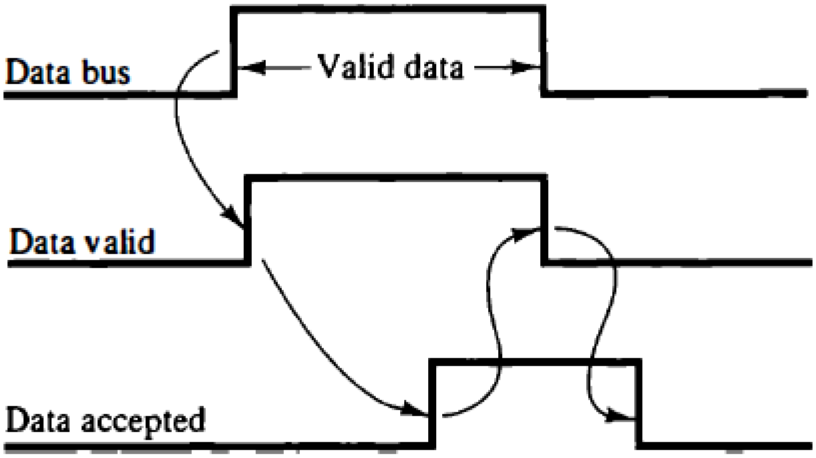
The other control line is in the other direction from the destination to the source. It is used by the destination unit to inform the source whether it can accept data. The sequence of control during the transfer depends on the unit that initiates the transfer.

Data Transfer Procedure When Initiated By the Source

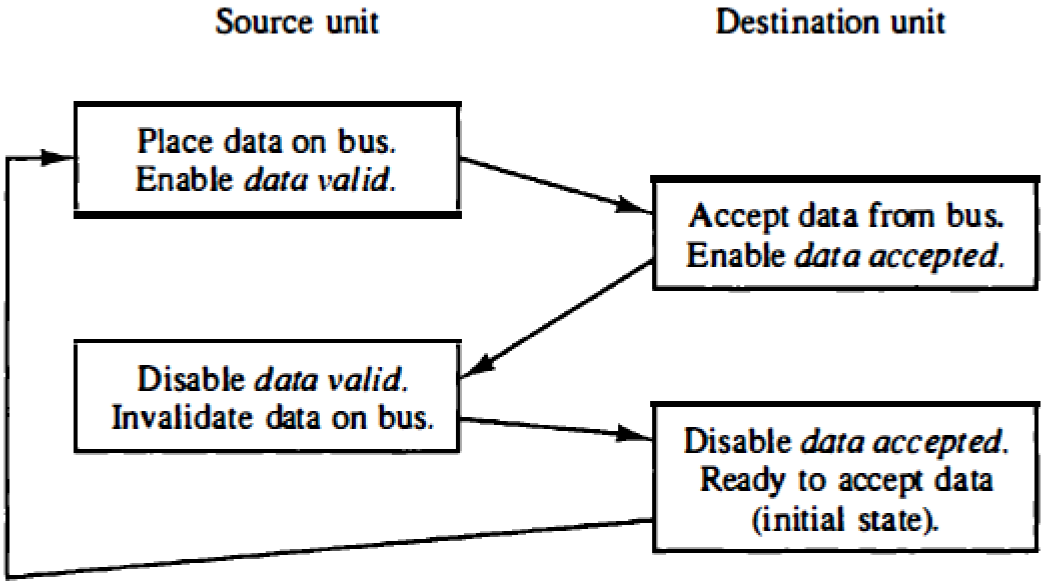
The two handshaking lines are data valid, which is generated by the source unit, and data accepted, generated by the destination unit. The timing diagram shows the exchange of signals between the two units. The sequence of events listed in part (c) shows the four possible states that the system



(a) Block diagram



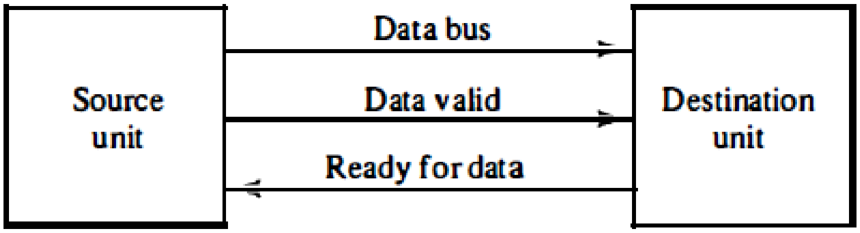
(b) Timing diagram



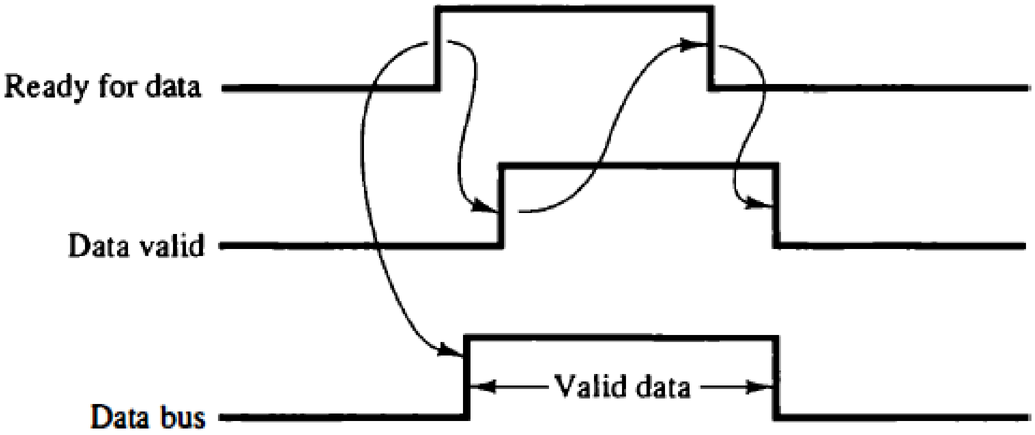
(c) Sequence of events

can be at any given time. The source unit initiates the transfer by placing the data on the bus and enabling its data valid signal. The data accepted signal is activated by the destination unit after it accepts the data from the bus. The source unit then disables its data valid signal, which invalidates the data on the bus.

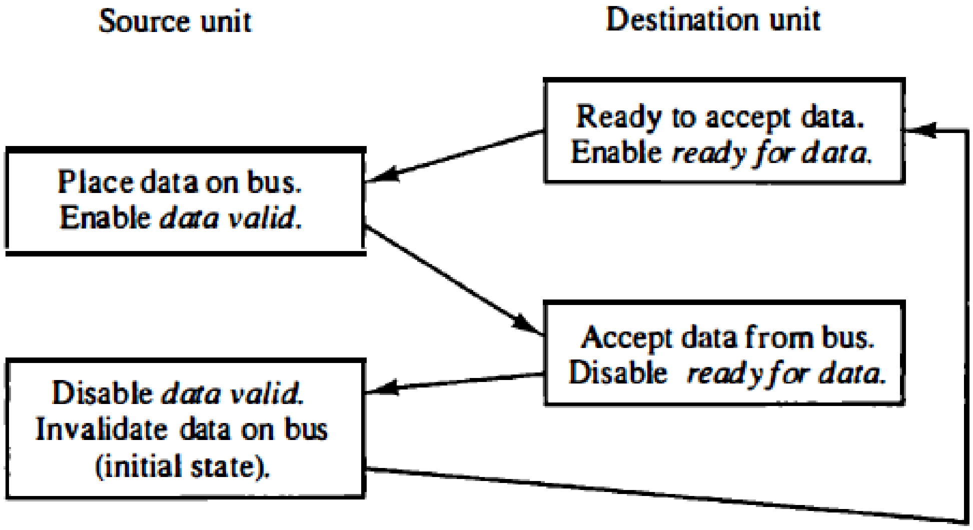
Destination -Initiated Transfer Using Handshaking



(a) Block diagram



(b) Timing diagram


Source unit
Destination unit
Ready to accept data. Enable *ready for data.*
Place data on bus. Enable *data valid.*
Accept data from bus. Disable *ready for data.*
Disable *data valid.* Invalidate data on bus (initial state).

(c) Sequence of events

The destination unit then disables its data accepted signal and the system goes into its initial state. The source dies not send the next data item until after the destination unit shows its readiness to accept new data by disabling its data accepted signal. This scheme allows arbitrary delays from one state to the next and permits each unit to respond at its own data transfer rate. The rate of transfer is determined by the slowest unit.

If one unit is faulty, the data transfer will not be completed. Such an error can be detected by means of a timeout mechanism, which produces an alarm if the data transfer is not completed within a predetermined time.

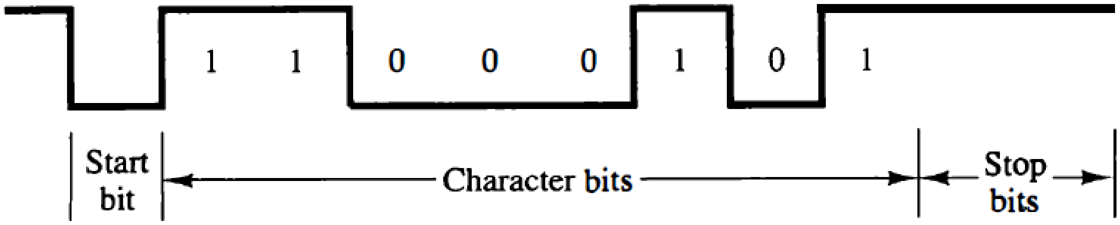
4.8.3. Asynchronous Serial Transfer

The transfer of data between two units may be done in parallel or serial. In parallel data transmission, each bit of the message has its own path and the total message is transmitted at the same time.

Serial transmission can be can be **synchronous or asynchronous**. In synchronous transmission, the two units share a common clock frequency and bits are transmitted continuously at the rate dictated by the clock pulses. In long-distant serial transmission, each unit is driven by a separate clock of the same frequency. Synchronization signals are transmitted periodically between the two units to keep their clocks in step with each other.

In asynchronous transmission, binary information is sent only when it is available and the line remains idle when there is no information to be transmitted. This is in contrast to synchronous transmission, where bits must be transmitted continuously to deep the clock frequency in both units synchronized with each other.

The convention is that the transmitter rests at the 1-state when no characters are transmitted. The first bit, called the start bit, is always a 0 and is used to indicate the beginning of a character. The last bit called the stop bit is always a 1. An example of this format is shown in Fig.



A transmitted character can be detected by the receiver from knowledge of the transmission rules:

1. When a character is not being sent, the line is kept in the 1-state.

2. The initiation of a character transmission is detected from the start bit, which is always 0.

3. The character bits always follow the start bit.

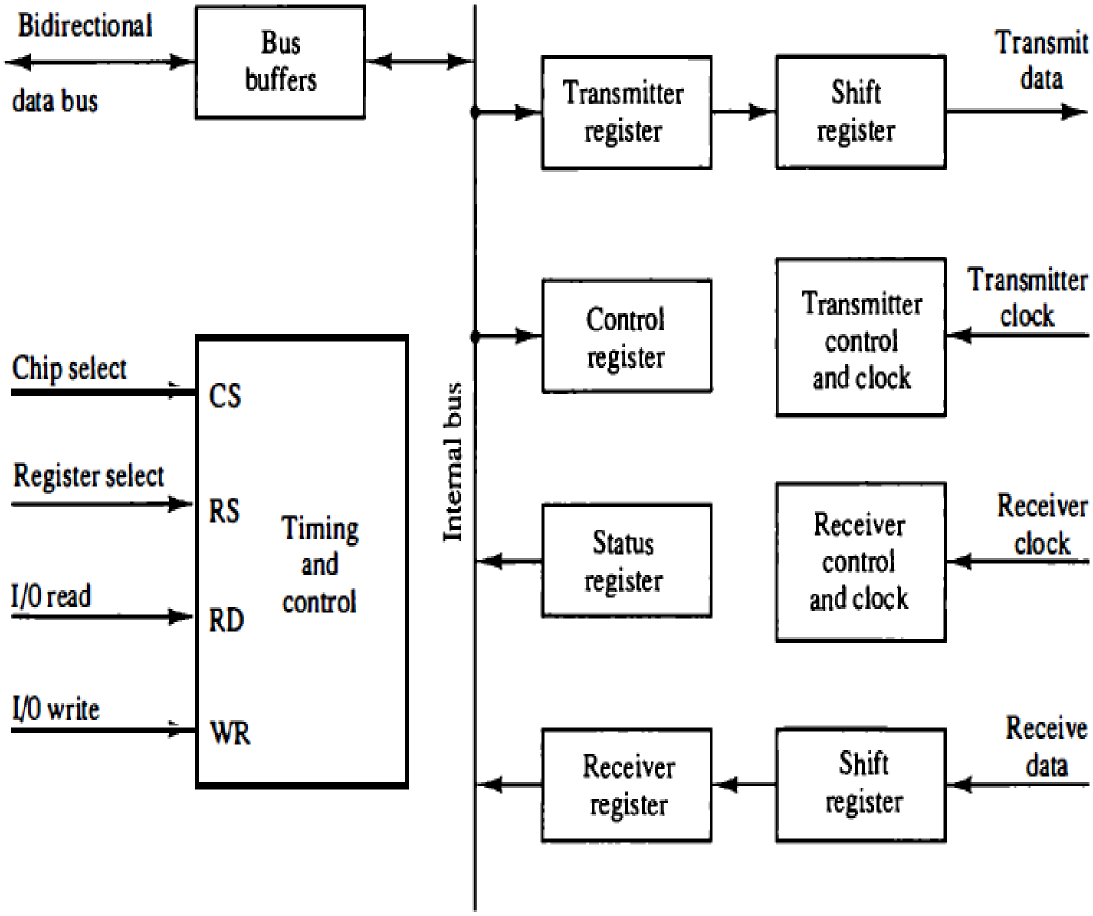
4. After the last bit of the character is transmitted, a stop bit is detected when the line returns to the 1-state for at least one bit time.

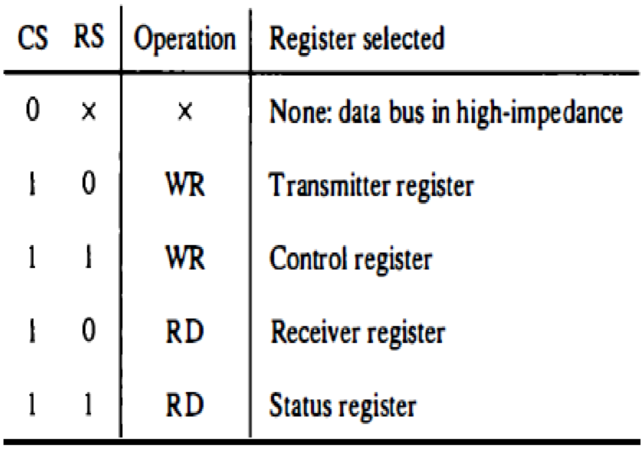
At the end of the character the line is held at the 1-state for a period of at least one or two bit times so that both the transmitter and receiver can resynchronize. The length of time that the line stays in this state depends on the amount of time required for the equipment to resynchronize. Some older electromechanical terminals use two stop bits, but newer terminals use one stop bit. The line remains in the 1-state until another character is transmitted. The stop time ensures that a new character will not follow for one or two bit times.

The baud rate is defined as the rate at which serial information is transmitted and is equivalent to the data transfer in bits per second. Ten characters per second with an 11-bit format has a transfer rate of 110 baud.

4.8.4. Asynchronous Communication Interface

The block diagram of an asynchronous communication interface is shown in below figure. It functions as both a transmitter and a receiver. The interface is initialized for a particular mode of transfer by means of a control byte that is loaded into its control register.





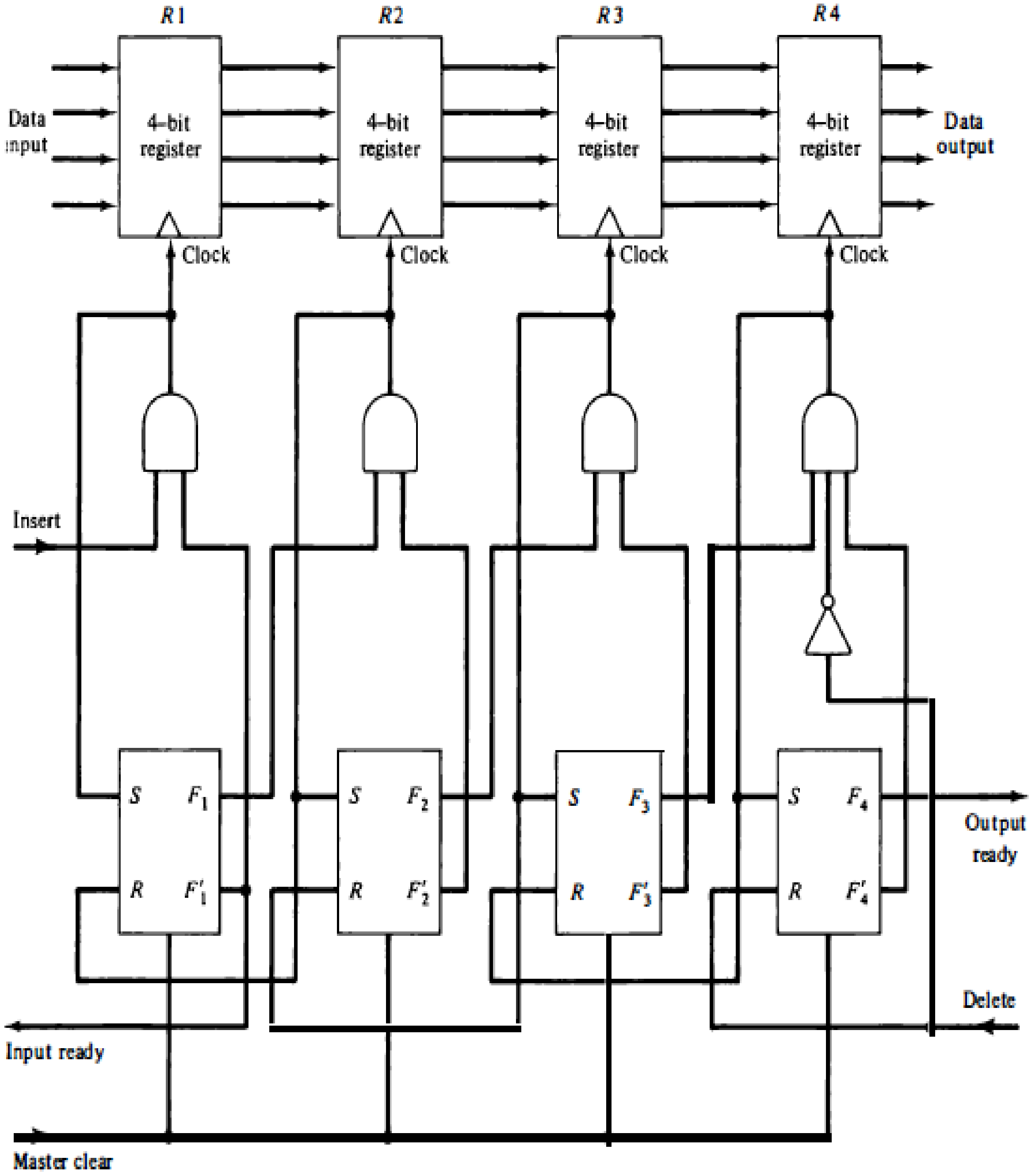
CS	RS	Operation	Register selected
0	×	×	None: data bus in high-impedance
1	0	WR	Transmitter register
1	1	WR	Control register
1	0	RD	Receiver register
1	1	RD	Status register

4.8.5. First –In, First – Out Buffer

a first –in ,first –out (FIFO) buffer is a memory unit that stores information in such a manner that the item first in is the item first out.

A FIFO buffer comes with separate input and output terminals. The important feature of this buffer is that it can input dat and output data at two different rates and the output data are always in the same order in which the data entered the buffer.

The logic diagram of a typical 4X4 FIFO buffer is shown below.



4.9. Modes of Transfer

Data transfer to and from peripherals may be handled in one of three possible modes:

1. Programmed I/O
2. Interrupt-initiated I/O
3. Direct memory access (DMA)

Programmed I/O operations are the result of I/O instructions written in the computer program.

When the interface determines that the device is ready for data transfer, it generates an interrupt request to the computer. Upon detecting the external interrupt signal, the CPU momentarily stops the task it is processing, branches to a service program to process the I/O transfer, and then returns to the task it was originally performing

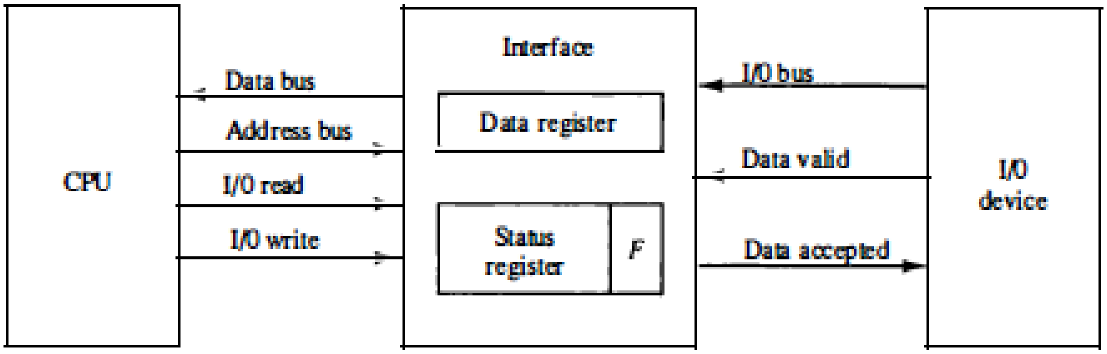
Transfer of data under programmed I/O is between CPU and peripheral. In direct memory access (DMA), the interface transfers data into and out of the memory unit through the memory bus. The CPU initiates the transfer by supplying the interface with the starting address and the number of words needed to be transferred and then proceeds to execute other tasks.

Many computers combine the interface logic with the requirements for direct memory access into one unit and call it an I/O processor (IOP). The IOP can handle many peripherals through a DMA and interrupt facility. In such a system, the computer is divided into three separate modules: the memory unit, the CPU, and the IOP.

4.9.1. Example of Programmed I/O

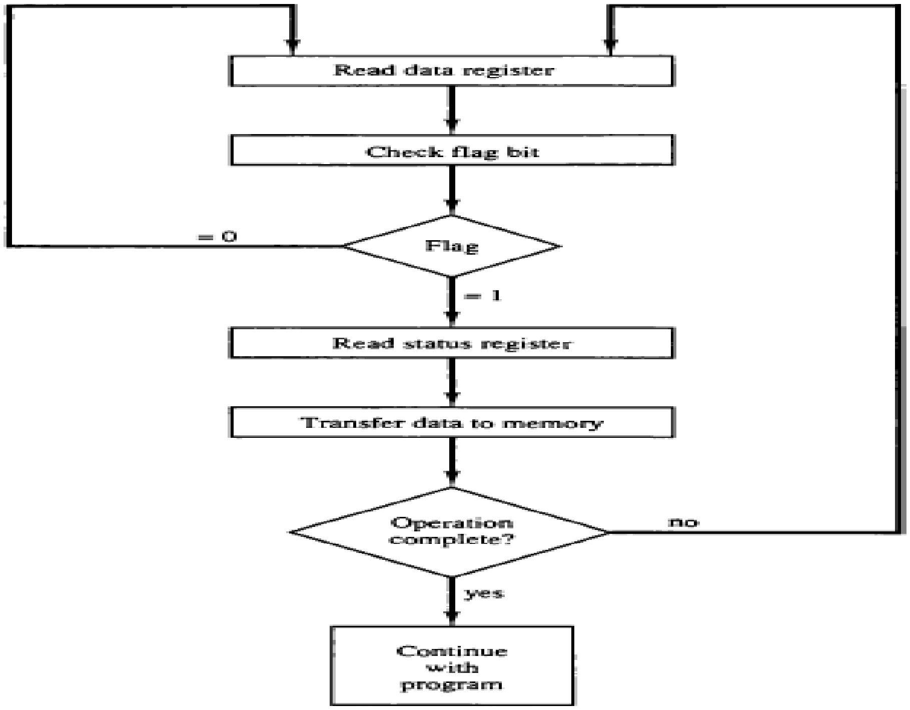
In the programmed I/O method, the I/O device dies not have direct access to memory. A transfer from an I/O device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from the device to the CPU, and a store instruction to transfer the data from the CPU to memory.

An example of data transfer from an I/O device through an interface into the CPU is shown in Fig.



F = Flag bit

The device transfers bytes of data one at a time as they are available. When a byte of data is available, the device places it in the I/O bus and enables its data valid line. The interface accepts the byte into its data register and enables the data accepted line. The interface sets a it in the status register that we will refer to as an F or “flag” bit. The device can now disable the data valid line, but it will not transfer another byte until the data accepted line is disabled by the interface. A flowchart of the program that must be written for the CPU is shown in Fig.



It is assumed that the device is sending a sequence of bytes that must be stored in memory. The transfer of each byte requires three instructions:

1. Read the status register.
2. Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.
3. Read the data register.

Each byte is read into a CPU register and then transferred to memory with a store instruction. A common I/O programming task is to transfer a block of words form an I/O device and store them in a memory buffer. A program that stores input characters in a memory buffer using the instructions mentioned in the earlier chapter.

4.9.2. Interrupt-Initiated I/O

The CPU responds to the interrupt signal by storing the return address from the program counter into a memory stack and then control branches to a service routine that processes the required I/O transfer. The way that the processor chooses the branch address of the service routine varies from tone unit to another. In principle, there are two methods for accomplishing this. One is called vectored interrupt and the other, no vectored interrupt. In a non vectored interrupt, the branch address is assigned to a fixed location in memory. In a vectored interrupt, the source that interrupts supplies the branch information to the computer. This information is called the **interrupt vector.**

4.9.3. Software Considerations

A computer must also have software routines for controlling peripherals and for transfer of data between the processor and peripherals. I/O routines must issue control commands to activate the peripheral and to check the device status to determine when it is ready for data transfer. Error checking and other useful steps often accompany the transfers. In interrupt-controlled transfers, the I/O software must issue commands to the peripheral to interrupt when ready and to service the interrupt when it occurs. In DMA transfer, the I/O software must initiate
the DMA channel to start its operation.

4.10. Priority Interrupt

Data transfer between the CPU and an I/O device is initiated by the CPU. However, the CPU cannot start the transfer unless the device is ready to communicate with the CPU.

A priority interrupts is a system that establishes a priority over the various sources to determine which condition is to be serviced first when two or more request arrive simultaneously. The system may also determine which conditions are permitted to interrupt the computer while another interrupt is being serviced. Higher-priority interrupt levels are assigned to request which, if delayed of interrupted, could have serious consequences. Devices with high-speed transfers such as keyboards receive low priority. When two devices interrupt the computer at the same time, the computer services the devices interrupt the computer at the same time, the computer services the device, with the higher priority first.

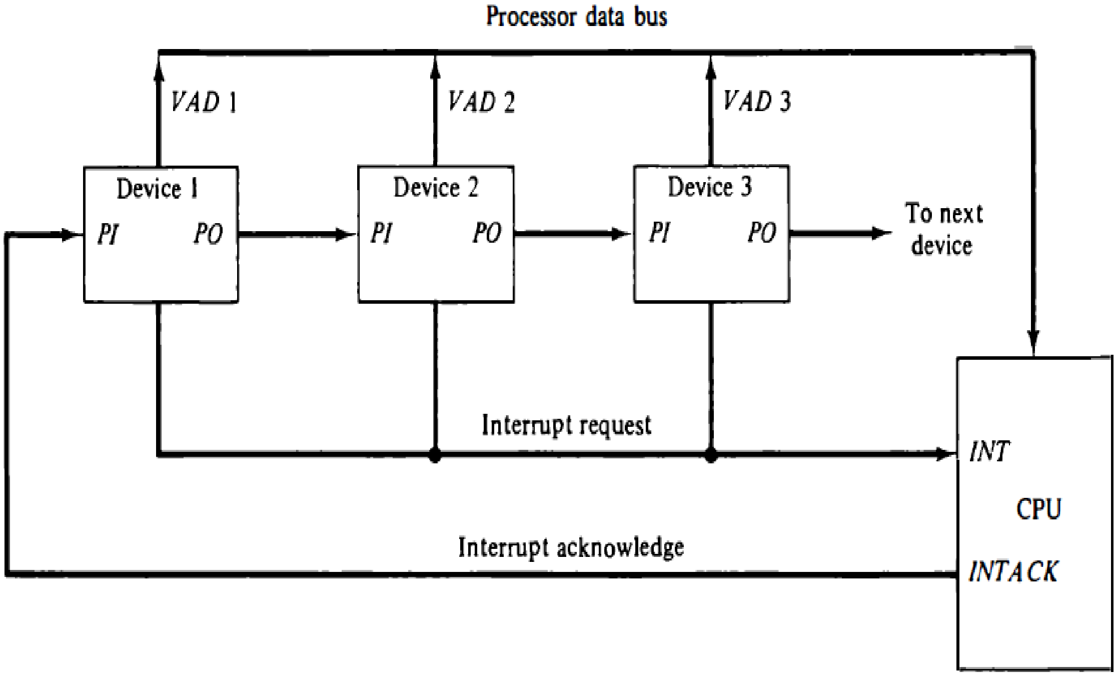
The disadvantage of the soft ware method is that if there are many interrupts, the time required to poll them can exceed the time available to service the I/O device. In this situation a hardware priority-interrupt unit can be used to speed up the operation.

A hardware priority-interrupt unit functions as an overall manager in an interrupt system environment. It accepts interrupt requests from many sources, determines which of the incoming requests has the highest priority, and issues an interrupt request to the computer based on this determination. To speed up the operation, each interrupt source has its own interrupt vector to access its own service routine directly. Thus no polling is required because all the decisions are established by the hardware priority-interrupt unit. The hardware priority function can be established by either a serial or a parallel connection of interrupt lines. The serial connection is also known as the **daisy chaining method.**

4.10.1. Daisy-Chaining Priority

The daisy-chaining method of establishing priority consists of a serial connection of all devices that request an interrupt.

The device with the highest priority is placed in the first position, followed by lower-priority devices up to the device with the lowest priority, which is placed last in the chain. This method of connection between three devices and the CPU is shown in Fig.



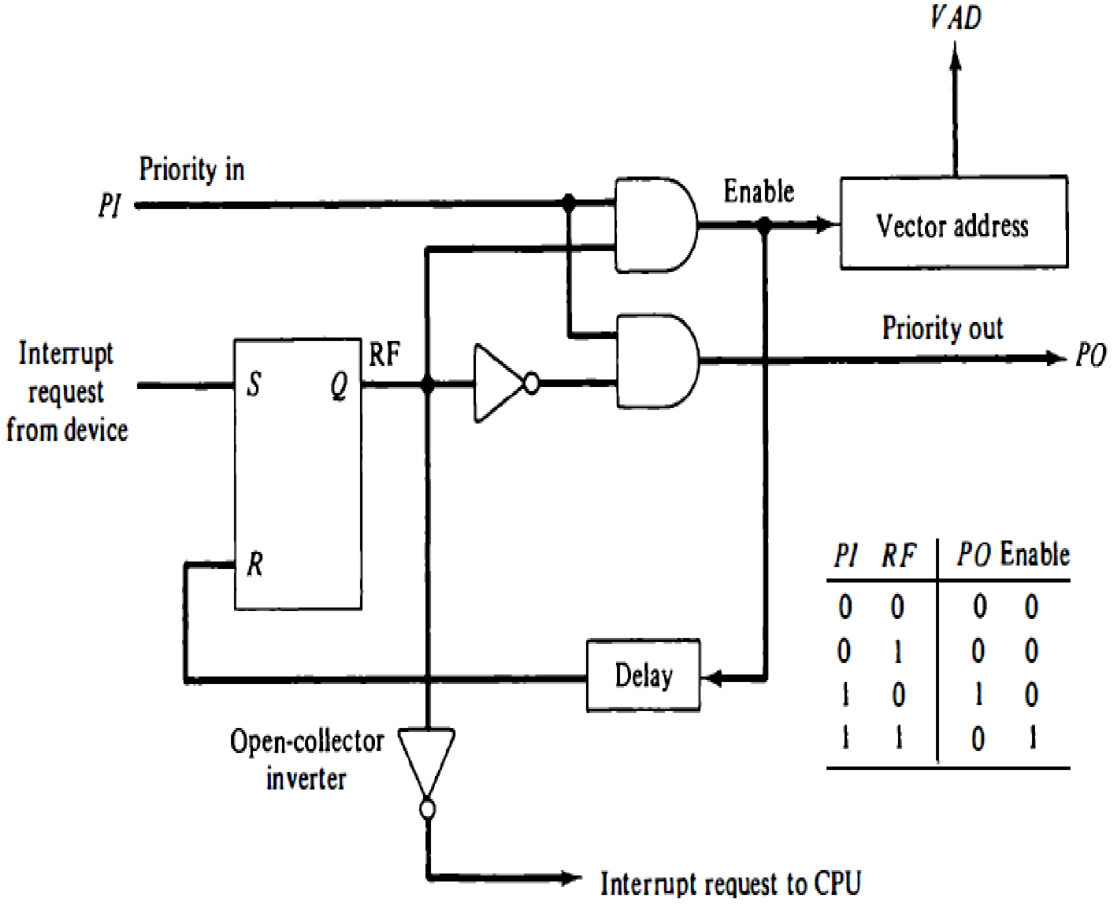
The interrupt request line is common to all devices and forms a wired logic connection. If any device has its interrupt signal in the low-level state, the interrupt line goes to the low-level state and enables the interrupt input in the CPU.

When no interrupts are pending, the interrupt line stays in the high-level state and no interrupts are recognized by the CPU.

This signal is received by device 1 at its PI (priority in) input. The acknowledge signal passes on to the next device through the PO (priority out) output only if device 1 is not requesting an interrupt. If device 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing a 0 in the PO output. It then proceeds to insert its own interrupt vector address (VAD) into the data bus for the CPU to use during the interrupt cycle.

A device with a 0 in its PI input generates a 0 in its PO output to inform the next-lower-priority device that the acknowledge signal has been blocked. A device that is requesting an interrupt and has a 1 in its PI input will intercept the acknowledge signal by placing a 0 in its PO output. If the device does not have pending interrupts, it transmits the acknowledge signal to the next device by placing a 1 in its PO output. Thus the device with PI = 1 and PO = 0 is the one with the highest priority that is requesting an interrupt, and this device places its VAD on the data bus. The daisy chain arrangement gives the highest priority to the device that receives the interrupt acknowledge signal from the CPU. The farther the device is from the first position, the lower is its priority.

Below figure shows the internal logic that must be included with in each device when connected in the daisy-chaining scheme.



PI	RF	PO	Enable
0	0	0	0
0	1	0	0
1	0	1	0
1	1	0	1

The device sets its RF flip-flop when it wants to interrupt the CPU. The output of the RF flip-flop goes through an open-collector inverter, a circuit that provides the wired logic for the common interrupt line. If PI = 0, both PO and the enable line to VAD are equal to 0, irrespective of the value of RF. If PI = 1 and RF = 0, then PO = 1 and the vector address is disabled. This condition passes the acknowledge signal to the next device through PO. The device is active when PI = 1 and RF = 1.

This condition places a 0 in PO and enables the vector address for the data bus. It is assumed that each device has its own distinct vector address. The RF flip-flop is reset after a sufficient delay to ensure that the CPU has received the vector address.

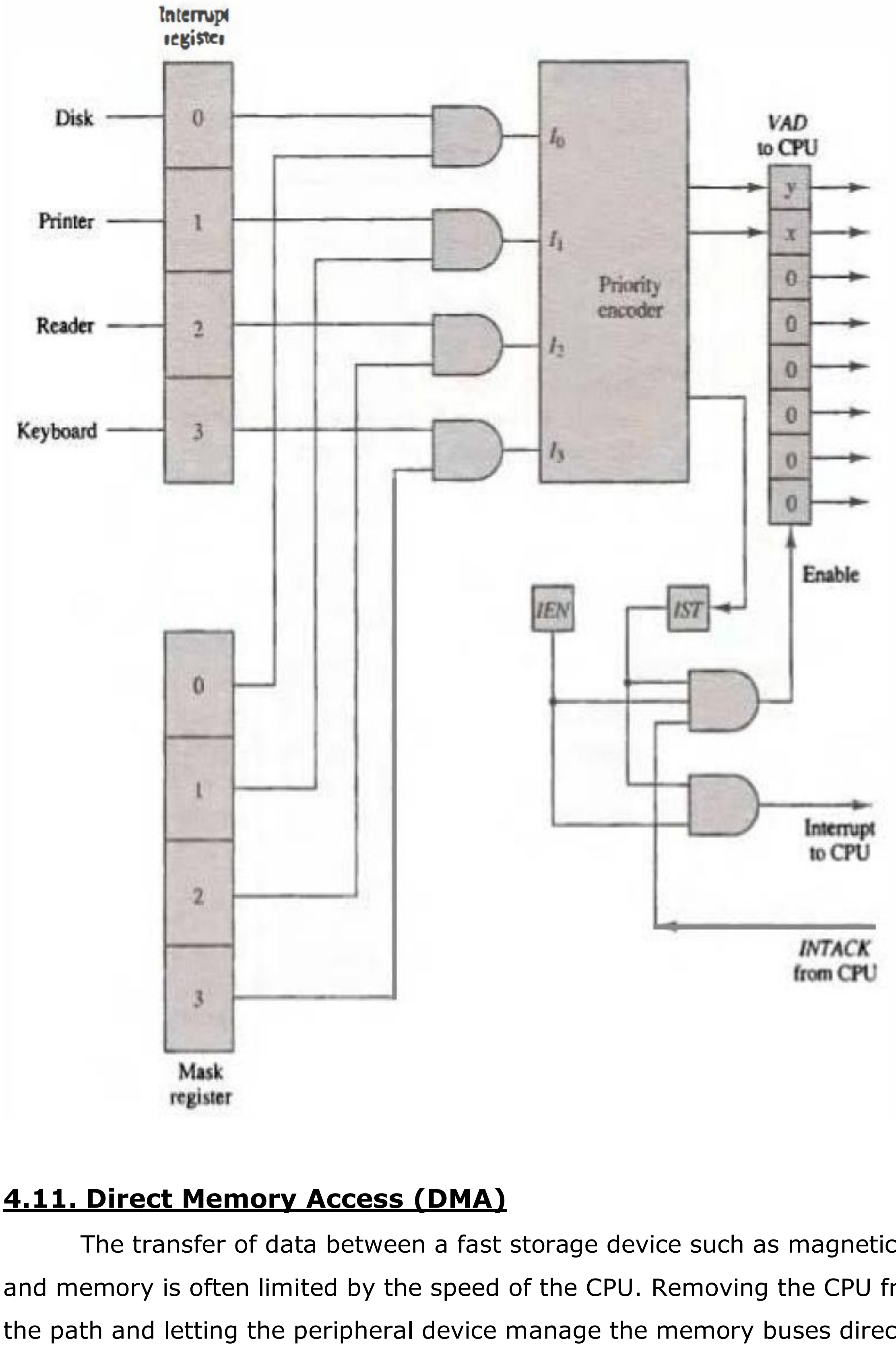
4.10.2. Parallel Priority Interrupt

The parallel priority interrupt method uses a register whose bits are set separately by the interrupt signal from each device. Priority is established according to the position of the bits in the register. In addition to the interrupt register the circuit may include a mask register whose purpose is to control the status of each interrupt request.

The mask register can be programmed to disable lower-priority interrupts while a higher-priority device is being serviced. It can also provide a facility that allows a high-priority device to interrupt the CPU while a lower-priority device is being serviced.

The priority logic for a system of four interrupt sources is shown in Fig.

It consists of an interrupt register whose individual bits are set by external conditions and cleared by program instructions. The magnetic disk, being a high-speed device, is given the highest priority. The printer has the next priority, followed by a character reader and a keyboard. The mask register has the same number of bits as the interrupt register. By means of program instructions, it is possible to set or reset any bit in the mask register.



4.11. Direct Memory Access (DMA)

The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer. This transfer technique is called **direct memory access (DMA).** During DMA transfer, the CPU is idle and has no control of the memory buses. A DMA controller takes over the buses to manage the transfer directly between the I/O device and memory.

The CPU may be placed in an idle state in a variety of ways. One common method extensively used in microprocessors is to disable the buses through special control signals. Following figure shows two control signals in the CPU that facilitate the DMA transfer.

The **bus request** (BR) input is used by the DMA controller to request the CPU to relinquish control of the buses. When this input is active, the CPU terminates the execution of the current instruction and places the address bus, the data bus, and the read and write lines into a high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have a logic significance.

The CPU activates the **Bus grant (BG)** output to inform the external DMA that the buses are in the high -impedance state.

When the DMA terminates the transfer, it disables the bus request line. The CPU disables the bus grant, takes control of the buses, and returns to its normal operation.

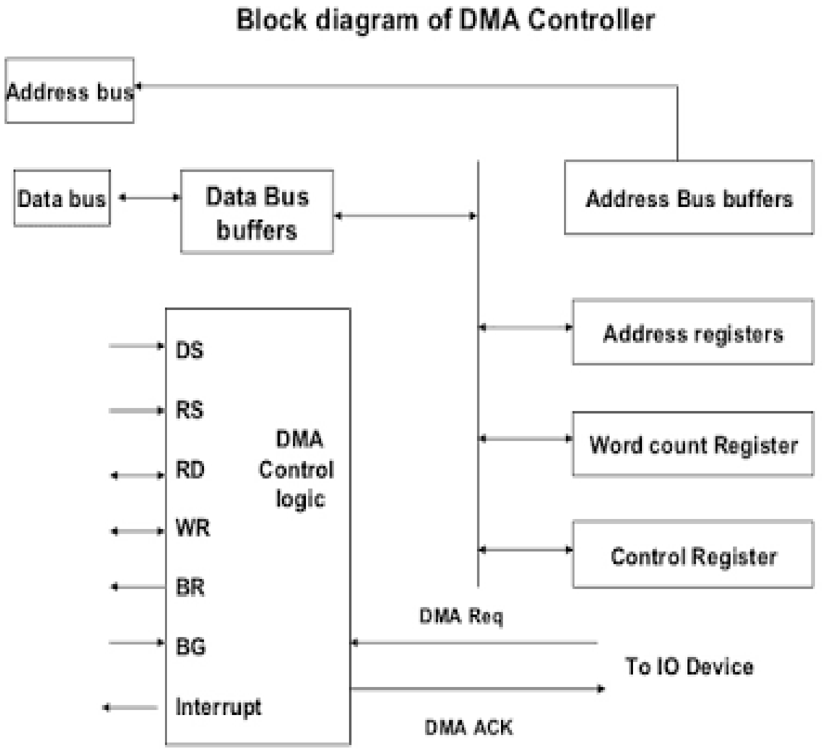
This mode of transfer is needed for fast devices such as magnetic disks, where data transmission cannot be stopped or slowed down until an entire block is transferred. An alternative technique **called cycle stealing** allows the DMA controller to transfer one data word at a time after which it must return control of the buses to the CPU.

4.11.1. DMA Controller

The DMA controller needs the usual circuits of an interface to communicate with the CPU and I/O device. In addition, it needs an address register, a word count register, and a set of address lines.

The word count register specifies the number of words that must be transferred. The data transfer may be done directly between the device and memory under control of the DMA.

Following figure shows the block diagram of a typical DMA controller.



The unit communicates with the CPU via the data bus and control lines. The registers in the DMA are selected by the CPU through the address bus by enabling the DS (DMA select) and RS (register select) inputs. The RD (read) and WR (write) inputs are bidirectional. When the BG (bus grant) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to the DMA registers. When BG = 1, the CPU has relinquished the buses and the DMA can communicate directly with the memory by specifying an address in the address bus and activating the RD or WR control. ; the DMA communicates with the external peripheral through the request and acknowledge lines by using a prescribed handshaking procedure.

The DMA controller has three registers: an address register, a word count register, and a control register. The **address register** contains an address to specify the desired location in memory. The address bits go through bus buffers into the address bus. The address register is incremented after each word that is transferred to memory.

The **word count register** is incremented after each word that is transferred to memory. The word count register holds the number of words to be transferred. This register is decremented by one after each word transfer and internally tested for zero.

The **control register** specifies the mode of transfer. All registers in the DMA appear to the CPU as I/O interface registers. Thus the CPU can read from or write into the DMA registers under program control via the data bus.

The CPU initializes the DMA by sending the following information through the data bus:

1. The starting address of the memory block where data are available (for read) or where data are to be stored (for write)
2. The word count, which is the number of words in the memory block
3. Control to specify the mode of transfer such as read or write
4. A control to start the DMA transfer

4.11.2. DMA Transfer

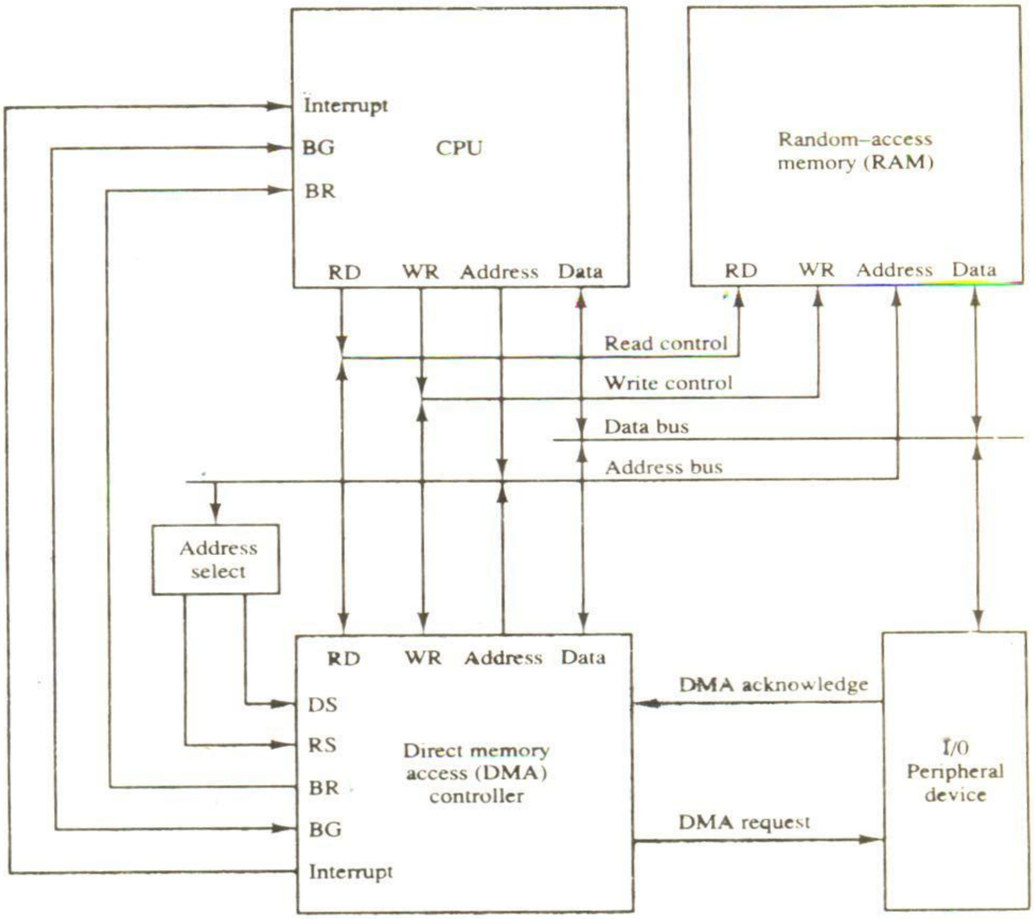
The position of the DMA controller among the other components in a computer system is illustrated in Fig. The CPU communicates with the DMA through the address and data buses as with any interface unit. The DMA has its own address, which activates the DS and RS lines.

The CPU initializes the DMA through the data bus. Once the DMA receives the start control command, it can start the transfer between the peripheral device and the memory.

When the peripheral device sends a DMA request, the DMA controller activates the BR line, informing the CPU to relinquish the buses. The CPU responds with its BG line, informing the DMA that its buses are disabled.

The DMA then puts the current value of its address register into the address bus, initiates the RD or WR signal, and sends a DMA acknowledge to the peripheral device. Note that the RD and WR lines in the DMA controller are bidirectional. The direction of transfer depends on the status of the BG line. When BG line. When BG = 0, the RD and WR are input lines allowing the CPU to communicate with the internal DMA registers. When BG = 1, the RD and WR and output lines from the DMA controller to the random-access memory to specify the read or write operation for the data.

When the peripheral device receives a DMA acknowledge, it puts a word in the data us (for write) or receives a word from the data bus (for read). Thus the DMA controls the read or write operations and supplies the address for the memory. The peripheral unit can then communicate with memory through the data bus for direct transfer between the two units while the CPU is momentarily disabled.



MEMORY ORGANIZATION

4.1. MEMORY HIERARCHY

The memory unit is an essential component in any digital computer since it is needed for storing programs and data. A very small computer with a limited application may be able to fulfill its intended task without the need of additional storage capacity. Most general-purpose computers would run more efficiently if they were equipped with additional storage beyond the capacity of the main memory. There is just not enough space in one memory unit to accommodate all the programs used in a typical computer. Moreover, most computer users accumulate and continue to accumulate large amounts of data-processing software.

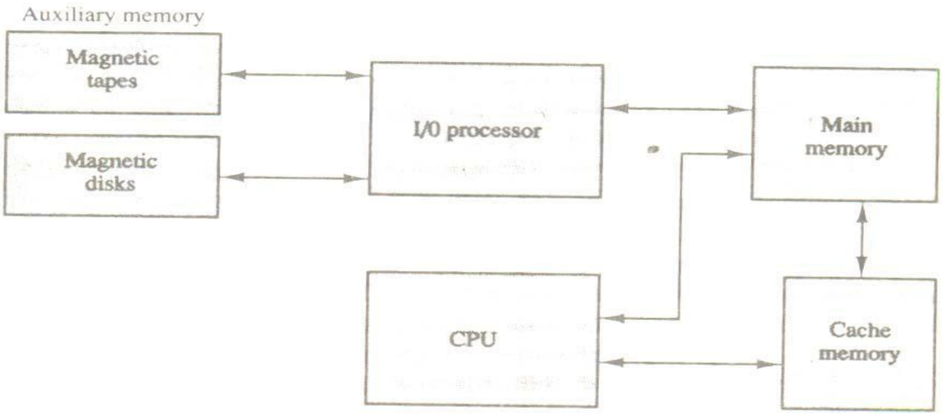
The memory unit that communicates directly with the CPU is called the main memory.

Devices that provide backup storage are called auxiliary memory. The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. They are used for storing system programs, large data files, and other backup information.

Only programs and data currently needed by the processor reside in main memory. All other information is stored in auxiliary memory and transferred to main memory when needed.

The total memory capacity of a computer can be visualized as being a hierarchy of components.

A special very-high speed memory called a cache is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate. The cache memory is employed in computer systems to compensate for the speed differential between main memory access time and processor logic. CPU logic is usually faster than main memory access time, with the result that processing speed is limited primarily by the speed of main memory.



Memory hierarchy in a computer system.

Making programs and data available at a rapid rate, it is possible to increase the performance rate of the computer.

While the I/O processor manages data transfers between auxiliary memory and main memory, the cache organization is concerned with the transfer of information between main memory and CPU. Thus each is involved with a different level in the memory hierarchy system. The reason for having two or three levels of memory hierarchy is economics.

As the storage capacity of the memory increases, the cost per bit for storing binary information decreases and the access time of the memory becomes longer.

The auxiliary memory has a large storage capacity, is relatively inexpensive, but has low access speed compared to main memory. The cache memory is very small, relatively expensive, and has very high access speed.

4.2. MAIN MEMORY

The main memory is the central storage unit in a computer system. It is a relatively large and fast memory used to store programs and data during the computer operation.

The principal technology used for the main memory is based on semiconductor integrated circuits. Integrated circuit RAM chips are available in two possible operating modes, **static and dynamic**. The static RAM consists essentially of internal flip-flops that store the binary information. The stored information remains valid as long as power is applied to unit.

The dynamic RAM stores the binary information in the form of electric charges that are applied to capacitors. The stored charge on the capacitors tend to discharge with time and the capacitors must be periodically recharged by refreshing the dynamic memory.

Refreshing is done by cycling through the words every few milliseconds to restore the decaying charge. The dynamic RAM offers reduced power consumption and larger storage capacity in a single memory chip.

The static RAM is easier to use and has shorted read and write cycles.Originally, RAM was used to refer to a random-access memory, but now it is used to designate a read/write memory to distinguish it from a read-only memory, although ROM is also random access. RAM is used for storing the bulk of the programs and data that are subject to change. ROM is used for storing programs that are permanently resident in the computer and for tables of constants that do not change in value one the production of the computer is completed.

The ROM portion of main memory is needed for storing an initial program called a bootstrap loader. The bootstrap loader is a program whose function is to start the computer software operating when power is turned on.

Since RAM is volatile, its contents are destroyed when power is turned off. The contents of ROM remain unchanged after power is turned off and on again transferred to the operating system, which prepares the computer for general use.

RAM and ROM chips are available in a variety of sizes. If the memory needed for the computer is larger than the capacity of one chip, it is necessary to combine a number of chips to form the required memory size

4.2.1. RAM AND ROM CHIPS

A RAM chip is better suited for communication with the CPU if it has one or more control inputs that select the chip only when needed. Another common feature is a bidirectional data bus that allows the transfer of data either from memory to CPU during a read operation or from CPU to memory during a write operation.

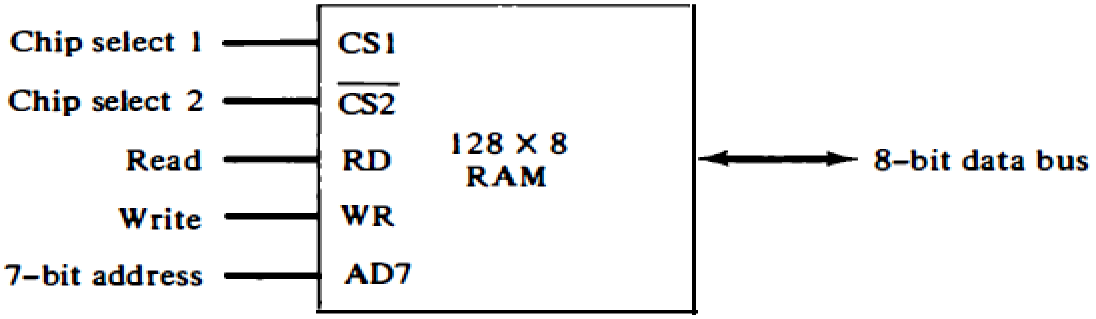
A bidirectional bus can be constructed with three-state buffers. A three-state buffer output can be placed in one of three possible states: a signal equivalent to logic 1, a signal equivalent to logic 0, or a high-impedance state. The logic 1 and 0 are normal digital signals. The high-impedance state behaves like an open circuit, which means that the output does not carry a signal and has no logic significance.

The block diagram of a RAM chip is shown in Fig .The capacity of the memory is 128 words of eight bits (one byte) per word. This requires a 7-bit address and an 8-bit bidirectional data bus.

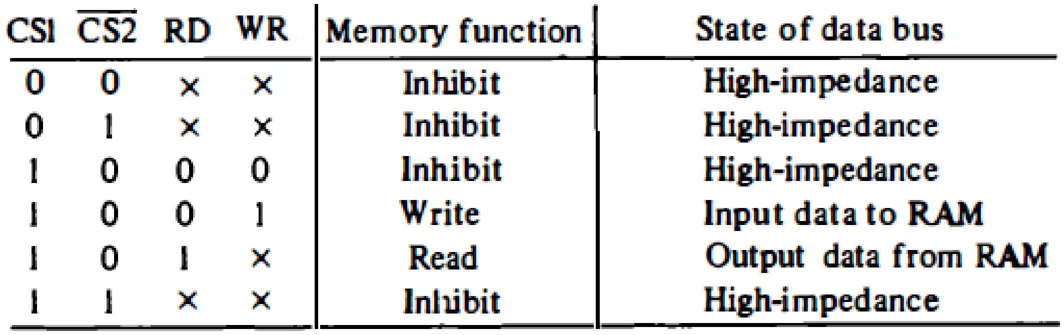
The read and write inputs specify the memory operation and the two chips select (CS) control inputs are for enabling the chip only when it is selected by the microprocessor.

The availability of more than one control input to select the chip facilitates the decoding of the address lines when multiple chips are used in the microcomputer.

The read and write inputs are sometimes combined into one line labeled R/W. When the chip is selected, the two binary states in this line specify the two operations or read or write.



(a) Block diagram

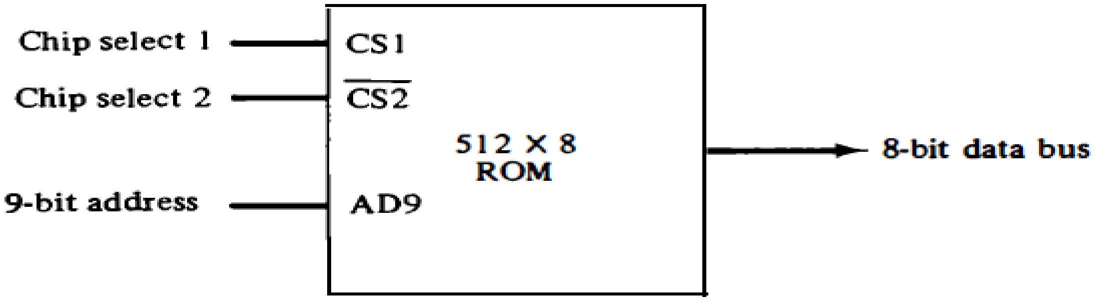


CS1	$\overline{CS2}$	RD	WR	Memory function	State of data bus
0	0	×	×	Inhibit	High-impedance
0	1	×	×	Inhibit	High-impedance
1	0	0	0	Inhibit	High-impedance
1	0	0	1	Write	Input data to RAM
1	0	1	×	Read	Output data from RAM
1	1	×	×	Inhibit	High-impedance

(b) Function table

The function table listed in Fig(b) specifies the operation of the RAM chip. The unit is in operation only when CSI = 1 and CS2 = 0. The bar on top of the second select variable indicates that this input in enabled when it is equal to 0. If the chip select inputs are not enabled, or if they are enabled but the read but the read or write inputs are not enabled, the memory is inhibited and its data bus is in a high-impedance state. When SC1 = 1 and CS2 = 0, the memory can be placed in a write or read mode. When the WR input is enabled, the memory stores a byte from the data bus into a location specified by the address input lines. When the RD input is enabled, the content of the selected byte is placed into the data bus. The RD and WR signals control the memory operation as well as the bus buffers associated with the bidirectional data bus.

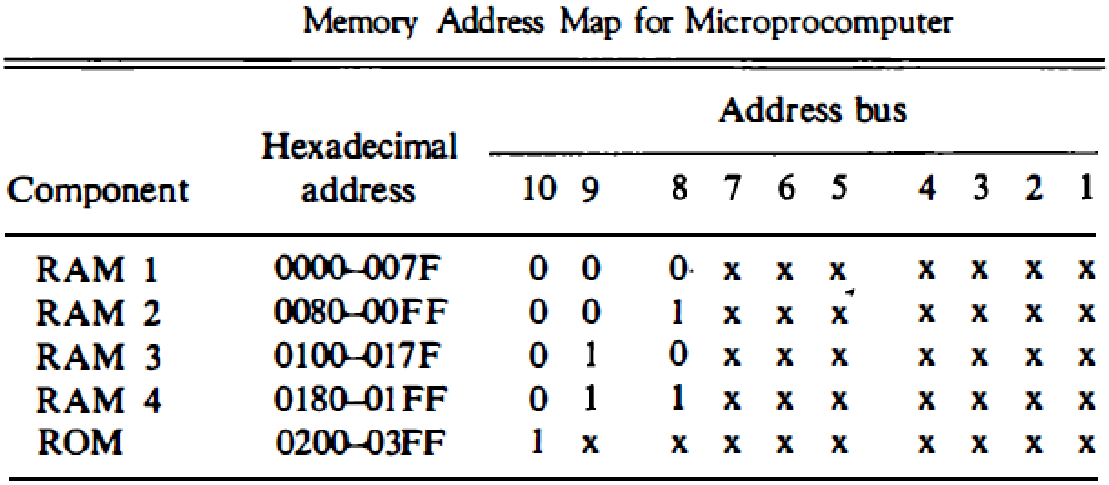
A ROM chip is organized externally in a similar manner. However, since a ROM can only read, the data bus can only be in an output mode. The block diagram of a ROM chip is shown in Fig. For the same-size chip, it is possible to have more bits of ROM occupy less space than in RAM. For this reason, the diagram specifies a 512-byte ROM, while the RAM has only 128 bytes.



The nine address lines in the ROM chip specify any one of the 512 bytes stored in it. The two chip select inputs must be CS1 = 1 and CS2 = 0 for the unit to operate. Otherwise, the data bus is in a high-impedance state. There is no need for a read or write control because ~~the~~ unit can only read. Thus when the chip is enabled by the two select inputs, the byte selected by the address lines appears on the data bus.

4.2.2. MEMORY ADDRESS MAP

The designer of a computer system must calculate the amount of memory required for the particular application and assign it to either RAM or ROM. The interconnection between memory and processor is then established form knowledge of the size of memory needed and the type of RAM and ROM chips available. The addressing of memory can be established by means of a table that specifies the memory address assigned to each chip. The table, called a memory address map, is a pictorial representation of assigned address space for each chip in the system. To demonstrate with a particular example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM. The RAM and ROM chips



Memory Address Map for Microprocomputer

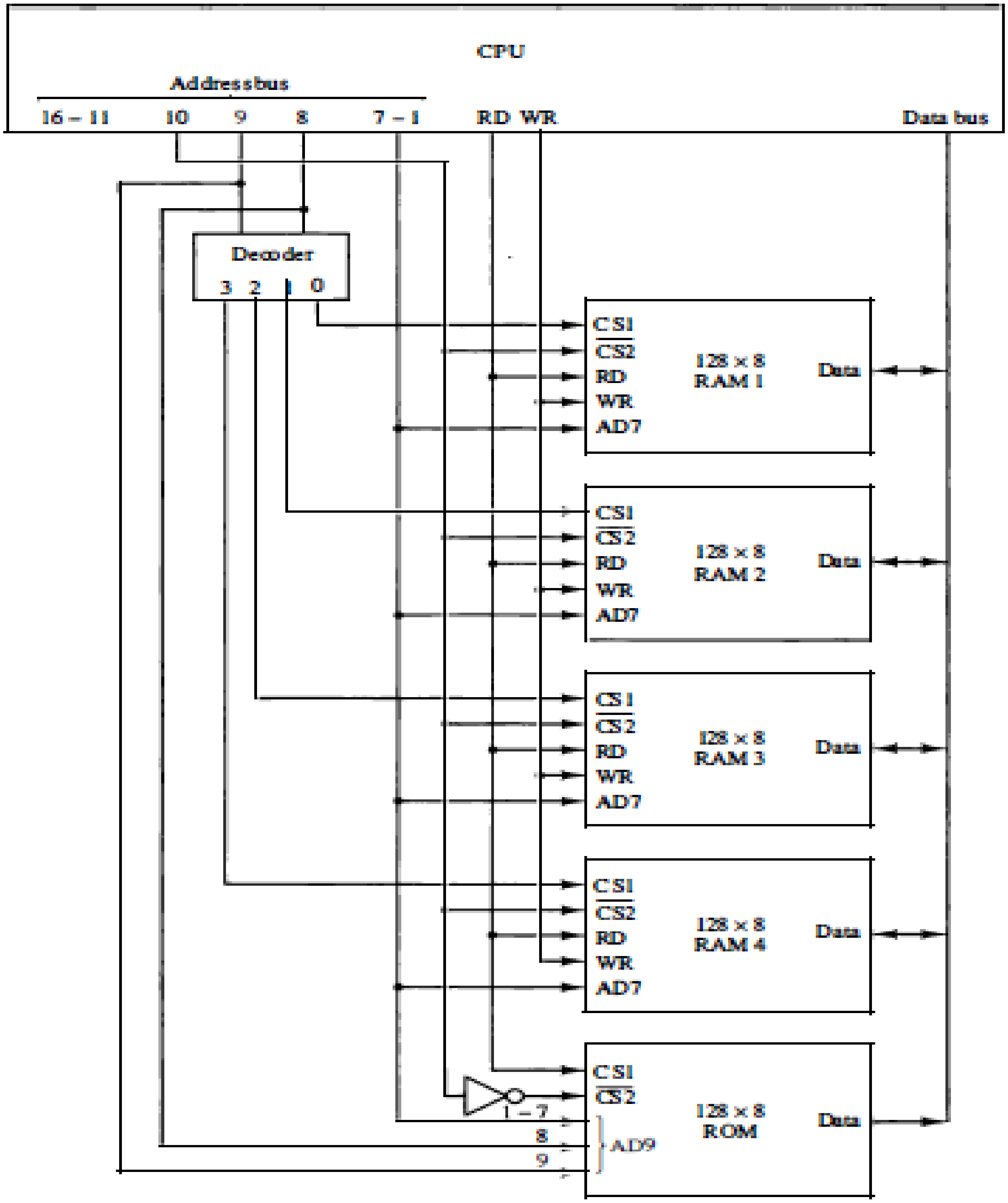
Component	Hexadecimal address	Address bus									
		10	9	8	7	6	5	4	3	2	1
RAM 1	0000–007F	0	0	0	x	x	x	x	x	x	x
RAM 2	0080–00FF	0	0	1	x	x	x	x	x	x	x
RAM 3	0100–017F	0	1	0	x	x	x	x	x	x	x
RAM 4	0180–01FF	0	1	1	x	x	x	x	x	x	x
ROM	0200–03FF	1	x	x	x	x	x	x	x	x	x

The component column specifies whether a RAM or a ROM chip is used. The hexadecimal address column assigns a range of hexadecimal equivalent addresses for each chip. The address bus lines are listed in the third column. Although there are 16 lines in the address bus, the table shows only 10 lines because the other 6 are not used in this example and are assumed to be zero. The small x's under the address bus lines designate those lines that must be connected to the address inputs in each chip. The RAM chips have 128 bytes and need seven address lines. The ROM chip has 512 bytes and needs 9 address lines. The x's are always assigned to the low-order bus lines: lines 1 through 7 for the RAM and lines 1 through 9 for the ROM. It is now necessary to distinguish between four RAM chips by assigning to each a different address. For this particular example we choose bus lines 8 and 9 to represent four distinct binary combinations. Note that any other pair of unused bus lines can be chosen for this purpose. The table clearly shows that the nine low-order bus lines constitute a memory space fro RAM equal to 2^9 = 512 bytes. The distinction between a RAM and ROM address is done with another bus line. Here we choose line 10 for this purpose. When line 10 is 0, the CPU selects a RAM, and when this line is equal to 1, it selects the ROM.

The equivalent hexadecimal address for each chip is obtained form the information under the address bus assignment. The address bus lines are subdivided into groups of four bits each so that each group can be represented with a hexadecimal digit. The first hexadecimal digit represents lines 13 to 16 and is always 0. The next hexadecimal digit represents lines 9 to 12, but lines 11 and 12 are always 0. The range of hexadecimal addresses for each component is determined from the x's associated with it. These x's represent a binary number that can range from an all-0's to an all-1's value.

4.2.3. MEMORY CONNECTION TO CPU

RAM and ROM chips are connected to a CPU through the data and address buses. The low-order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through its chip select inputs. The connection of memory chips to the CPU is shown in Fig. This configuration gives a memory capacity of 512 bytes of RAM and 512 bytes of ROM. Each RAM receives the seven low-order bits of the address bus to select one of 128 possible bytes. The particular RAM chip selected is determined from lines 8 and 9 in the address bus. This is done through a 2X4 decoder whose outputs go to the SCI input in each RAM chip. Thus, when address lines 8 and 9 are equal to 00, the first RAM chip is selected. When 01, the second RAM chip is selected, and so on. The RD and WR outputs from the microprocessor are applied to the inputs of each RAM chip.



The selection between RAM and ROM is achieved through bus line 10. The RAMs are selected when the bit in this line is 0, and the ROM when the bit is 1. The other chip select input in the ROM is connected to the RD control line for the ROM chip to be enabled only during a read operation. Address bus lines 1 to 9 are applied to the input address of ROM without going through the decoder. This assigns addresses 0 to 511 to RAM and 512 to 1023 to ROM. The data bus of the ROM has only an output capability, whereas the data bus connected to the RAMs can transfer information in both directions.

4.3. AUXILIARY MEMORY

The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. Other components used, but not as frequently, are magnetic drums, magnetic bubble memory, and optical disks. To understand fully the physical mechanism of auxiliary memory devices one must have a knowledge of magnetics, electronics, and electromechanical systems. Although the physical properties of these storage devices can be quite complex, their logical properties can be characterized and compared by a few parameters. The important characteristics of any device are its access mode, access time, transfer rate, capacity, and cost.

The average time required to reach a storage location in memory and obtain its contents is called the **access time.** In electromechanical devices with moving parts such as disks and tapes, the access time consists of a seek time required to position the read-write head to a location and a transfer time required to transfer data to or from the device. Because the seek time is usually much longer than the transfer time, auxiliary storage is organized in records or blocks.

A record is a specified number of characters or words. Reading or writing is always done on entire records. The transfer rate is the number of characters or words that the device can transfer per second, after it has been positioned at the beginning of the record.

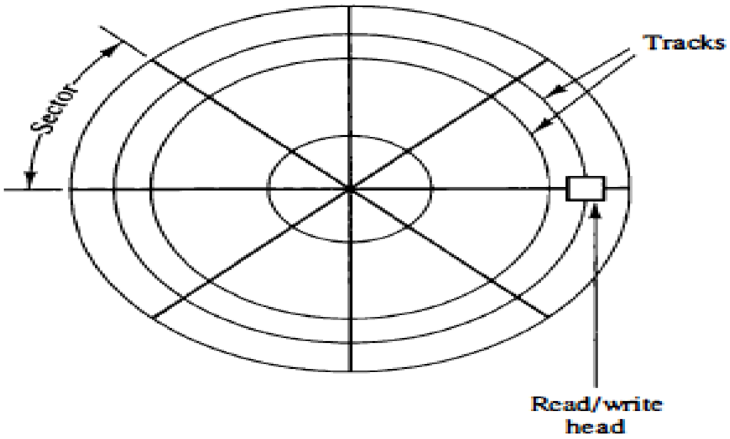
Magnetic drums and disks are quite similar in operation. Both consist of high-speed rotating surfaces coated with a magnetic recording medium. The rotating surface of the drum is a cylinder and that of the disk, a round flat plate. The amount of surface available for recording in a disk is greater than in a drum of equal physical size. Therefore, more information can be stored on a disk than on a drum of comparable size. For this reason, disks have replaced drums in more recent computers

4.3.1. MAGNETIC DISKS

A magnetic disk is a circular plate constructed of metal or plastic coated with magnetized material. Often both sides of the disk are used and several disks may be stacked on one spindle with read/write heads available on each surface. All disks rotate together at high speed and are not stopped or started from access purposes. Bits are stored in the magnetized surface in spots along concentric circles called tracks. The tracks are commonly divided into sections called **sectors.** In most systems, the minimum quantity of information which can be transferred is a sector. The sub division of tone disk surface into tracks and sectors.

Some units use a single read/write head from each disk surface and some different read/write head. The address can then select a particular track electronically through a decoder circuit. This type of unit is more expensive and is found only in very large computer systems.

Disks may have multiple heads and simultaneous transfer of bits from several tracks at the same time.



Disks that are permanently attached to the unit assembly and cannot be removed by the occasional user are called hard disks. A disk drive with removable disks is called a floppy disk. The disks used with a floppy disk drive are small removable disks made of plastic coated with magnetic recording material. There are two sizes commonly used, with diameters of 5.25 and 3.5 inches. The 3.5-inch disks are smaller and can store more data than can the 5.25-inch disks. Floppy disks are extensively used in personal computers as a medium for distributing software to computer users.

4.3.2. MAGNETIC TAPE

A magnetic tape transport consists of the electrical, mechanical, and electronic components to provide the parts and control mechanism for a magnetic-tape unit. The tape itself is a strip of plastic coated with a magnetic recording medium. Bits are recorded as magnetic spots on the tape along several tracks. Usually, seven or nine bits are recorded simultaneously to form a character together with a parity bit. Read/write heads are mounted one in each track so that data can be recorded and read as a sequence of characters.

Magnetic tape units can be stopped, started to move forward or in reverse, or can be rewound. However, they cannot be started or stopped fast enough between individual characters. For this reason, information is recorded in blocks referred to as records. Gaps of unrecorded tape are inserted between records where the tape can be stopped.

4.4. ASSOCIATIVE MEMORY

Many data-processing applications require the search of items in a table stored in memory. An assembler program searches the symbol address table in order to extract the symbol’s binary equivalent. An account number may be searched in a file to determine the holder’s name and account status.

The established way to search a table is to store all items where they can be addressed in sequence. The search procedure is a strategy for choosing a sequence of addresses, reading the content of memory at each address, and comparing the information read with the item being searched until a match occurs. The number of accesses to memory depends on the location of the item and the efficiency of the search algorithm.

The time required to find an item stored in memory can be reduced considerably if stored data can be identified for access by the content of the data itself rather than by an address. A memory unit accessed by content is called **an associative memory or content addressable memory (CAM).**

This type of memory is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location. When a word is written in an associative memory, no address is given. The memory is capable of finding an empty unused location to store the word. When a word is to be read from an associative memory, the content of the word, or part of the word, is specified.

The memory locaters all words which match the specified content and marks them for reading. Because of its organization, the associative memory is uniquely suited to do parallel searches by data association.

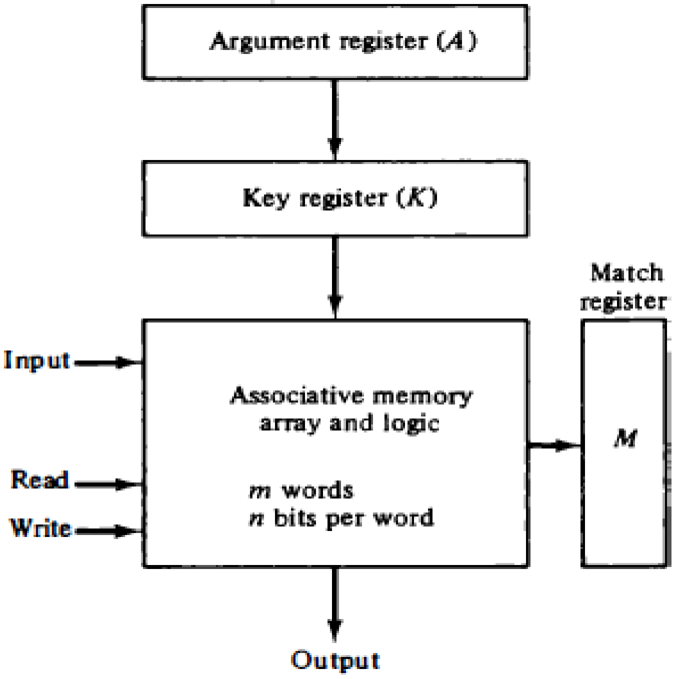
An associative memory is more expensive then a random access memory because each cell must have storage capability as well as logic circuits for matching its content with an external argument.

For this reason, associative memories are used in applications where the search time is very critical and must be very short.

4.4.1. HARDWARE ORGANIZATION

The block diagram of an associative memory consists of a memory array and logic from words with n bits per word. The argument register A and key register K each have n bits, one for each bit of a word.

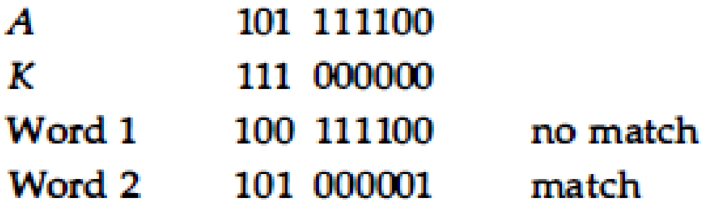
Block diagram for Associate Memory



The match register M has m bits, one for each memory word. Each word in memory is compared in parallel with the content of the argument register. The words that match the bits of the argument register set a corresponding bit in the match register. After the matching process, those bits in the match register that have been set indicate the fact that their corresponding words have been matched. Reading is accomplished by a sequential access to memory for those words whose corresponding bits in the match register have been set.

The key register provides a mask for choosing a particular field or key in the argument word. The entire argument is compared with each memory word if the key register contains all 1's. Otherwise, only those bits in the argument that have 1's in their corresponding position of the key register are compared. Thus the key provides a mask or identifying piece of information which specifies how the reference to memory is made.

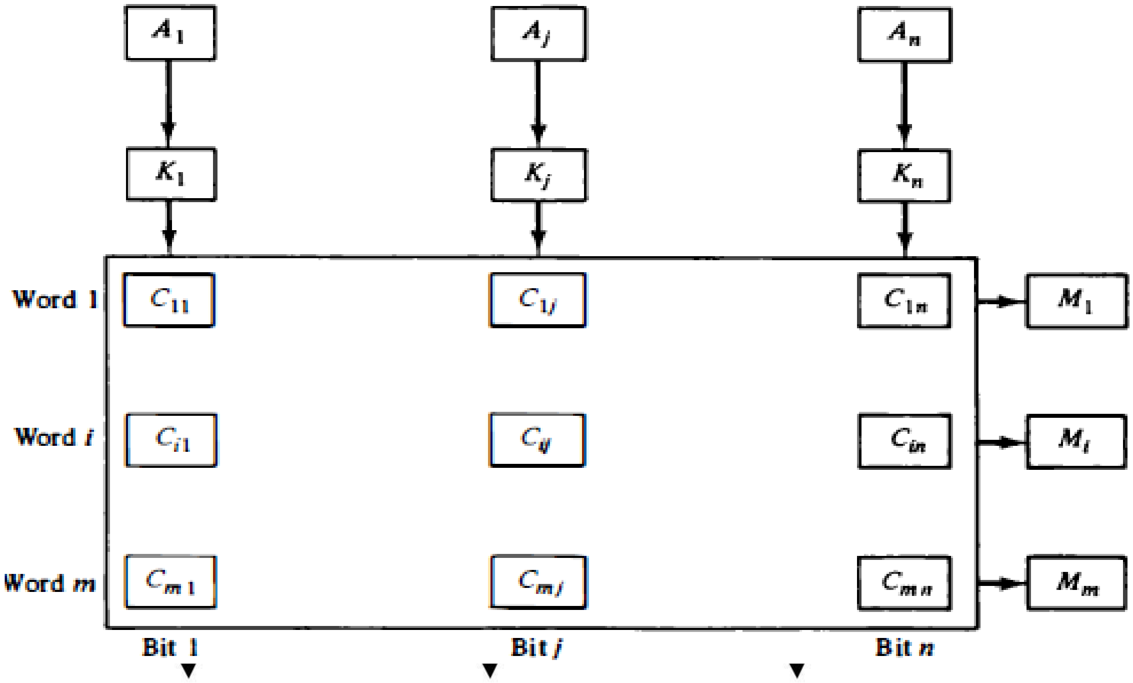
To illustrate with a numerical example, suppose that the argument register A and the key register K have the bit configuration shown below. Only the three leftmost bits of A are compared with memory words because K has 1's in these positions.



A	101 111100	
K	111 000000	
Word 1	100 111100	no match
Word 2	101 000001	match

Word 2 matches the unmasked argument field because the three leftmost bits of the argument and the word are equal.

The relation between the memory array and external registers in an associative memory is shown in below figure.



The cells in the array are marked by the letter C with two subscripts. The first subscript gives the word number and the second specifies the bit position in the word. Thus cell C_{ij} is the cell for bit j in word i. A bit A_j in the argument register is compared with all the bits in column j of the array provided that K_j = 1. This is done for all columns j = 1, 2,…,n. If a match occurs between all the unmasked bits of the argument and the bits in word i, the corresponding bit M_i in the match register is set to 1.

If one or more unmasked bits of the argument and the word do not match, M_i is cleared to 0.

Flop storage element F_{ij} and the circuits for reading, writing, and matching the cell. The input bit is transferred into the storage cell during a write operation. The bit stored is read out during a read operation. The match logic compares the content of the storage cell with the corresponding unmasked bit of the argument and provides an output for the decision logic that sets the bit in M_i.

4.4.2. MATCH LOGIC

The match logic for each word can be derived from the comparison algorithm for two binary numbers. First, we neglect the key bits and compare the argument in A with the bits stored in the cells of the words. Word i is equal to the argument in A if A_j = F_{ij} for j = 1, 2,…, n. Two bits are equal if they are both 1 or both 0. The equality of two bits can be expressed logically by the Boolean function

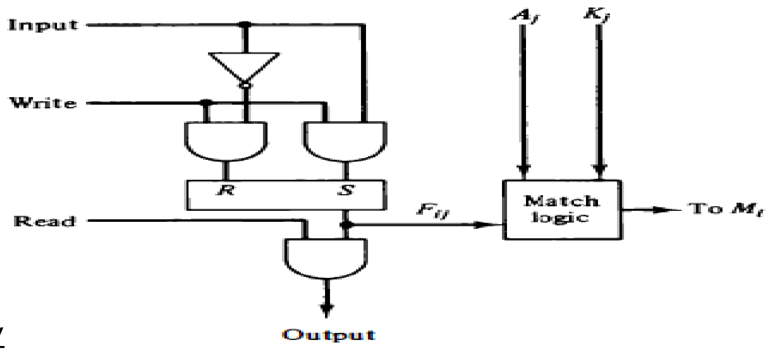
$$x_j = A_j F_{ij} + A'_j F_{ij}'$$

where $x_j = 1$ if the pair of bits in position j are equal; otherwise, $x_j = 0$.

For a word i to be equal to the argument in A we must have all x_j variables equal to 1. This is the condition for setting the corresponding match bit M_i to 1. The Boolean function for this condition is

$$M_i = x_1 \ x_2 \ x_3 \ \dots \ x_n$$

and constitutes the AND operation of all pairs of matched bits in a word.



One cell of associative memory

We now include the key bit K_j in the comparison logic. The requirement is that if $K_j = 0$, the corresponding bits of A_j and F_{ij} need no comparison. Only when $K_j = 1$ must they be compared. This requirement is achieved by ORing each term with K_j' , thus:

$$x_j + Kj = \begin{cases} x_j & \text{if } K_j = 1 \\ 1 & \text{if } K_j = 0 \end{cases}$$

When $K_j = 1$, we have $K_j' = 0$ and $x_j + 0 = x_j$. When $K_j = 0$, then $K_j' = 1$ $x_j + 1 = 1$. A term $(x_j + K_j')$ will be in the 1 state if its pair of bitsis not compared. This is necessary because each term is ANDed with all other terms so that an output of 1 will have no effect. The comparison of the bits has an effect only when $K_j = 1$. The match logic for word i in an associative memory can now be expressed by the following Boolean function:

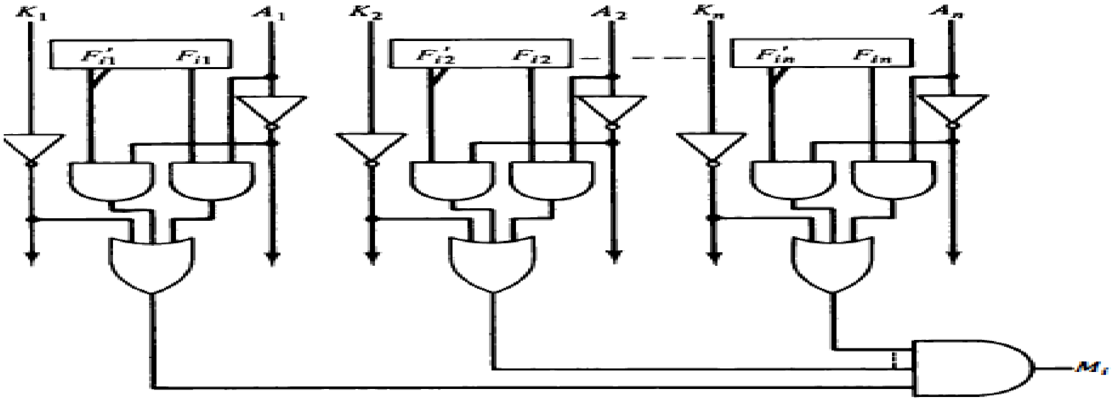
$$\mathbf{M_i = (x_1 + K'_j)\,(x_2 + K'_j)\,(x_3 + K'_j) \ .... \ (x_n + K'_j)}$$

Each term in the expression will be equal to 1 if its corresponding $K'_j = 0$. if $K_j = 1$, the term will be either 0 or 1 depending on the value of x_j. A match will occur and M_i will be equal to 1 if all terms are equal to 1. If we substitute the original definition of x_j. the Boolean function above can be expressed as follows:

$$M_i = \prod_{j=1}^{n} (A_j F_{ij} + A_j' F_{ij}' + K_j')$$

Where ∏ is a product symbol designating the AND operation of all n terms. We need m such functions, one for each word i = 1, 2, 3, …., m.

The circuit for catching one word is shown in below figure.



Each cell requires two AND gates and one OR gate. The inverters for A_j and K_j are needed once for each column and are used for all bits in the column. The output of all OR gates in the cells of the same word go to the input of a common AND gate to generate the match signal for M_i. M_i will be logic 1 if a catch occurs and 0 if no match occurs. Note that if the key register contains all 0's, output M_i will be a 1 irrespective of the value of A or the word. This occurrence must be avoided during normal operation.

4.4.3. READ OPERATION

The matched words are read in sequence by applying a read signal to each word line whose corresponding M_i bit is a 1. In most applications, the associative memory stores a table with no two identical items under a given key. In this case, only one word may match the unmasked argument field. By connecting output M_i directly to the read line in the same word position (instead of the M register), the content of the matched word will be presented automatically at the output lines and no special read command signal is needed. Furthermore, if we exclude words having a zero content, an all-zero output will indicate that no match occurred and that the searched item is not available in memory.

4.4.4. WRITE OPERATION

If the entire memory is loaded with new information at once prior to a search operation then the writing can be done by addressing each location in sequence. This will make the device a random-access memory for writing and a content addressable memory for reading. The advantage here is that the address for input can be decoded as in a random-access memory. Thus instead of having m address lines, one for each word in memory, the number of address lines can be reduced by the decoder to d lines, where m = 2^d.

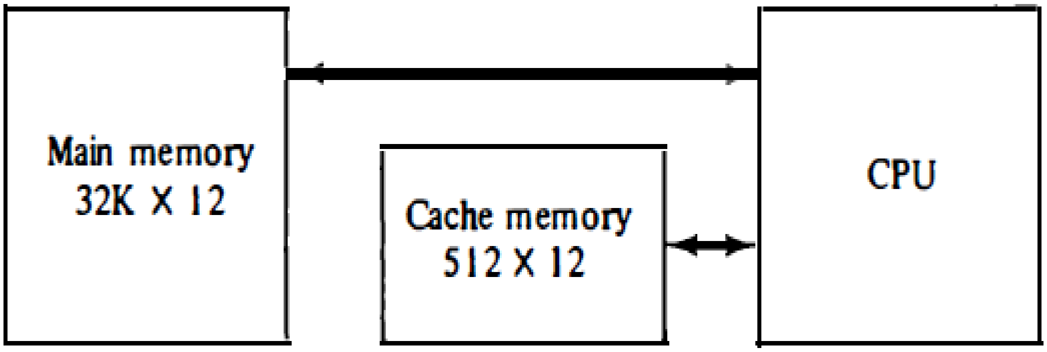
If unwanted words have to be deleted and new words inserted one at a time, there is a need for a special register to distinguish between active and inactive words. This register, sometimes called a tag register, would have as many bits as there are words in the memory. For every active word stored in memory, the corresponding bit in the tag register is set to 1. A word is deleted from memory by clearing its tag bit to 0. Words are stored in memory by scanning the tag register until the first 0 bit is encountered. This gives the first available inactive word and a position for writing a new word. After the new word is stored in memory it is made active by setting its tag bit to 1. An unwanted word when deleted from memory can be cleared to all 0's if this value is used to specify an empty location.

4.5. CACHE MEMORY

Analysis of a large number of typical programs has shown that the references, to memory at any given interval of time tend to be confined within a few localized areas in memory. The phenomenon is known as the **property of locality of reference.**

The locality of reference property, which states that over a short interval of time, the addresses generated by a typical program refer to a few localized areas of memory repeatedly, while the remainder of memory is accessed relatively frequently.

If the active portions of the program and data are placed in a fast small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. Such a fast small memory is referred to as **a cache memory.** It is placed between the CPU and main memory as illustrated in figure.



The cache memory access time is less than the access time of main memory by a factor of 5 to 10. The cache is the fastest component in the memory hierarchy and approaches the speed of CPU components.

The fundamental idea of cache organization is that by keeping the most frequently accessed instructions and data in the fast cache memory.

The basic operation of the cache is as follows. When the CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast memory. If the word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. A block of words containing the one just accessed is then transferred from main memory to cache memory.

The block size may vary from one word (the one just accessed) to about 16 words adjacent to the one just accessed. In this manner, some data are transferred to cache so that future references to memory find the required words in the fast cache memory.

The performance of cache memory is frequently measured in terms of a quantity **called hit ratio.** When the CPU refers to memory and finds the word in cache, it is said to produce **a hit.** If the word is not found in cache, it is in main memory and it counts as a **miss.** The ratio of the number of hits divided by the total CPU references to memory (hits plus misses) is the **hit ratio.**

For example, a computer with cache access time of 100 ns, a main memory access time of 1000 ns, and a hit ratio of 0.9 produces an average access time of 200 ns. This is a considerable improvement over a similar computer without a cache memory, whose access time is 1000 ns.

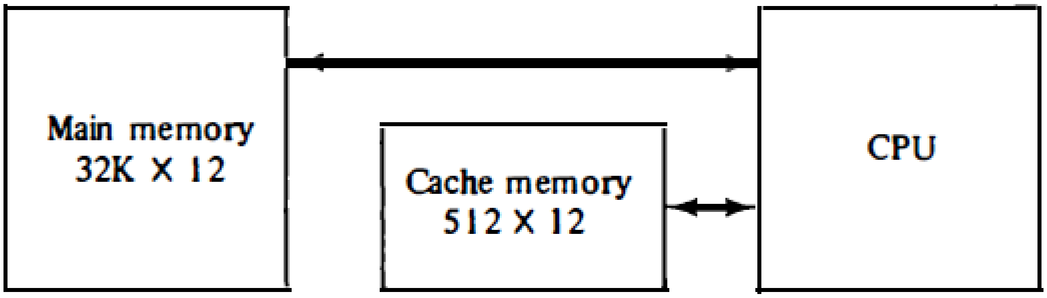
The basic characteristic of cache memory is its fast access time. Therefore, very little or no time must be wasted when searching for words in the cache.

The transformation of data from main memory to cache memory is referred to as a **mapping process.**

Three types of mapping procedures are of practical interest when considering the organization of cache memory:

1. Associative mapping
2. Direct mapping
3. Set-associative mapping

To helping the discussion of these three mapping procedures we will use a specific example of a memory organization as shown in figure.

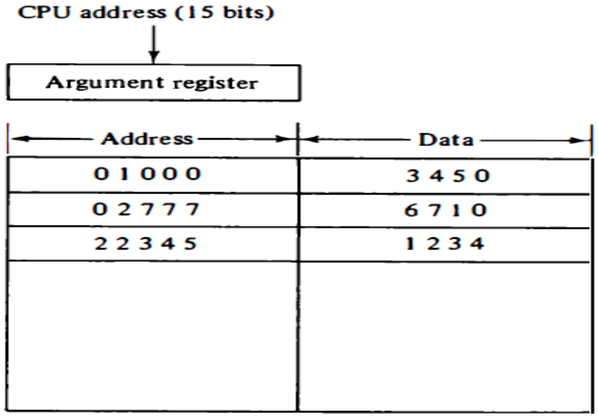


The main memory can store 32K words of 12 bits each. The cache is capable of storing 512 of these words at any given time. For every word stored in cache, there is a duplicate copy in main memory.

The CPU communicates with both memories. It first sends a 15-bit address to cache. If there is a hit, the CPU accepts the 12 -bit data from cache. If there is a miss, the CPU reads the word from main memory and the word is then transferred to cache.

4.5.1 ASSOCIATIVE MAPPING

The fasters and most flexible cache organization uses an associative memory. This organization is illustrated in Fig.

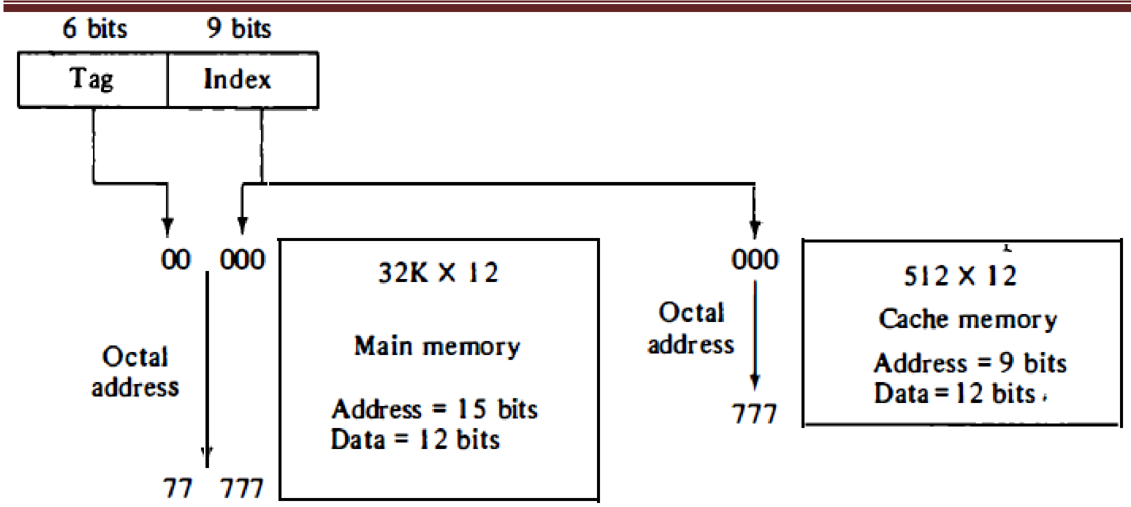


Address	Data
01000	3450
02777	6710
22345	1234

The associative memory stores both the address and content (data) of the memory word. This permits any location in cache to store any word from main memory. The diagram shows three words presently stored in the cache. The address value of 15 bits is shown as a five-digit octal number and its corresponding 12-bit word is shown as a four-digit octal number. A CPU address of 15 bits is placed in the argument register and the associative memory is searched for a matching address. If the address is found, the corresponding 12-bit data is read and sent to the CPU. If no match occurs, the main memory is accessed for the word. This constitutes a first-in first-out (FIFO) replacement policy.

4.5.2. DIRECT MAPPING

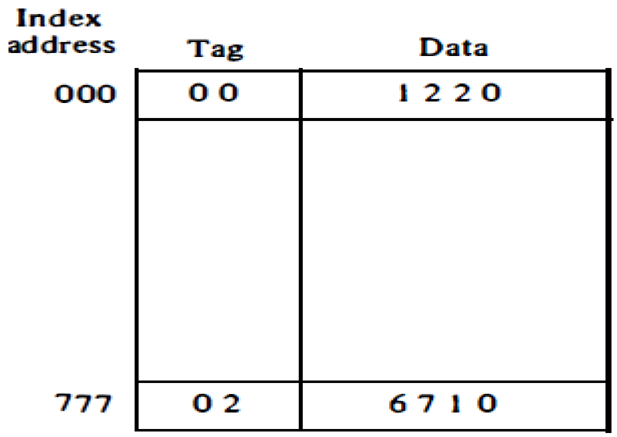
Associative memories are expensive compared to random-access memories because of the added logic associated with each cell. The possibility of using a random-access memory for the cache is investigated in Fig.



The CPU address of 15 bits is divided into two fields. The nine least significant bits constitute the index field and the remaining six bits form the tag and the index bits. The number of bits in the index field is equal to the number of address bits required to access the cache memory.

In the general case, there are 2^k words in cache memory and 2^n words in main memory. The n-bit memory address is divided into two fields: k bits for the index field and n − k bits for the tag field. The direct mapping cache organization uses the n-bit address to access the main memory and the k-bit index to access the cache.

The internal organization of the words in the cache memory is as shown in Fig.

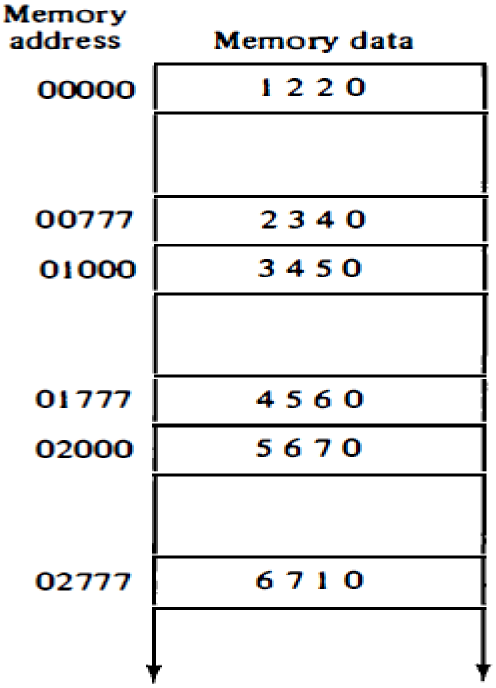


Index address	Tag	Data
000	0 0	1 2 2 0
777	0 2	6 7 1 0

Each word in cache consists of the data word and its associated tag. When a new word is first brought into the cache, the tag bits are stored alongside the data bits. When the CPU generates a memory request, the index field is used for the address to access the cache.

The tag field of the CPU address is compared with the tag in the word read from the cache. If the two tags match, there is a hit and the desired data word is in cache. If the two tags match, there is a hit and the desired data word is in cache. If there is no match, there is a miss and the required word is read from main memory.

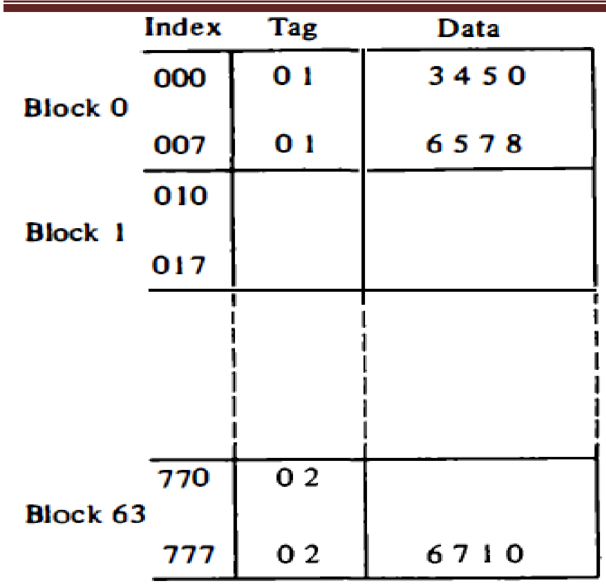
To see how the direct-mapping organization operates, consider the numerical example shown in Fig below.



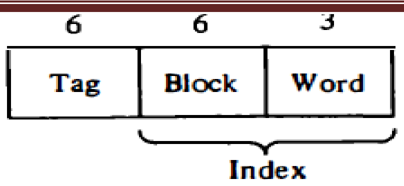
Memory address	Memory data
00000	1 2 2 0
00777	2 3 4 0
01000	3 4 5 0
01777	4 5 6 0
02000	5 6 7 0
02777	6 7 1 0

The word at address zero is presently stored in the cache (index = 000, tag = 00, data = 1220). Suppose that the CPU now wants to access the word at address 02000. The index address is 000, so it is sued to access the cache. The two tags are then compared. The cache tag is 00 but the address tag is 02, which does not produce a match. Therefore, the main memory is accessed and the data word 5670 is transferred to the CPU. The cache word at index address 000 is then replaced with a tag of 02 and data of 5670.

The same organization but using a block size of 8 words is shown in Fig.



	Index	Tag	Data
Block 0	000	0 1	3 4 5 0
	007	0 1	6 5 7 8
Block 1	010		
	017		
Block 63	770	0 2	
	777	0 2	6 7 1 0



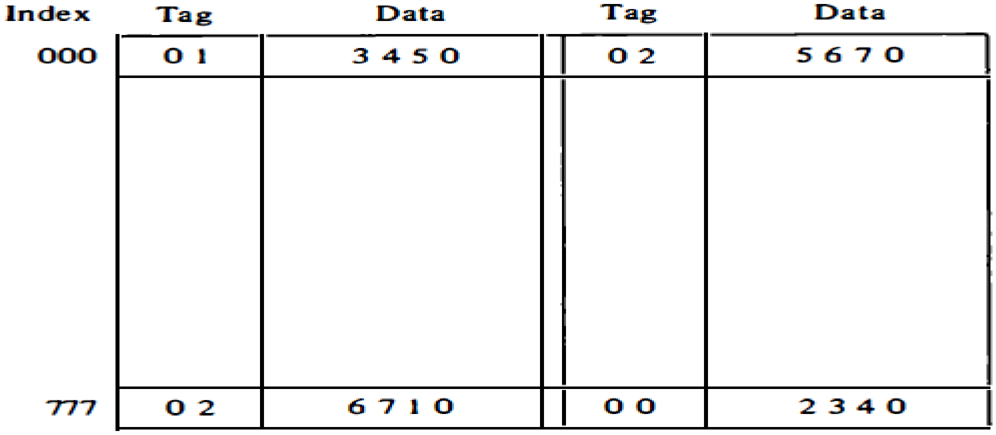
The index field is now divided into two parts: **the block field and the word field**. In a 512-word cache there are 64 block of 8 words each, since 64X8 = 512. The block number is specified with a 6-bit field and the word within the block is specified with a 3-bit field.

The tag field stored within the cache is common to all eight words of the same block. Every time a miss occurs, an entire block of eight words must be transferred from main memory to cache memory. Although this takes extra time, the hit ratio will most likely improve with a larger block size because of the sequential nature of computer programs.

4.5.3. SET-ASSOCIATIVE MAPPING

It was mentioned previously that the disadvantage of direct mapping is that two words with the same index in their address but with different tag values cannot reside in cache memory at the same time. A third type of cache organization, called **set-associative mapping**, is an improvement over the direct-mapping organization in that each word of cache can store two or more words of memory under the same index address. Each data word is stored together with its tag and the number of tag-data items in one word of cache is said to form a set.

An example of a set-associative cache organization for a set size of two is shown in Fig.



Index	Tag	Data	Tag	Data
000	01	3450	02	5670
777	02	6710	00	2340

Each index address refers to two data words and their associated tags. Each tag requires six bits and each data word has 12 bits, so the word length is 2(6 + 12) = 36 bits. An index address of nine bits can accommodate 512 words. Thus the size of cache memory is 512 X 36.

It can accommodate 1024 words of main memory since each word of cache contains two data words. In general, a set-associative cache of set size k will accommodate k words of main memory in each word of cache.

With reference to the main memory content the following is illustrated. The words stored at addresses 01000 and 02000 of main memory are stored in cache memory at index address 000. Similarly, the words at addresses 02777 and 00777 are stored in cache at index address 777.

When a miss occurs in a set-associative cache and the set is full, it is necessary to replace one of the tag-data items with a new value. The most common replacement algorithms used are: random replacement, first-in, first out (FIFO), and least recently used (LRU). With the random replacement policy the control chooses one tag-data item for replacement at random. The FIFO procedure selects for replacement the item that has been in the set the longest. The LRU algorithm selects for replacement the item that has been least recently used by the CPU. Both FIFO and LRU can be implemented by adding a few extra bits in each word of cache.

WRITING INTO CACHE

For write operation, there are two ways that the system can proceed. The simplest and most commonly used procedure is to up data main memory with every memory write operation, with cache memory being updated in parallel if it contains the word at the specified address. This is called the **write-through method.**

The second procedure is called the **write-back method.** In this method only the cache location is updated during a write operation. The location is then marked by a flag so that later when the words are removed from the cache it is copied into main memory.

CACHE INITIALIZATION

One more aspect of cache organization that must be taken into consideration is the problem of initialization. The cache is initialized when power is applied to the computer or when the main memory is loaded with a complete set of programs from auxiliary memory. After initialization the cache is considered to be empty, built in effect it contains some non-valid data. It is customary to include with each word in cache a valid bit to indicate whether or not the word contains valid data. The cache is initialized by clearing all the valid bits to 0. The valid bit of a particular cache word is set to 1 the first time this word is loaded from main memory and stays set unless the cache has to be initialized again.

4.6. VIRTUAL MEMORY

In a memory hierarchy system, programs and data are brought into main memory as they are needed by the CPU. Virtual memory is a concept used in some large computer systems that permit the user to construct programs as though a large memory space were available, equal to the totality of auxiliary memory. Each address that is referenced by the CPU goes through an address mapping from the so-called virtual address to a physical address in main memory.

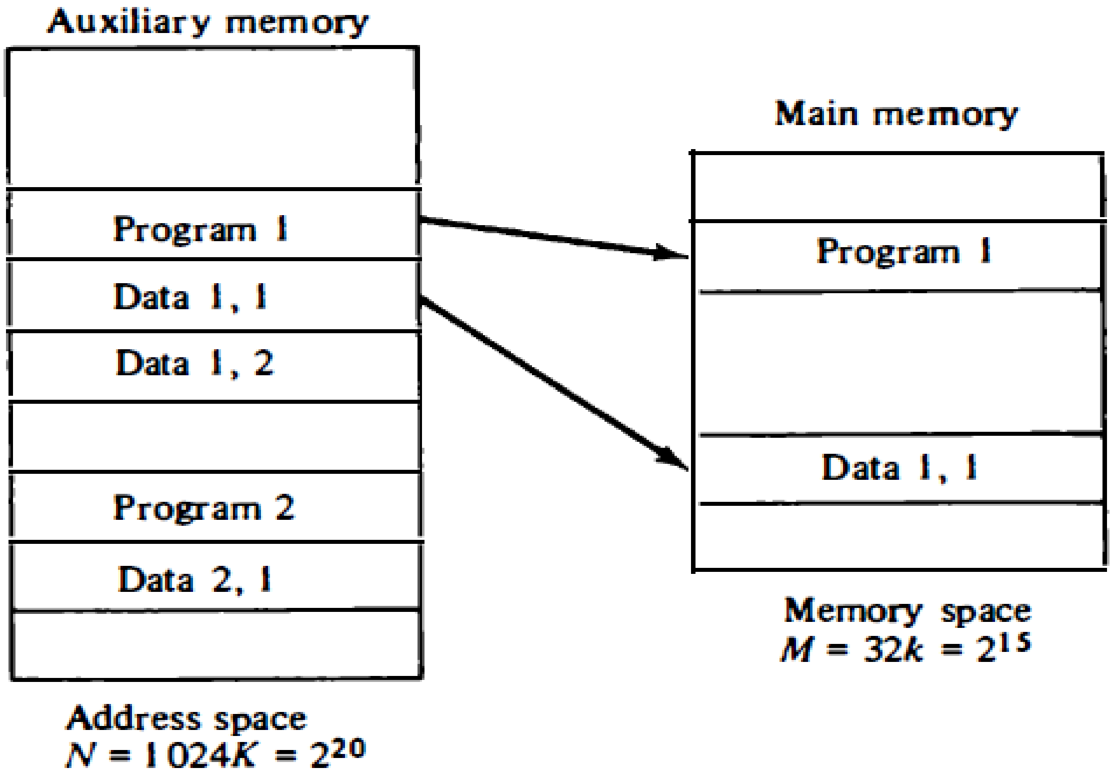
Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory. A virtual memory system provides a mechanism for translating program-generated addresses into correct main memory locations. This is done dynamically, while programs are being executed in the CPU. The translation or mapping is handled automatically by the hardware by means of a **mapping table.**

4.6.1. ADDRESS SPACE AND MEMORY SPACE

An address used by a programmer will be called a **virtual address,** and the set of such addresses the address space. An address in main memory is called a **location or physical address.** The set of such locations is called the **memory space.** Thus the address space is the set of addresses generated by programs as they reference instructions and data; the memory space consists of the actual main memory locations directly addressable for processing. In most computers the address and memory spaces are identical.

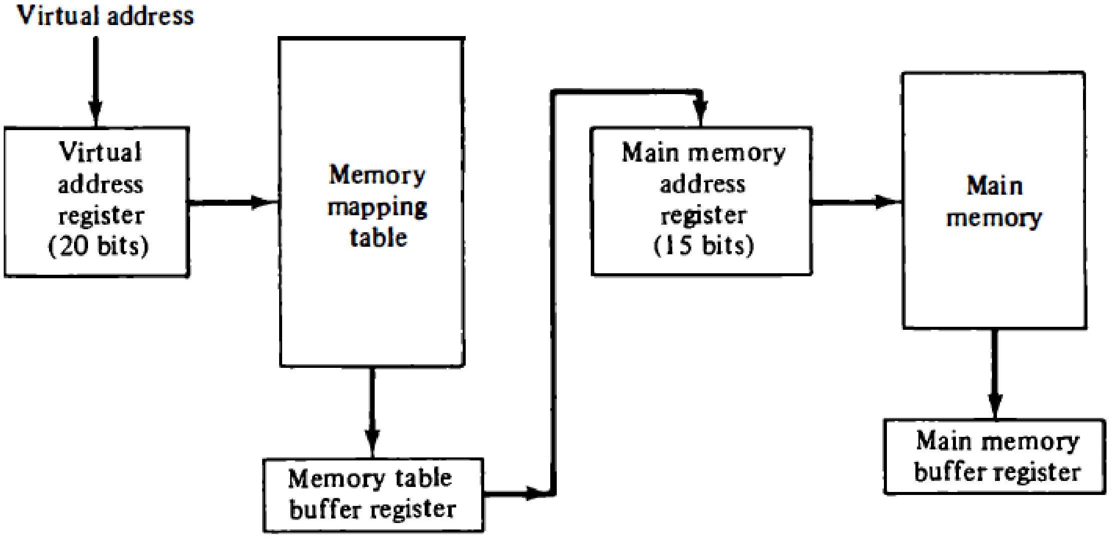
As an illustration, consider a computer with a main -memory capacity of 32K words (K = 1024). Fifteen bits are needed to specify a physical address in memory since 32K = 2^{15}. Suppose that the computer has available auxiliary memory for storing 2^{20} = 1024K words. Thus auxiliary memory has a capacity for storing information equivalent to the capacity of 32 main memories. Denoting the address space by N and the memory space by M, we then have for this example N = 1024K and M = 32K.

In a multiprogram computer system, programs and data are transferred to and from auxiliary memory and main memory based on demands imposed by the CPU. Suppose that program 1 is currently being executed in the CPU. Program 1 and a portion of its associated data re moved from auxiliary memory into main memory as shown in figure



Portions of programs and data need not be in contiguous locations in memory since information is being moved in and out, and empty spaces may be available in scattered locations in memory.

The mapping table may be stored in a separate memory as shown in Fig.


Virtual address
Virtual address register (20 bits)
Memory mapping table
Memory table buffer register
Main memory address register (15 bits)
Main memory
Main memory buffer register

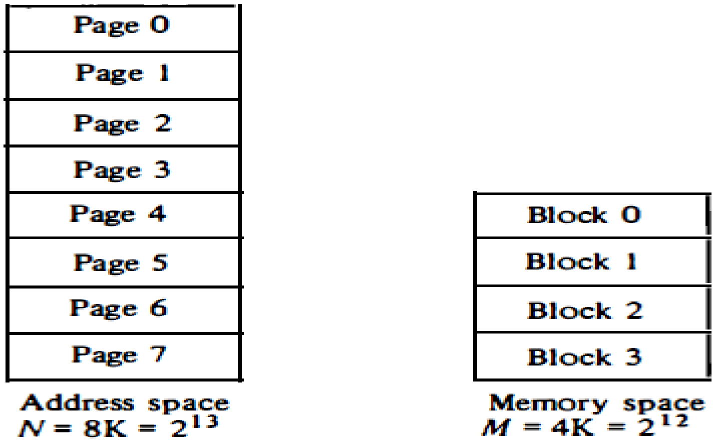
or in main memory. In the first case, an additional memory unit is required as well as one extra memory access time. In the second case, the table takes space from main memory and two accesses to memory are required with the program running at half speed. A third alternative is to use an associative memory as explained below.

4.6.2. ADDRESS MAPPING USING PAGES

The table implementation of the address mapping is simplified if the information in the address space and the memory space are each divided into groups of fixed size. The physical memory is broken down into groups of equal size called blocks, which may range from 64 to 4096 words each. The term page refers to groups of address space of the same size. For example, if a page or block consists of 1K words, then, using the previous example, address space is divided into 1024 pages and main memory is divided into 32 blocks. Although both a page and a block are split into groups of 1K words, a page refers to the organization of address space, while a block refers to the organization of memory space.

The programs are also considered to be split into pages. Portions of programs are moved from auxiliary memory to main memory in records equal to the size of a page. The term “page frame” is sometimes used to denote a block.

Consider a computer with an address space of 8K and a memory space of 4K. If we split each into groups of 1K words we obtain eight pages and four blocks as shown in below Fig.



At any given time, up to four pages of address space may reside in main memory in any one of the four blocks.

The mapping from address space to memory space is facilitated if each virtual address is considered to be represented by two numbers: a page number address and a line within the page. In a computer with 2^p words per page, p bits are used to specify a line address and the remaining high-order bits of the virtual address specify the page number. In the example figure below.

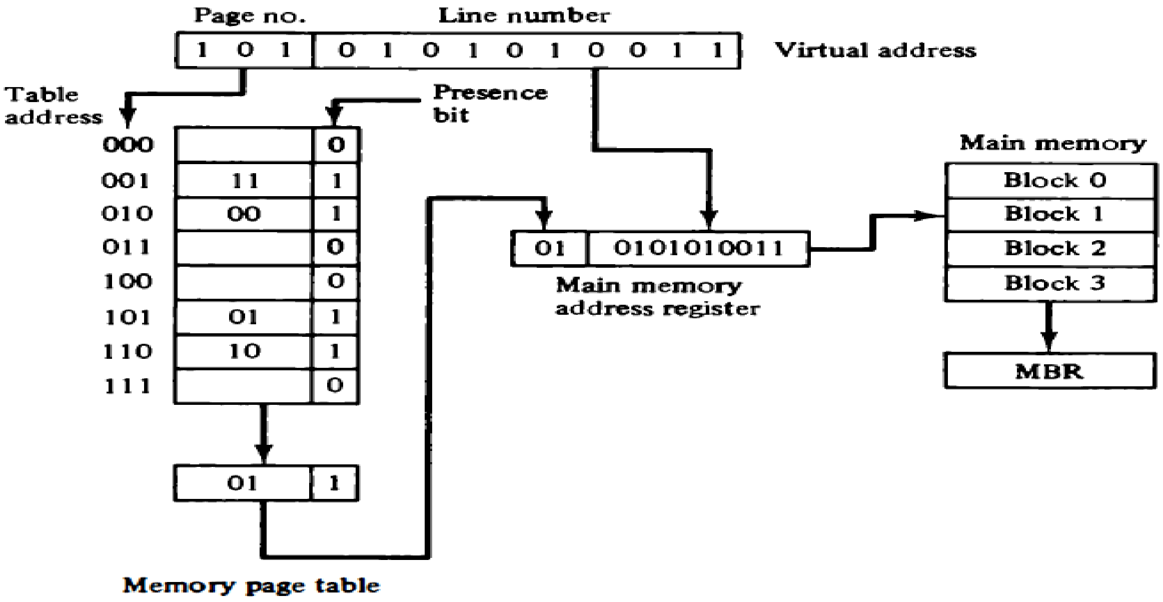


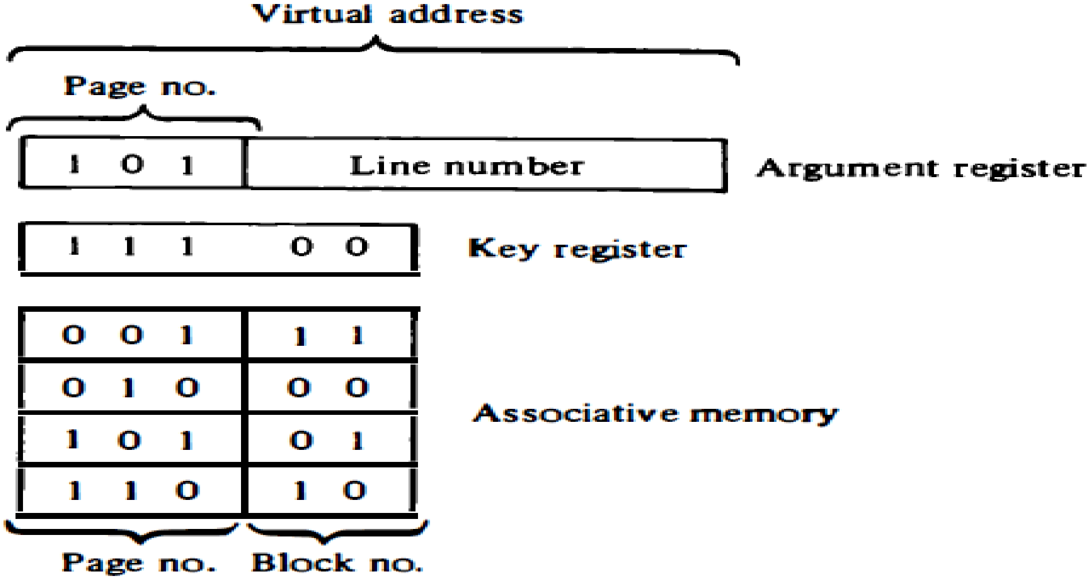
Table address		Presence bit
000		0
001	11	1
010	00	1
011		0
100		0
101	01	1
110	10	1
111		0

A virtual address has 13 bits. Since each page consists of 2^{10} = 1024 words, the high-order three bits of a virtual address will specify one of the eight pages and the low-order 10 bits give the line address within the page. Note that the line address in address space and memory space is the same; the only mapping required is from a page number to a block number.

4.6.3. **ASSOCIATIVE MEMORY PAGE TABLE**

A efficient way to organize the page table would be to construct it with a number of words equal to the number of blocks in main memory. In this way the size of the memory is reduced and each location is fully utilized. This method can be implemented by means of an associative memory with each word in memory containing a page number together with its corresponding block number. The page field in each word is compared with the page number in the virtual address. If a match occurs, the word is read from memory and its corresponding block number is extracted.

Consider again the case of eight pages and four blocks as in the previous example of Fig. We replace the random access memory-page table with an associative memory of four words as shown in following Fig.



Each entry in the associative memory array consists of two fields. The first three bits specify a field fro storing the page number. The last two bits constitute a field for storing the block number. The virtual address is placed in the argument register. The page number bits in the argument are compared with all page numbers in the page field of the associative memory. If the page number is found, the 5-bit word is read out from memory. The corresponding block number, being in the same word, is transferred to the main memory address register. If no match occurs, a call to the operating system is generated to bring the required page from auxiliary memory.

4.6.4. PAGE REPLACEMENT

A virtual memory system is a combination of hardware and software techniques. The memory management software system handles all the software operations for the efficient utilization of memory space. It must decide

(1) Which page in main memory ought to be removed to make room for a new page,
(2) When a new page is to be transferred from auxiliary memory to main memory, and
(3) Where the page is to be placed in main memory.

The hardware mapping mechanism and the memory management software together constitute the architecture of a virtual memory.

The program is executed from main memory until it attempts to reference a page that is still in auxiliary memory. This condition is called **page fault.** When page fault occurs, the execution of the present program is suspended until the required page is brought into main memory. Since loading a page from auxiliary memory to main memory is basically an I/O operation, the operating system assigns this task to the I/O processor. In the meantime, controls transferred to the next program in memory that is waiting to be processed in the CPU. Later, when the memory block has been assigned and the transfer completed, the original program can resume its operation.

The policy for choosing pages to remove is determined from the replacement algorithm that is used. The goal of a replacement policy is to try to remove the page least likely to be referenced in the immediate future.

Two of the most common replacement algorithms used are the first-in first-out (FIFO) and the least recently used (LRU).

The FIFO algorithm selects for replacement the page the has been in memory the longest time. Each time a page is loaded into memory, its identification number is pushed into a FIFO stack. FIFO will be full whenever memory has no more empty blocks.

The FIFO replacement policy has the advantage of being easy to implement.

It has the disadvantage that under certain circum-stances pages are removed and loaded form memory too frequently.

The LRU algorithm can be implemented by associating a counter with every page that is in main memory. When a page is referenced, its associated counter is set to zero. At fixed intervals of time, the counters associated with all pages presently in memory are incremented by 1. The least recently used page is the page with the highest count. The counters are often called aging registers, as their count indicates their age, that is, how long ago their associated pages have been referenced.

Unit-V

Pipeline and Multiprocessor

5.1. Parallel Processing

Parallel processing is a term used to denote a large class of techniques that are used to provide simultaneous data processing tasks for the purpose of increasing the computational speed of computer system.

Instead of processing each instruction sequentially as in a conventional computer, a parallel processing system is able to perform concurrent data processing to achieve faster execution time.

For example, while an instruction is being executed in the ALU, the next instruction can be read from memory. The system may have two or more processors operating concurrently.

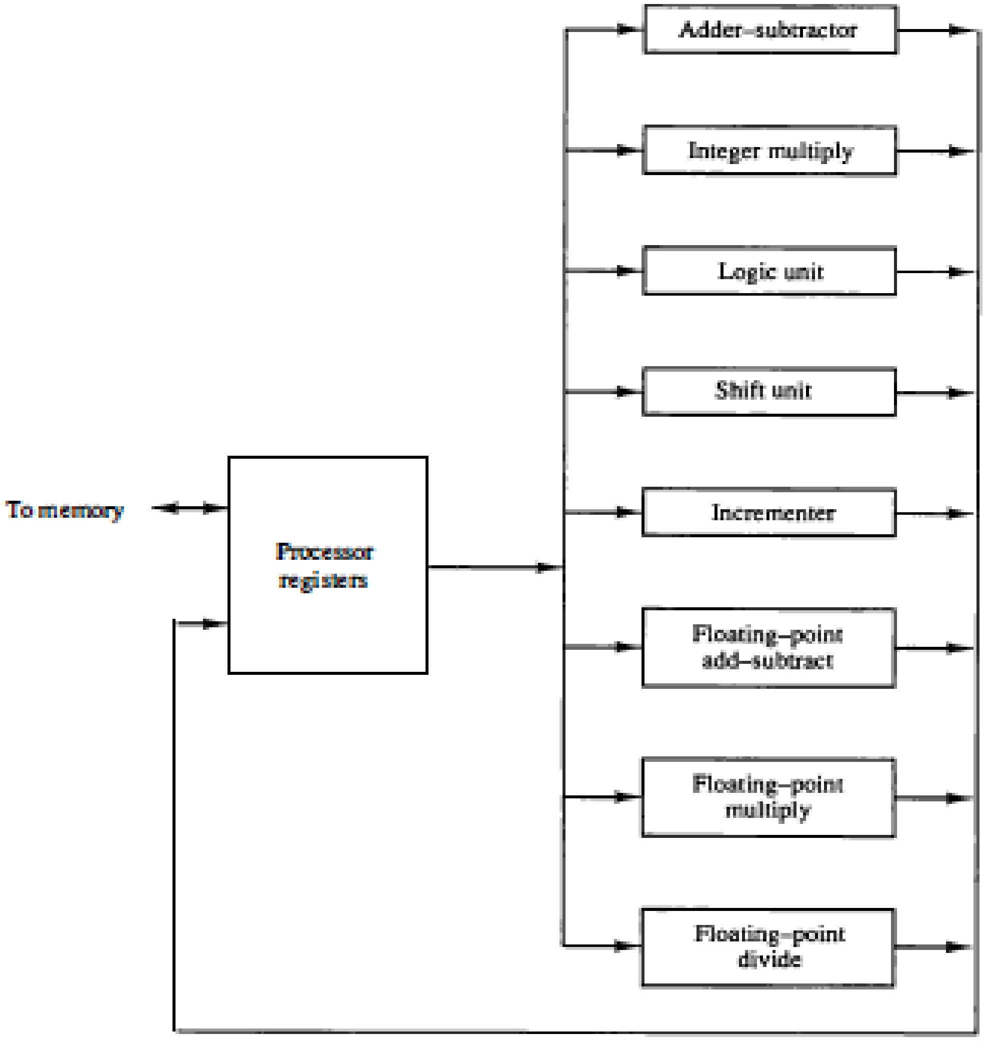
ADVANTAGES OF PARALLEL PROCESSING:

1. It speeds up the computer processing capability.
2. Increases its throughput, i.e., the amount of processing that can be accomplished during a given interval of time.
3. The amount of hardware increases with parallel processing and with it the cost of the system increases. However, technological developments have reduced hardware costs to the point where parallel processing techniques are economically feasible.

Parallel processing can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously. Parallel processing is established by distributing the data among the multiple functional units.
Fig shows one possible way of separating the execution time into 8 functional units operating in parallel. The operands in the registers are applied to one of the units depending on the operation specified by the instruction associated with the operands.

The operation performed in each functional unit is indicated in each block of the diagram. The adder and integer multiplier perform the arithmetic operations with integer numbers. The floating point operations are separated into three circuits operating in parallel.

The logic, shift and increment operations can be performed concurrently on different data. All units are independent of each other, so one number can be shifted while another number is being incremented.



Flynn’s classification divides computers into four major groups

Single instruction stream, single data stream (SISD)

Single instruction stream, multiple data stream (SIMD)

Multiple instruction streams, single data stream (MISD)

Multiple instruction stream, multiple data stream (MIMD)

SISD:

SISD represents the organization of a single computer containing a control unit, a processor unit and a memory unit. Instructions are executed sequentially and the system may or may not have internal parallel processing capabilities. Parallel processing in this case may be achieved by means of multiple functional units or by pipeline processing.

SIMD:

SIMD represents an organization that includes many processing units under the supervision of a common control unit. All processors receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously.

5.2. Pipelining

Pipelining is a technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments.

A pipeline can be visualized as a collection of processing segments through which binary information flows. Each segment performs partial processing dictated by the way the task is partitioned.

The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments.

5.2.1. Pipeline Organization

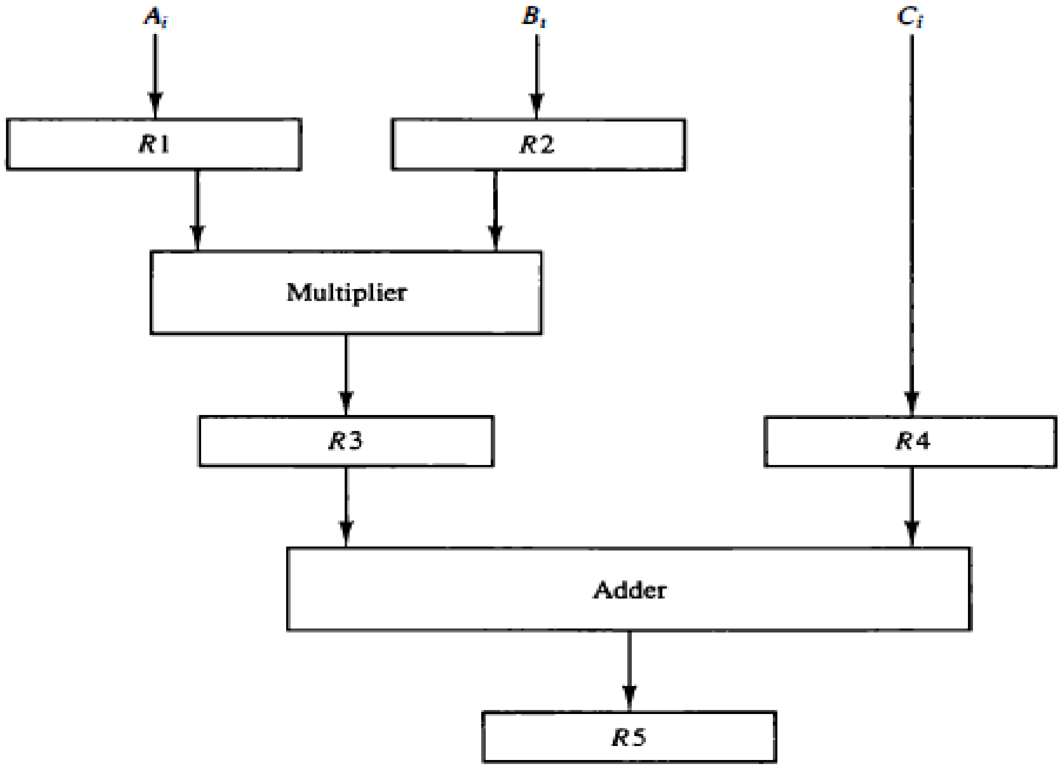
The simplest way of viewing the pipeline structure is to imagine that each segment consists of an input register followed by a combinational circuit. The register holds the data and the combinational circuit performs the sub operation in the particular segment. The output of the combinational circuit is applied to the input register of the next segment. A clock is applied to all registers after enough time has elapsed to perform all segment activity. In this way the information flows through the pipeline one step at a time.

Example demonstrating the pipeline organization

Suppose we want to perform the combined multiply and add operations with a stream of numbers.

$A_i * B_i + C_i$ for i=1, 2, 3 ….7

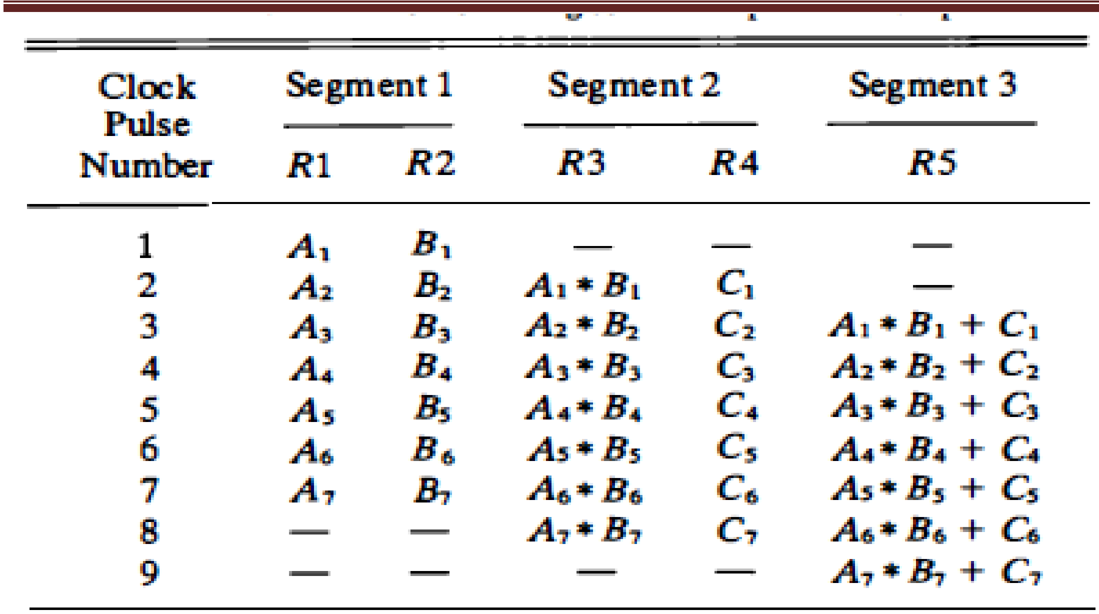
Each sub operation is to implemented in a segment within a pipeline. Each segment has one or two registers and a combinational circuit as shown in fig.



R1 through r5 are registers that receive new data with every clock pulse. The multiplier and adder are combinational circuits. The sub operations performed in each segment of the pipeline are as follows:

R1<- A_i	R2<-B_i	Input A_i and B_i
R3<-R1*R2	R4<-C_i	multiply and input C_i
R5<-R3+R4		add C_i to product

The five registers are loaded with new data every clock pulse.

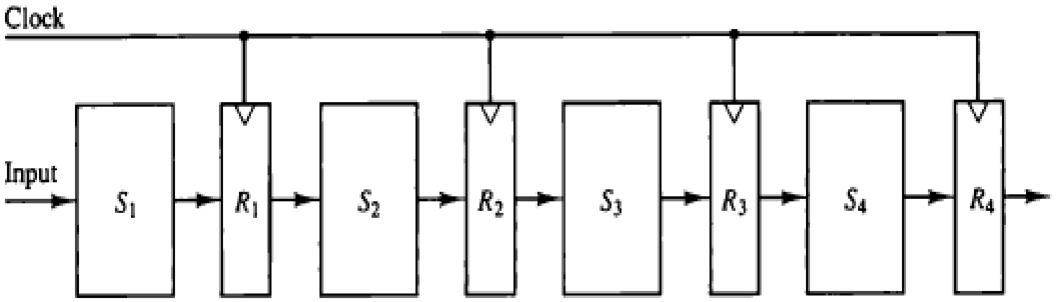


Clock Pulse Number	Segment 1		Segment 2		Segment 3
	$R1$	$R2$	$R3$	$R4$	$R5$
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

The first clock pulse transfers A1 and B1 into R1 and R2. The second clock pulse transfers the product of R1 and R2 into R3 and C1 into R4. The same clock pulse transfers A2 and B2 into R1 and R2. The third clock pulse operates on all three segments simultaneously. It places A3 and B3 into R1 and R2, transfers the product of R1 and R2 into R3, transfers C2 into R4, and places the sum of R3 and R4 into R5. It takes three clock pulses to fill up the pipe and retrieve the first output from R5. From there on, each clock produces a new output and moves the data one step down the pipeline. This happens as long as new input data flow into the system.

<u>5.2.2.FOUR SEGMENT Pipeline</u>

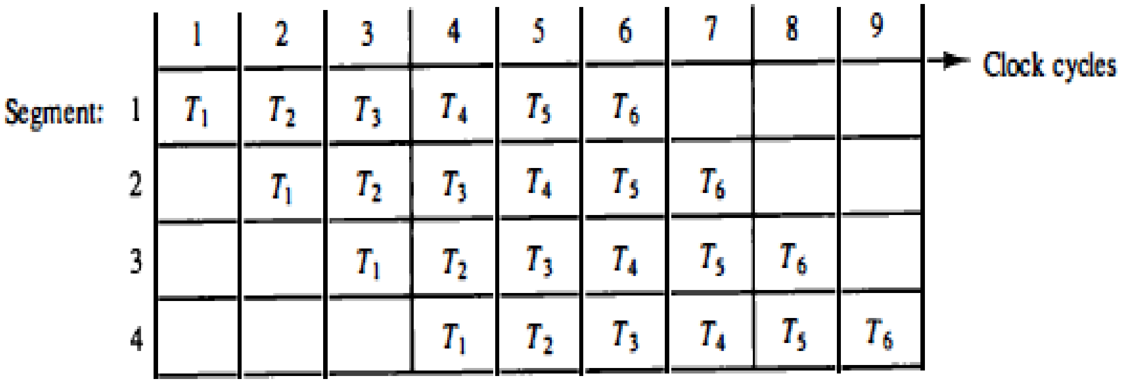
The general structure of four segment pipeline is shown in fig. the operands are passed through all four segments in affixed sequence. Each segment consists of a combinational circuit Si that performs a sub operation over the data stream flowing through the pipe. The segments are separated by registers Ri that hold the intermediate results between the stages. Information flows between adjacent stages under the control of a common clock applied to all the registers simultaneously.



SPACE TIME DIAGRAM:

The behavior of a pipeline can be illustrated with a space time diagram. This is a diagram that shows the segment utilization as a function of time.

Fig The horizontal axis displays the time in clock cycles and the vertical axis gives the segment number. The diagram shows six tasks T1 through T6 executed in four segments. Initially, task T1 is handled by segment 1. After the first clock,



	1	2	3	4	5	6	7	8	9	→ Clock cycles
Segment: 1	T_1	T_2	T_3	T_4	T_5	T_6				
2		T_1	T_2	T_3	T_4	T_5	T_6			
3			T_1	T_2	T_3	T_4	T_5	T_6		
4				T_1	T_2	T_3	T_4	T_5	T_6	

segment 2 is busy with T1, while segment 1 is busy with task T2. Continuing in this manner, the first task T1 is completed after fourth clock cycle. From then on, the pipe completes a task every clock cycle.

Consider the case where a k-segment pipeline with a clock cycle time tp is used to execute n tasks. The first task T1 requires a time equal to ktp to complete its operation since there are k segments in a pipe. The remaining n-1 tasks emerge from the pipe at the rate of one task per clock cycle and they will be completed after a time equal to (n-1) tp. Therefore, to complete n tasks using a k segment pipeline requires

k+ (n-1) clock cycles.

Consider a non pipeline unit that performs the same operation and takes a time equal to tn to complete each task. The total time required for n tasks is n t_n. The speedup of a pipeline processing over an equivalent non pipeline processing is defined by the ratio

$$\mathbf{S=nt_n / (k+n-1)t_p}$$

As the number of tasks increases, n becomes much larger than k-1, and k+n-1 approaches the value of n. under this condition the speed up ratio becomes

$$\mathbf{S=t_n/t_p}$$

If we assume that the time it takes to process a task is the same in the pipeline and non pipeline circuits, we will have $t_n=kt_p$. Including this assumption speed up ratio reduces to

$$\mathbf{S=kt_p/tp=k}$$

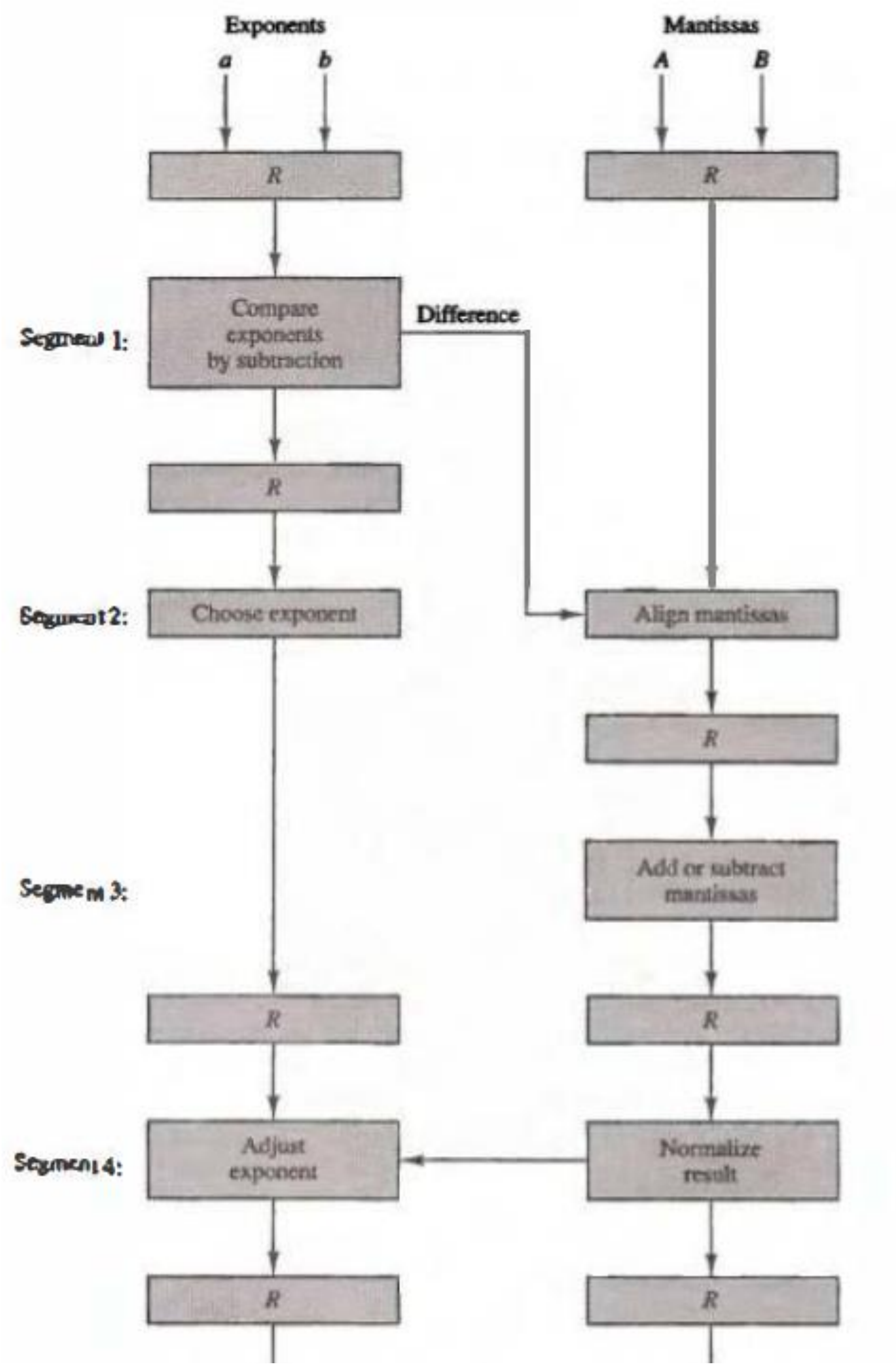
5.3. ARITHMETIC PIPELINE

An arithmetic pipeline divides an arithmetic operation into sub operations for execution in the pipeline segments. Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating point operations, multiplication of fixed point numbers, and similar computations encountered in scientific problems.

5.3.1. Pipeline Unit For Floating Point Addition And Subtraction:

The inputs to the floating point adder pipeline are two normalized floating point binary numbers.

$$X=A*2^a$$
$$Y=B*2^b$$



A and B are two fractions that represent the mantissa and a and bare the exponents. The floating point addition and subtraction can be performed in four segments. The registers labeled are placed between the segments to store intermediate results. The sub operations that are performed in the four segments are:

1. Compare the exponents
2. Align the mantissa.
3. Add or subtract the mantissas.
4. Normalize the result.

The exponents are compared by subtracting them to determine their difference. The larger exponent is chosen as the exponent of the result. The exponent difference determines how many times the mantissa associated with the smaller exponent must be shifted to the right. This produces an alignment of the two mantissas.

The two mantissas are added or subtracted in segment3. The result is normalized in segment 4. When an overflow occurs, the mantissa of the sum or difference is shifted to right and the exponent incremented by one. If the underflow occurs, the number of leading zeroes in the mantissa determines the number of left shits in the mantissa and the number that must be subtracted from the exponent.

5.4. INSTRUCTION PIPELINE

An instruction pipeline operates on a stream of instructions by overlapping the fetch, decode, and execute phases of instruction cycle. An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instruction fetch and executes phases to overlap and perform simultaneous operations.

Consider a computer with an instruction fetch unit and an instruction execute unit designed to provide a two segment pipeline. The instruction fetch segment can be implemented by means of a first in first out (FIFO) buffer. Whenever the execution unit is not using memory, the control increments the program counter and uses it address value to read consecutive instructions from memory. The instructions are inserted into the FIFO buffer so that they can be executed on a first in first out basis. Thus an instruction stream can be placed in queue, waiting for decoding and processing by the execution segment.

In general the computer needs to process each instruction with the following sequence of steps.

1. Fetch the instruction.
2. Decode the instruction.
3. Calculate the effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in the proper place.

5.4.1. Four Segment Instruction Pipeline

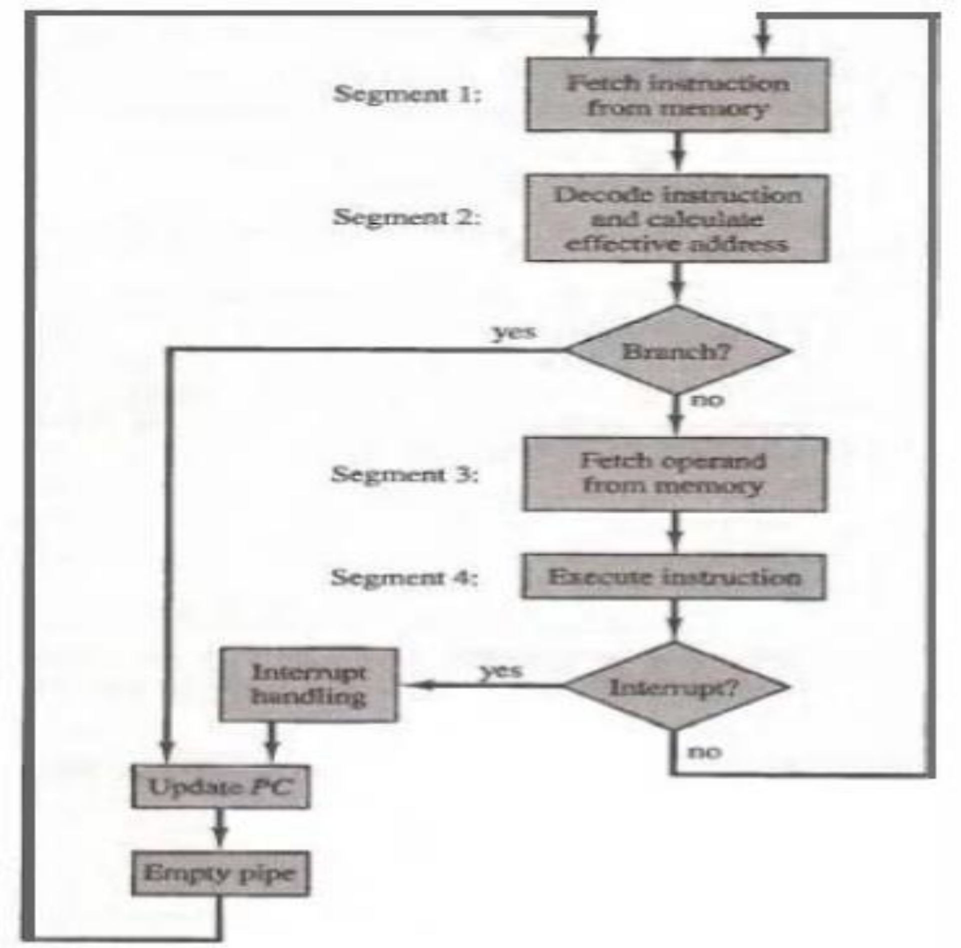
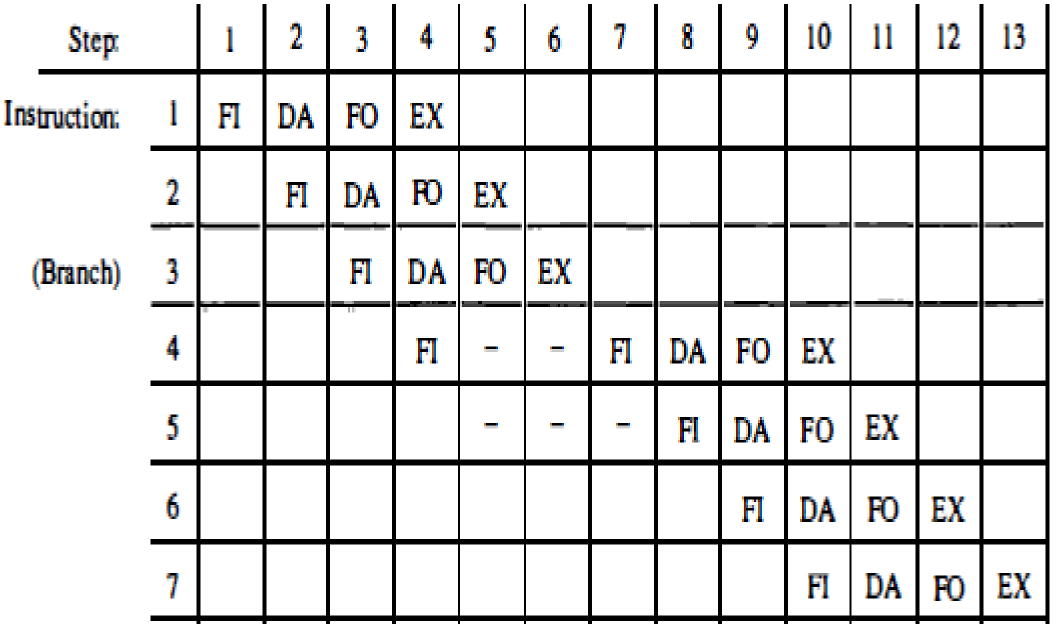


Fig shows the instruction cycle in the CPU can be processed with a four segment pipeline. While an instruction is being executed in segment 4, the next instruction in sequence is busy with fetching an operand from memory in segment 3. the effective address may be calculated in a separate arithmetic circuit for the third instruction, and whenever the memory is available, the fourth and all subsequent instructions are placed in an instruction FIFO.

Fig shows the operation of the instruction pipeline. The time in the horizontal axis is divided into steps of equal duration. The four segments are represented in the diagram with an abbreviated symbol.



Step:		1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction:	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	-	-	FI	DA	FO	EX			
	5					-	-	-	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

1. FI is the segment that fetches an instruction.
2. DA is the segment that decodes the instruction and calculates the effective address.
3. FO is the segment that fetches the operand.
4. EX is the segment that executes the instruction.

It is assumed that the processor has separate instruction and data memories so that the operation in FI and FO can proceed at the same time. In the absence of a branch instruction, each segment operates on different instructions. Thus, in step 4, instruction 1 is being executed in segment EX; the operand for instruction 2 is being fetched into segment FO; instruction 3 is being decoded in segment DA; and instruction 4 is being fetched from memory in segment FI.

Assume now this instruction is a branch instruction. As soon as this instruction is decoded in segment DA in step 4, the transfer from FI to DA of the other instructions are halted until the branch instruction is executed in step 6.

PIPELINE CONFLICTS:

1. RESOURCE CONFLICTS: They are caused by access to memory by two segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories.

2. DATA DEPENDENCY: these conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.

3. BRANCH DIFFERENCE: they arise from branch and other instructions that change the value of PC.

5.4.2. DATA DEPENDENCY

A difficulty that may cause a degradation of performance in an instruction pipeline collision of data or address. A collision occurs when an instruction cannot proceed because previous instructions did not complete certain operations.

A data dependency occurs when an instruction needs data that are not yet available. For example an instruction in the FO segment may need to fetch an operand that is being generated at the same time by the previous instruction in segment EX. Therefore, the second instruction must wait for the data to become available by the first instruction.

An address dependency may occur when an operand address cannot be calculated because the information needed by the addressing mode is not available. For example, an instruction with register indirect mode can not proceed to fetch the operand if the previous instruction is loading the address into the register. Therefore operand access to memory must be delayed until the required address is available.

Pipelined computers deal with such conflicts between data dependencies in a variety of ways.

The most straight forward method is to insert HARDWARE INTERLOCKS. An interlock is a circuit that detects instructions whose source operands are destinations of instructions farther up in the pipeline.

Detection of this situation causes the instruction whose source is not available to be delayed by enough clock cycles to resolve the conflict.

Another technique called OPERRND FORWADING uses special hardware to detect a conflict and then avoid it by routing the data through special paths between pipeline segments.

A procedure employed in some computers is to give the responsibility for solving data conflicts problems to the compiler. The compiler for such computers are designed to detect a data conflict and reorder the instruction as necessary to delay the loading of the conflicting data by inserting no-operation instructions. This method is referred to as DELAYED LOAD.

5.4.3. Handling of Branch Instructions

One of the major problems in operating the instruction pipeline is the occurrence of the branch instructions. A branch instruction can be conditional or unconditional. An unconditional branch always alters the sequential program flow by loading the program counter with the target address. In a conditional branch, the control selects the target instruction if the condition is satisfied or the next sequential instruction if the condition is not satisfied.

Pipeline computers employ various hardware techniques to minimize the performance of degradation caused by instruction branching.

One way of handling a conditional branch is to pre fetch the target instruction in addition to the instruction following the branch. Both are saved until the branch is executed. If the branch condition is successful, the pipeline continues from the branch target instruction

Another possibility is the use of BRANCH TARGET BUFFER or BTB. The BTB is an associative memory included in the fetch segment of the pipeline. Each entry in the BTB consists of the address of the previously executed branch instruction and the target instruction for that branch. It also stores the next few instructions after the branch target instruction. When the pipeline decodes a branch instruction, it searches the associative memory BTB for the address of the instruction. If it is in the BTB, the instruction is available directly and pre fetch continues from the new path. If the instruction is not in the BTB the pipeline shifts to a new instruction stream and stores the target instruction in the BTB.

5.5. Characteristics of multiprocessors

A multiprocessor system is an interconnection of two or more CPUs with memory and input-output equipment. The term "processor" in multiprocessor can mean either a central processing unit (CPU) or an input-output processor (IOP). Multiprocessors are classified as multiple instruction stream, multiple data stream (MIMD) systems.

A multiprocessor system is controlled by one operating system that provides interaction between processors and all the components of the system. A multiprocessor improves the reliability of the system so that a failure or error in one part has a limited effect on the rest of the system. If a fault causes one processor to fail, a second processor can be assigned to perform the function of the disabled processor. This causes the loss in efficiency of the system but with no delay.

Multiprocessors can improve performance by decomposing the program into parallel executable tasks. This can be achieved in two ways.

*Multiple independent jobs can be made to operate in parallel.

*A single job can be partitioned into multiple parallel tasks.

Multiprocessors are classified by the way their memory is organized

1. Shared memory or tightly coupled multiprocessor:

A multiprocessor system with common shared memory is classified as shared memory. It provides a cache memory with each CPU. There will be a global common memory that all CPUs can access. Information can therefore be shared among the CPUs by placing it in the common global memory.

2. Disturbed memory or loosely coupled system:

Each processor element in a loosely coupled system has its own private local memory. The processors relay programs and data to other processors in packets, which consists of an address, the data content, and error detecting code.

5.6. Interconnection Structures

There are several physical forms available for establishing an interconnection network. Some of these schemes are

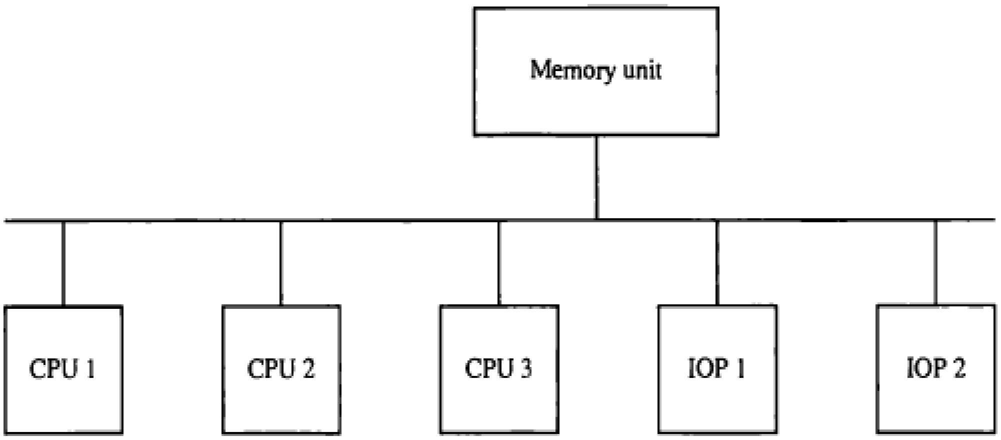
1. Time-shared common bus
2. Multiport memory
3. Crossbar switch
4. Multistage switching network
5. Hypercube system

1. Time-shared common bus:

A common bus multi processor system consists of a number of processors connected through a common path to a memory unit. Only one processor can communicate with the memory or another processor at any given time. Any processor wishing to initiate a transfer must first determine the availability of the bus. A command is issued to inform the destination unit what operation is to be performed. The receiving unit recognizes its address and responds to the control signals from the sender, after this the transfer will be initiated.

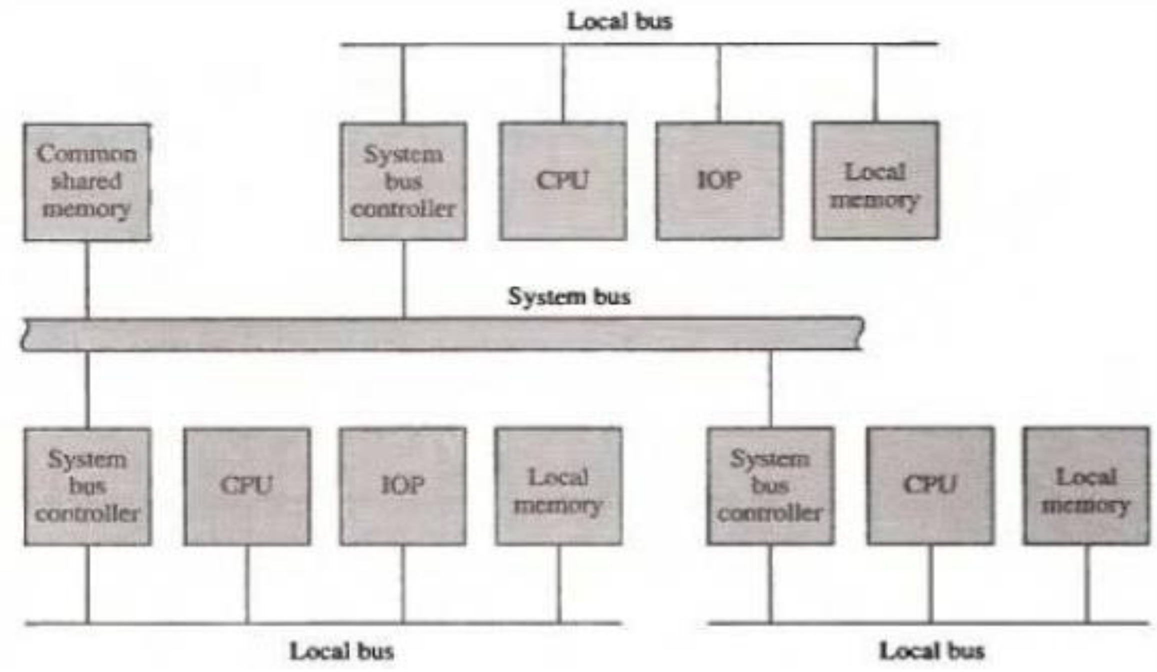
The system may exhibit transfer conflicts since all processors share one common bus. These conflicts must be resolved by putting a bus controller that establishes priorities among the requesting units.

The processors in the system are kept busy through the implementation of two or more buses to permit multiple simultaneous bus transfer. It increases the system cost and complexity.



Time shared common bus organization

Consider implementation of dual bus structure. In this a number of local buses connected to its own local memory and to one or more processors. Each local bus is connected to a CPU, an IOP or any combination of processors. A system bus controller links each local bus to common system bus. I/O devices connected to local memory as well as to local IOP are available to local processor



System bus structure for multiple processors

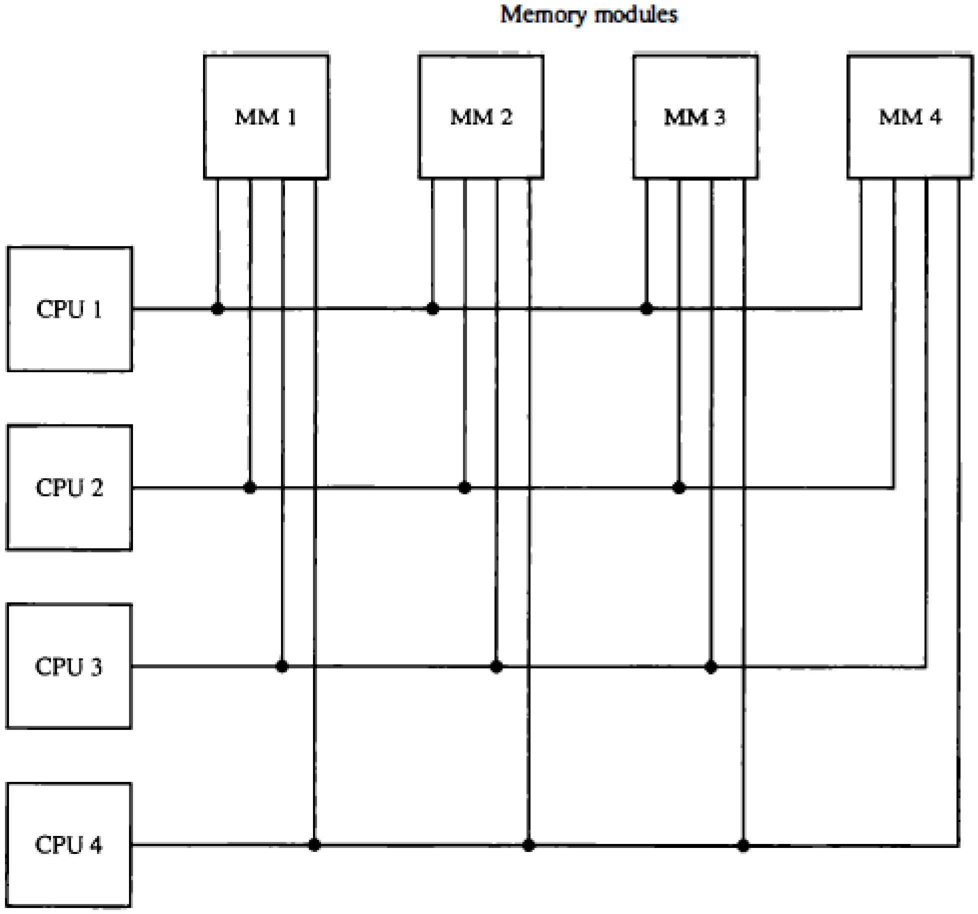
If an IOP is connected directly to the system bus, the I/O devices attached to it may be made available to the processors. Only one processor can communicate with the shared memory and other common resources through the system bus at any given time.

The other processors are kept busy communicating with their local memory and I/O devices. Part of local memory may be designed as a cache memory attached to CPU. In this way average access time of local memory can be made to approach the cycle time of CPU to which it is attached.

2. Multiport memory:

A multiport memory system employs separate buses between each memory module (MM) and each CPU. Each processor bus is connected to each memory module. A processor bus consists of the address, data and control lines required to communicate with memory. MM is said to have 4ports and each port accommodates one of the buses. The module must have internal control logic to determine which port will have access to memory at any given time.

The advantage of multiport memory organization is the higher transfer rate because of multiple paths between processor and main memory. The disadvantage is it requires expensive control logic and a large number of cables and connectors. This structure is appropriate for a system with large number of processors.



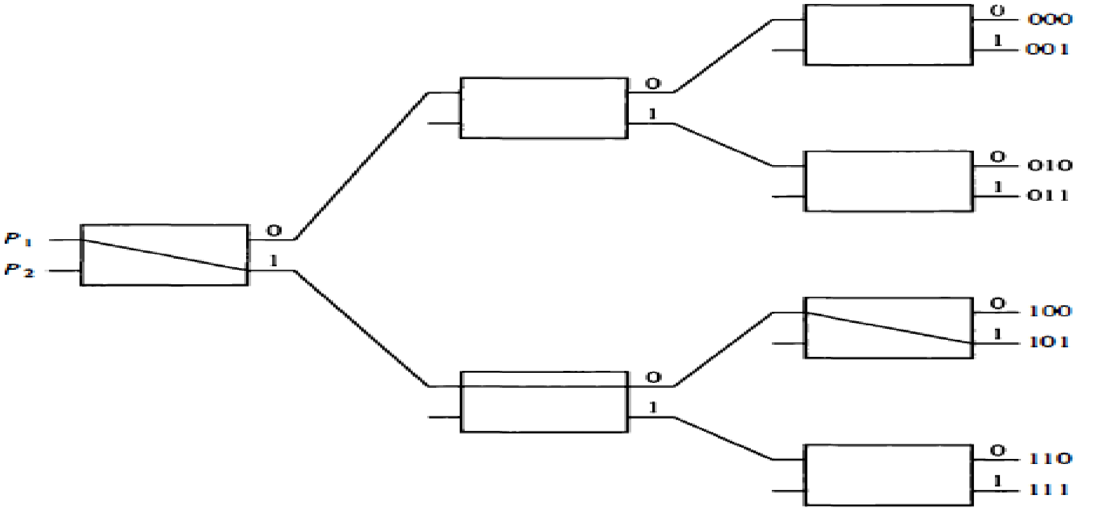
Multiport memory organization

4. Multistage Switching Network:

The basic component of a multistage network is a two-input, two-output interchange switch. A 2*2 switch has two inputs, labeled A and B and two-outputs labeled 0 and 1. The switch has the capability of connecting inputs A or B to either of the outputs. If inputs A and B both requests for the same output only one of them will be connected other will be blocked.

The two processors P1 and P2 are connected through switches to eight memory modules marked in binary from 000 through 111. The first bit of destination number determines the switch output in the first level.

For example to connect P1 to memory 101,it is necessary to form paths from P1 to o/p 1 in t he 1st level switch, o/p 0 in the 2nd level switch and o/p in the 3rd level.



Omega Network:

Omega switching network have been proposed for multistage switching network to control processor-memory communication. In this configuration there is exactly one path from each source to any particular destination. A particular request is initiated in the switching network by source, which sends a 3-bit pattern representing the destination number. As the binary pattern moves through the network, each level examines a different bit to determine the 2*2 switch setting. When the request arrives on either input of 2*2 switch, it is routed to the upper output if the specified bit is 0 or to the lower output if it is 1.

Omega Network in a tightly coupled multiprocessor:

In this the source is a memory module and the destination is a memory module. The first pass through the network sets up the path. Succeeding paths are used to transfer the address into memory and then transfer the data in read or write direction.

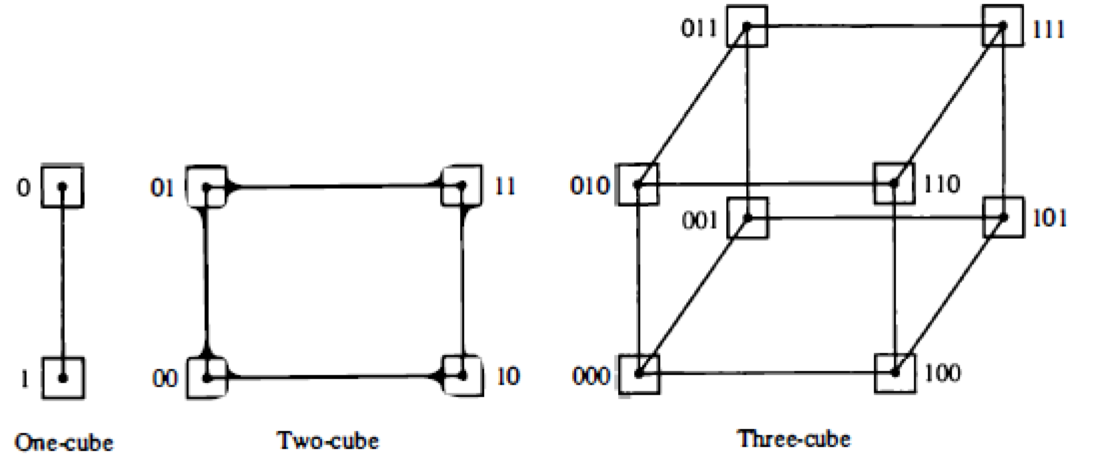
Omega Network in loosely coupled multiprocessor:

In this both the source and the destination are the processing elements. After the path is established the source processor transfers a message to the destination processor.

5. Hypercube Interconnection

The hypercube or binary n-cube multiprocessor structure is a loosely coupled system composed of 2 power of n processors interconnected in an ndimensional binary cube. Each processor forms a node of the cube. Each processor has direct communication paths to n other neighbor processors. Each processor address differs from that of its neighbors by exactly one bit position.

Routing messages through an n-cube structure may take from one to n links from a source node to a destination node. For example, in a three-cube structure node 000 can communicate directly with node 001. It must cross at least two links to communicate with 011. Computing the ex-or of the source node address with the destination node address can develop a routing procedure.



5.7. Interprocessor Arbitration

Computer systems contain a number of buses at various levels to facilitate the transfer of information between components. A bus that connects the major components in a multiprocessor system such as CPUs, IOPs and memory is called a system bus. The processors in a shared memory multiprocessor system request access to common memory through the system bus. The requesting processor may wait if another processor is currently utilizing the system bus. Arbitration must then be performed to resolve this multiple contention for the shared resources. Arbitration logic is a part of system bus controller placed between the local bus and the system bus.

System Bus:

A typical system bus consists of approximately 100 signal lines. These lines are divided into three functional groups data, address and control lines.

Six bus arbitration signals

These are used for Interprocessor arbitration.

Bus request	BREQ
Common Bus request	CBRQ
Bus busy	BUSY
Bus clock	BCLK
Bus priority in	BPRN
Bus priority out	BPRO

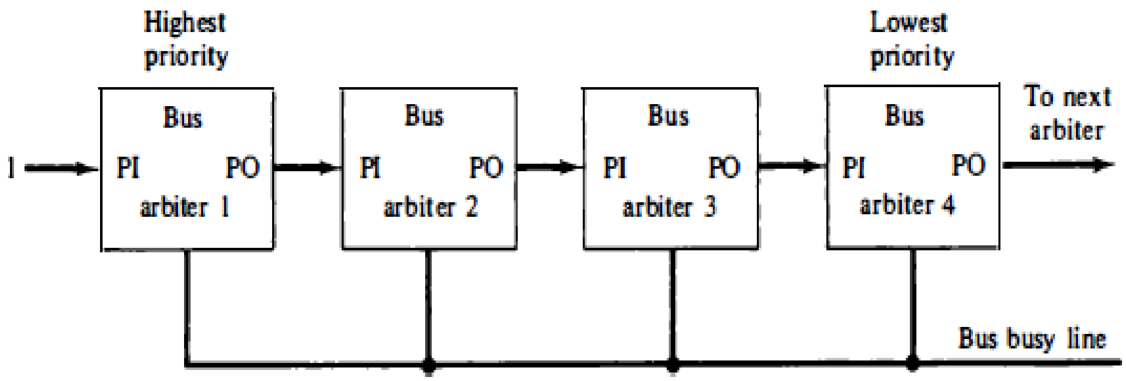
Functions of the Bus Arbitration Signals

The bus priority –in BPRN and the bus priority-out BPRO are used for a daisy-chain connection of bus arbitration circuits. The bus BUSY signal is an open collector output to instruct all arbiters when the bus is busy. The common bus request CBRQ is also an open collector output used to instruct the arbiter if there are any other arbiters of lower-priority requesting the use of bus. The signals used to connect parallel arbitration procedure are bus request BREQ and priority-in BRPN for request and ack signals respectively. The bus clock BCLK is used to synchronize all bus transactions

Serial Arbitration Procedure:

A hardware bus priority resolving techniques can be established by means of a serial or parallel connection of units requesting control of the system bus.

Serial priority resolving technique is obtained from a daisy-chain interconnection of bus arbitration circuits. Each processor has its own bus arbitration logic with priority in and priority out lines. The priority out (PO) of each arbiter is connected to priority in (PI) of next lower-priority. The PI of highest priority unit is maintained at logic 1.



Serial Arbitration

The processor whose arbiter has a PI=1 and PO=0 is one that is given the control of the system bus. The PO output for a particular arbiter is equal to 1 f its PI input is equal to 1 and the processor associated with the arbiter logic is not requesting the control of the bus.

Parallel Arbitration Logic:

The parallel arbitration logic uses an external priority encoder and a decoder. Each bus arbiter in this parallel scheme has a bus request output line and a bus ack input line. The processor takes control of the bus if its ack input line is enabled. First the request lines from the four arbiters 4*2 priority encoder. The output of the encoder generates a 2-bit code, which represents the highest priority unit among those requesting the bus. The 2-bit code from the encoder output drives a 2*4 decoder, which enables the proper ack line to grant bus access to the highest priority unit.

Dynamic Arbitration Algorithms:

In a dynamic priority algorithm gives the system the capability for changing the priority of the devices while the system is in operation. The few arbitration procedures that follow dynamic priority algorithms are:

Time slice: It allocates a fixed length time slice of bus time that is offered sequentially to each processor. The service given to each component is independent of its location along the bus. No preference is given to any particular since each is given same amount of time to communicate with the bus.

Polling: In a bus system that uses polling, poll lines replace bus grant signal.

The bus controller to define an address for each device connected to the bus uses Poll lines. After a number of bus cycles, the polling process continues by choosing different processor.

LRU: The least recently used (LRU) algorithm gives the highest priority to the requesting device that has not used the bus for the longest interval. With this procedure no processor is favored over any other since the priorities are dynamically changed to give every device an opportunity to access the bus.

FIFO: In the first-come, first-serve scheme, requests are served in the order received. For this the bus controller establishes a queue according to the time that the bus requests arrive. Each processor must wait for its turn to use the bus on FIFO basis.

Rotating daisy-chain: This procedure is a dynamic extension of daisy-chain algorithm. In this scheme there is no central bus controller and the priority line is connected from the priority-out of the last device back to the priority-in of the first device in a closed loop. Each arbiter priority for a given bus cycle is given by its position along the bus priority line from the arbiter whose processor is currently controlling the bus. Once, an arbiter releases the bus it has lowest priority.

5.8. Interprocessor Communication

The various processors in a multiprocessor system must be provided with a facility for communicating with each other.

In a shared memory multi processor system, the most common procedure is to set aside a portion of memory that is accessible to all processors. The primary use of common memory is to act as a message center similar to a mailbox, where each processor can leave messages for other processors and pick up messages intended for it. The sending processor structures a request, a message or a procedure and places it in the memory mail box, whether it has meaningful information, and which processor it is intended. The receiving processor can check the mailbox periodically if there are valid messages for it.

In addition to shared memory, a multiprocessor system may have other shared resources. To prevent conflict use of shared resources by several processors there must be a provision for assigning resources to these processors. This task is given to the operating system.

There are three organizations that have been used in the design of the operating system for multiprocessors:

1. Master-slave configuration: one processor designated the master, always executes the operating system functions. The remaining processors as slaves do not perform the operating system functions. If a slave processor needs an operating system service, it must request it by interrupting the master and waiting until the current program can be interrupted.

2. Separate operating system: Each processor can execute the operating system routines its needs. This is more suitable for loosely coupled systems where every processor may have its own copy of the entire os.

3. Distributed operating system: In this each particular os has one processor at a time. This type of organization is also called floating os since the routine floats from one processor to another and the execution of routine may be assigned to different processors at different times.

In a loosely coupled multiprocessor system the memory is distributed among the processors and there is no shared memory for passing information. The communications between processors is by means of message passing through I/O channels. When the sending processor and the receiving processor name each other as source and destination, a channel of communication is established.

5.9. Interprocessor Synchronization

Synchronization is needed to enforce the correct sequence of processes and to ensure mutually exclusive access to shared writable data. A number of hardware mechanisms for mutual exclusion have been developed. One of the most popular methods is binary semaphore.

Mutual exclusion with a Semaphore: It is necessary to protect data from being changed simultaneously by two or more processors. This mechanism has been termed mutual exclusion. Mutual exclusion must be provided in a multiprocessor system to enable one processor to exclude or lock out access to a shared resource by other processors when it is in a mutual section. It is a program sequence that, once begun, must complete execution before another processor accesses the same-shared resource.

A binary variable called a semaphore is often used to indicate whether or not a processor is executing a critical section. It is a software-controlled flag that is stored in a memory location that all processors can access. When it is equal to 1, it means that a processor is executing a critical program, so that the shared memory is not available to other processors. When it is equal to 0, the shared memory is available to any requesting processor.

Testing and setting the semaphore is itself a critical operation and must be performed as a single indivisible operation. This action would allow simultaneous execution of a critical section at the same time, which can result in erroneous initialization of control parameters and a loss of essential information.

A semaphore can be initialized by means of a test and set instruction in conjunction with a hardware lock mechanism. A hardware lock is a processor-generated signal that serves to prevent other processors from using the system bus as long as the signal is active. The other processor changes the semaphore between the time that the processor is testing it and the time that it is setting it.

Cache Coherence:

The primary advantage of cache is to its ability to reduce the average access time in uniprocessors. When a processor finds a writ operation in cache during write operation there are two commonly used procedures to update memory.

1. write-through policy:

Both cache and maim memory are updated with every write operation

2. Write back policy:

Only cache is updated and the location is marked so that it can be copied later into the main memory.

A memory scheme is coherent if the value returned on a load instruction is always the value given by the latest store instruction with the same address. Cache coherence problems exist in multiprocessors with private caches because of the need to share writable data.

Conditions for cache incoherence

To illustrate the problem of cache coherence, consider three processor configurations with private caches. Sometime during the operation an element X from main memory is loaded into the three processors P1, P2, P3. It is also copied into the private caches of the three processors. Consider X as 52. The load on X to the three processors results in consistent copies in the cache and main memory.

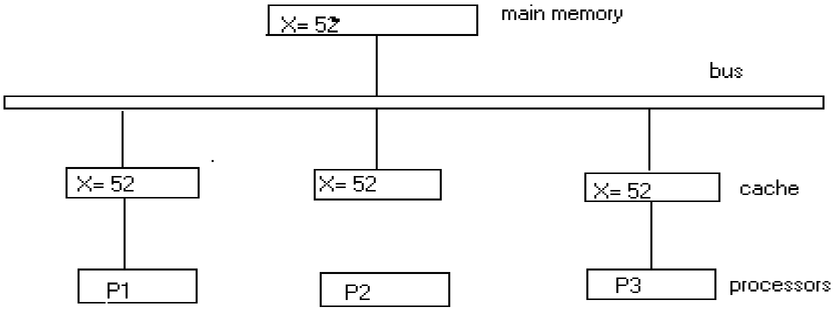
A store to X into the cache of processor P1 updates memory to the new value in a write through policy. A write through policy maintains consistency between memory and the originating cache, but the other two caches are inconsistent still they hold the same old value.

In writ back policy; main memory is not updated at the time of store. The copies in the other two caches and main memory are inconsistent. Memory is updated when the modified data is copied back into the main memory.

Solutions to the cache coherence problem:

Cache coherence problems can be solved by means of software and hardware combinations.

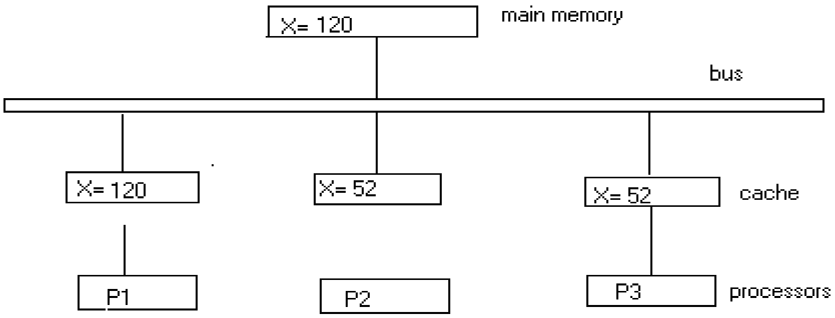
In the software solution, it is desirable to attach a private cache to each processor. If cache allows nonshared and read only data, such items are called cachable. Shared write only data are non cachable which remains in main memory. This method restricts the type of data stored in caches



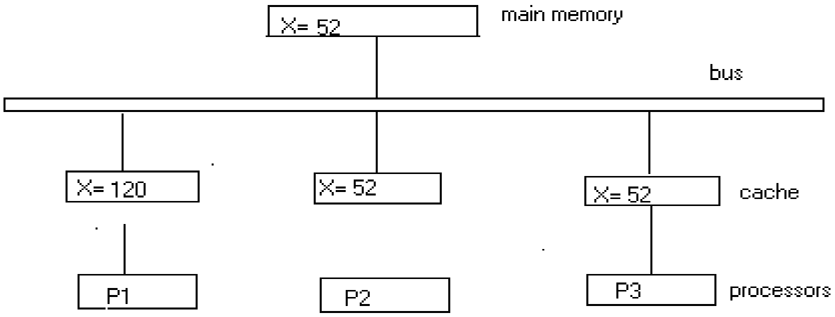
Cache configuration after a load X

Another scheme that allows writable data to exist in one cache is a method that employs centralized global table in its complier. Each block is identified as read only (RO) or write and read (WR). Only one copy of cache is RO. Thus if data is updated in RW block other caches are not affected because they do not have a copy of this block.

In the hardware solution, a cache controller called snoopy cache controller is specially designed to allow it to monitor all bus requests from CPUs and IOPs. All caches attached to the bus constantly monitor these network for possible write operations. It is basically a hardware unit designed to maintain a bus –watching mechanism over all the caches attaches to the bus. In this way inconsistent ways are prevented.



With write through cache policy



With write back cache policy