UNIT-I

Basic Structure of Computers: Basic Organization of Computers, Historical Perspective, Bus Structures, Data Representation: Data types, Complements, Fixed Point Representation. Floating, Point Representation. Other Binary Codes, Error Detection Codes.

Computer Arithmetic: Addition and Subtraction, Multiplication Algorithms, Division Algorithms.

1.Basic Organization 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.,

Functional unit

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

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.

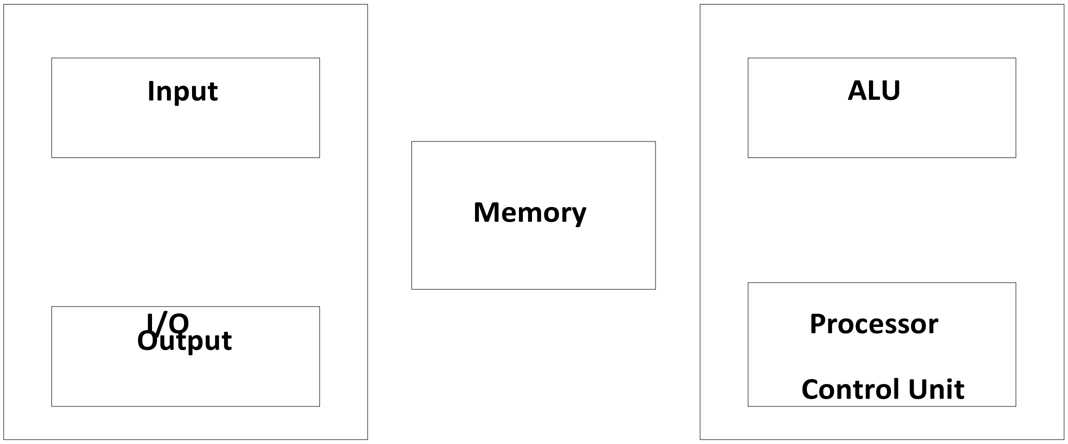


Fig . Functional units of computer

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 after 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 may 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.

1.2. 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,

6. First the instruction is fetched from the memory into the processor.
7. The operand at LOCA is fetched and added to the contents of R_0
8. 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 computer s, 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.

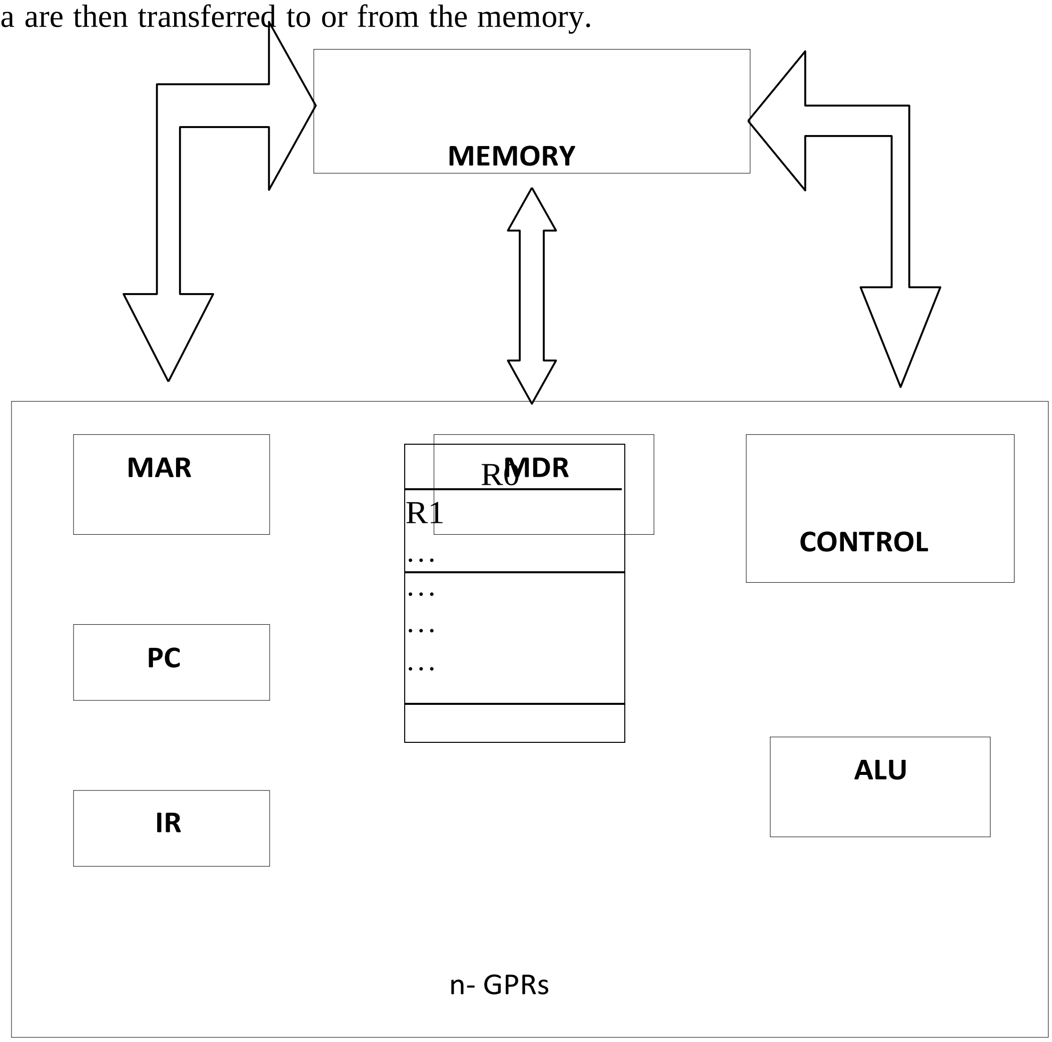


Fig b : Connections between the processor and the memory

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.

2. GENERATION OF COMPUTERS:

Development of technologies used to fabricate the processors, memories and I/O units of the computers has been divided into various generations as given below:

- First generation
- Second generation
- Third generation
- Fourth generation
- Beyond the fourth generation

First generation:

1946 to 1955: Computers of this generation used Vacuum Tubes. The computes were built using stored program concept. Ex: ENIAC, EDSAC, IBM 701.

Computers of this age typically used about ten thousand vacuum tubes. They were bulky in size had slow operating speed, short life time and limited programming facilities.

Second generation:

1955 to 1965: Computers of this generation used the germanium transistors as the active switching electronic device. Ex: IBM 7000, B5000, IBM 1401. Comparatively smaller in size About ten times faster operating speed as compared to first generation vacuum tube based computers. Consumed less power, had fairly good reliability. Availability of large memory was an added advantage.

Third generation:

1965 to 1975: The computers of this generation used the Integrated Circuits as the active electronic components. Ex: IBM system 360, PDP minicomputer etc. They were still smaller in size. They had powerful CPUs with the capacity of executing 1 million instructions per second (MIPS). Used to consume very less power consumption.

Fourth generation:

1976 to 1990: The computers of this generation used the LSI chips like microprocessor as their active electronic element. HCL horizen III, and WIPRO‟S Uniplus+ HCL‟s Busybee PC etc.
They used high speed microprocessor as CPU. They were more user friendly and highly reliable systems. They had large storage capacity disk memories.

Beyond Fourth Generation:

1990 onwards: Specialized and dedicated VLSI chips are used to control specific functions of these computers. Modern Desktop PC‟s, Laptops or Notebook Computers

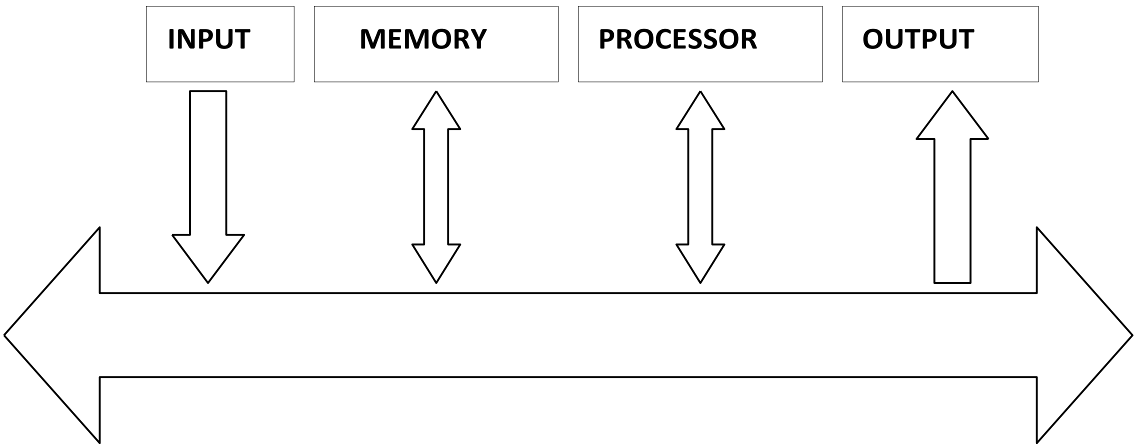
3. BUS STRUCTURES:

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

Group of lines that serve as connecting path for several devices is called a bus (one bit per line). Individual parts must communicate over a communication line or path for exchanging data, address and control information as shown in the diagram below. Printer example – processor to printer. A common approach is to use the concept of buffer registers to hold the content during the transfer.

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.

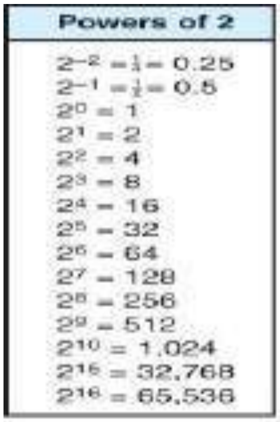
4. Data Representation: Data types:

- Bit: The most basic unit of information in a digital computer is called a **bit**, which is a contraction of binary digit.
- Byte: In 1964, the designers of the IBM System/360 main frame computer established a convention of using groups of 8 bits as the basic unit of **addressable** computer storage. They called this collection of **8 bits a byte**.
- Word: Computer words consist of two or more **adjacent** bytes that are sometimes addressed and almost always are manipulated collectively. The word size represents the data size that is handled **most efficiently** by a particular architecture. **Words** can be 16 bits, 32 bits, 64 bits.
- Nibbles: Eight-bit bytes can be divided into two 4-bit halves call **nibbles**.

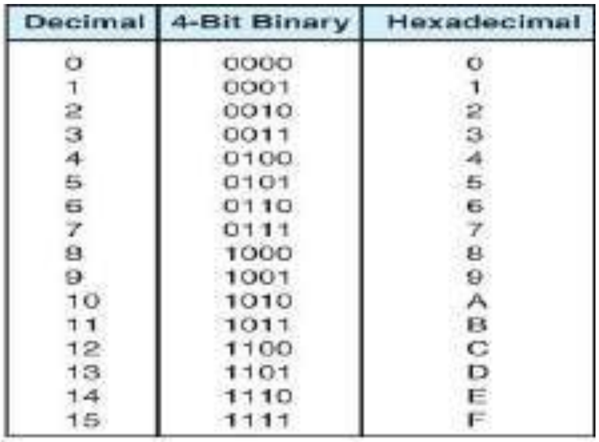
Number Systems:

- Radix (or Base): The general idea behind positional numbering systems is that a numeric value is represented through increasing powers of a **radix** (or base).

System	Radix	Allowable Digits
Decimal	10	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Binary	2	0, 1
Octal	8	0, 1, 2, 3, 4, 5, 6, 7
Hexadecimal	16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F



Powers of 2
$2^{-2} = \frac{1}{4} = 0.25$
$2^{-1} = \frac{1}{2} = 0.5$
$2^0 = 1$
$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$
$2^8 = 256$
$2^9 = 512$
$2^{10} = 1,024$
$2^{15} = 32,768$
$2^{16} = 65,536$



Decimal	4-Bit Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

TABLE 2.1 Some Number to Remember

- EXAMPLE 2.1 Three numbers represented as powers of a radix.

243.51_{10} $= 2 * 10^2 + 4 * 10^1 + 3 * 10^0 + 5 * 10^{-1} + 1 * 10^{-2}$
212_3 $= 2 * 3^2 + 1 * 3^1 + 2 * 3^0 = 23_{10}$

- There are two important groups of number base conversions:

1. Conversion of decimal numbers to base-r numbers
2. Conversion of base-r numbers to decimal numbers

- EXAMPLE 2.3 Convert 104_{10} to base 3 using the division-remainder method.

104_{10} = **10212_3**

3|104	2
3| 34	1
3| 11	2
3| 3	0
3|1	1
0	

- EXAMPLE 2.4 Convert 147_{10} to binary 147_{10}

= **10010011_2**

2|147	1
2| 73	1
2|36	0
2|18	0
2|9	1
2|4	0

```
2|2      0
2|1      1
0
```

- A binary number with N bits can represent unsigned integer from 0 to $2^n - 1$.

- **Overflow**: the result of an arithmetic operation is outside the range of allowable precision for the give number of bits.

Converting Fractions:

- EXAMPLE 2.6 Convert 0.4304_{10} to base 5.

  $0.4304_{10} = 0.2034_5$

- EXAMPLE 2.7 Convert 0.34375_{10} to binary with 4 bits to the right of the binary point. Reading from top to bottom, $0.34375_{10} = 0.0101_2$ to four binary places. We simply **discard** (or truncate) our answer when the desired accuracy has been achieved.

  $0.34375_{10} = \mathbf{0.0101_2}$

```
  0.34375
X        2
0.68750 X2
1.37500 X2
0.75000 X2
1.50000
```

- EXAMPLE 2.8 Convert 3121_4 to base 3

  First, convert to decimal $3121_4 = 217_{10}$ Then convert to base 3 $217_{10} = 22001_3$
  We have $3121_4 = 22001_3$

Converting between Power-of-Two Radices:

- EXAMPLE 2.9 Convert 110010011101_2 to octal and hexadecimal.

  $\underline{110\ 010\ 011\ 101}_2 = 6235_8$ Separate into groups of **3** for octal conversion

  $\underline{1100\ 1001\ 1101}_2 = C9D_{16}$ Separate into groups of **4** for octal conversion

5. Signed Integer Representation:

- By convention, a “1” in the **high-order bit** indicate a negative number.

Signed Magnitude:

- A signed-magnitude number has a sign as its left-most bit (also referred to as the high-order bit or the most significant bit) while the remaining bits represent the magnitude (or **absolute value**) of the numeric value.
- N bits can represent **$-(2^{n-1} - 1)$ to $2^{n-1} - 1$**

- EXAMPLE 2.10 Add 01001111_2 to 00100011_2 using signed-magnitude arithmetic.

  01001111_2 (79) + 00100011_2 (35) = 01110010_2 (114)
  There is no overflow in this example

- EXAMPLE 2.11 Add 01001111_2 to 01100011_2 using signed-magnitude arithmetic.
- An overflow condition and the carry is **discarded**, resulting in an **incorrect** sum.

  We obtain the erroneous result of
  01001111_2 (79) + 01100011_2 (99) = 0110010_2 (50)

- EXAMPLE 2.12 Subtract 01001111_2 from 01100011_2 using signed-magnitude arithmetic.

  We find 011000011_2 (99) - 01001111_2 (79) = 00010100_2 (20)
  in signed-magnitude representation.

- EXAMPLE 2.14
- EXAMPLE 2.15
- The signed magnitude has **two** representations for **zero**, **10000000** and **00000000** (and mathematically speaking, the simple shouldn’t happen!).

Complement’s:

- ***One’s Complement***
  - This sort of bit-flipping is very simple to implement in computer hardware.
  - EXAMPLE 2.16 Express 2310 and -910 in 8-bit binary one’s complement form.

    23_{10} = + (00010111_2) = 00010111_2
    -9_{10} = - (00001001_2) = 11110110_2

  - EXAMPLE 2.17
  - EXAMPLE 2.18
- The primary disadvantage of one’s complement is that we still have **two**

representations for **zero**: **00000000** and **11111111**

- ***Two's Complement***
  - Find the one's complement and add 1.
  - EXAMPLE 2.19 Express 23_{10}, -23_{10}, and -9_{10} in 8-bit binary two's complement form.

    $23_{10} = +(00010111_2) = 00010111_2$
    $-23_{10} = -(00010111_2) = 11101000_2 + 1 = 11101001_2$
    $-9_{10} = -(00001001_2) = 11110110_2 + 1 = 11110111_2$

  - EXAMPLE 2.20 Add 9_{10} to -23_{10} using two's complement arithmetic.

    00001001_2 (9_{10}) + 11101001_2 (-23_{10}) = 11110010_2 (-14_{10})

00001001	<= Carries
00001001_2	(9)
+ 11101001_2	+(-23)
11110010_2	(-14)

  - EXAMPLE 2.21 Find the sum of 23_{10} and -9_{10} in binary using two's complement arithmetic.

    00010111_2 (23_{10}) + 11110111_2 (-9_{10}) = 00001110_2 (14_{10})

11110111	<= Carries
00010111_2	(23)
+ 11110111_2	+(- 9)
00001110_2	(14)

  - **A Simple Rule for Detecting an Overflow Condition**: *If the carry in the sign bit* ***equals*** *the carry out of the bit, no overflow has occurred. If the carry into the sign bit is* ***different*** *from the carry out of the sign bit, over (and thus an error) has occurred.*
  - EXAMPLE 2.22 Find the sum of 126_{10} and 8_{10} in binary using two's complement arithmetic.

    01111110_2 (126_{10}) + 00001000_2 (8_{10}) = 10000110_2 (-122_{10})

01111000	<= Carries
01111110_2	(126)
+ 00001000_2	+(8)
10000110_2	(-122)

A one is carried into the leftmost bit, but a zero is carried out. Because these carries are not equal, an overflow has occurred.

  - N bits can represent **$-(2^{n-1})$ to 2^{n-1} -1**. With signed-magnitude number, for example, 4 bits allow us to represent the value -7 through +7. However using two's complement, we can represent the value -8 through +7.

- Integer Multiplication and Division
  - For each digit in the multiplier, the multiplicand is "shifted" one bit to the **left**. When the multiplier is 1, the "shifted" multiplicand is added to a running sum of partial products.
  - EXAMPLE Find the product of 00000110_2 (6_{10}) and 00001011_2 (11_{10}).

00000110 (6)

x 00001011 (11)

Multiplicand		**Partial Products**
00000110	+	**00000000** (1; add multiplicand and shift left)
00001100	+	**00000110** (1; add multiplicand and shift left)
00011000	+	**00010010** (0; Don't add, just shift multiplicand left)
00110000	+	**00010010** (1; add multiplicand and shift left)
	=	**01000010** (**Product**; 6 X 11 = **66**)

- When the divisor is much smaller than the dividend, we get a condition known as **divide underflow**, which the computer sees as the equivalent of division by **zero**.

- Computer makes a distinction between integer division and floating-point division.
- With integer division, the answer comes in two parts: a **quotient** and a ***remainder***.
  - Floating-point division results in **a number** that is expressed as a binary fraction.
  - Floating-point calculations are carried out in dedicated circuits call floating- point units, or FPU.

Unsigned numbers vs Signed numbers:

- If the 4-bit binary value 1101 is unsigned, then it represents the decimal value 13, but as a signed two's complement number, it represents -3.
- C programming language has **int** and **unsigned int** as possible types for integer variables.
- If we are using 4-bit unsigned binary numbers and we add 1 to 1111, we get 0000 ("return to zero").
- If we add 1 to the largest positive 4-bit two's complement number 0111 (+7), we get 1000 (-8).

Computers, Arithmetic, and Booth's Algorithm:

- Consider the following standard pencil and paper method for multiplying two's complement numbers (-5 X -4):

1011 (-5)

x 1100 (-4)

+ 0000 (0 in multiplier means simple shift)

+ 0000 (0 in multiplier means simple shift)

+ 1011 (1 in multiplier means add multiplicand and shift)

+ 1011 (1 in multiplier means add multiplicand and shift)

10000100 (-4 X -5 = **-124**)

Note that: "Regular" multiplication clearly yields the **incorrect** result.

- Research into finding better arithmetic algorithms has continued apace for over 50 years. One of the many interesting products of this work is Booth's algorithm.
- In most cases, Booth's algorithm carries out multiplication faster and more accurately than naïve pencil-and-paper methods.
- The general idea of Booth's algorithm is to increase the speed of a multiplication when there are consecutive zeros or ones in the multiplier.
- Consider the following standard multiplication example (3 X 6):

0011 (3)

x 0110 (6)

+ 0000 (0 in multiplier means simple shift)

+ 0011 (1 in multiplier means add multiplicand and shift)

+ 0011 (1 in multiplier means add multiplicand and shift)

+ 0000 (0 in multiplier means simple shift)

0010010 (3 X 6 = **18**)

- In Booth's algorithm, if the multiplicand and multiplier are **n-bit** two's complement numbers, the result is a **2n-bit** two's complement value. Therefore, when we perform our intermediate steps, we must **extend** our n-bit numbers to 2n-bit numbers. For example, the 4-bit number 1000 (-8) extended to 8 bits would be 11111000.

- Booth's algorithm is interested in pairs of bits in the multiplier and proceed according to the following rules:

  - If the current multiplier bit is 1 and the preceding bit was, we are at the beginning of a string of ones, so **subtract (10 pair)** the multiplicand form the product.
  - If the current multiplier bit is 0 and the preceding bit was 1, we are at the end of a string of ones, so we **add (01 pair)** the multiplicand to the product.
  - If it is a **00** pair, or a **11** pair, do no arithmetic operation (we are in the middle of a string of zeros or a string of ones). Simply **shift**. The power of the algorithm is in this step: we can now treat a string of ones as a string of zeros and do nothing more than shift.

0011 (3)

x 0110 (6)

+ 00000000 (00 = simple **shift;** assume a mythical 0 as the previous bit)

+ 11111101 (10 = **subtract** = add 1111 1101, extend sign)

+ 00000000 (11 simple **shift**)

+ 00000011 (01 = **add**)

01000010010(3 X 6 = **18; 010 ignore extended sign** bit that go beyond 2n)

Note that: **010** Ignore extended sign bit that go beyond 2n.

- EXAMPLE 2.23 (-3 X 5) Negative 3 in 4-bit two's complement is 1101. Extended to 8 bits, it is 11111101. Its complement is 00000011. When we see the rightmost 1 in the multiplier, it is the beginning of a string of 1s, so we treat it as if it were the string 10:

1101 (-3; for subtracting, we will add -3's complement, or 00000011)

x 0101 (5)

+ 00000011 (10 = **subtract** 1101 = add 0000 0011)

+ 11111101 (01 = **add** 1111 1101 to product: note sign extension)

+ 00000011 (10 = **subtract** 1101 = add 0000 0011)

+ 11111101 (01 = **add** 1111 1101 to product)

100111110001 (-3 X 5 = **-15;** using the 8 rightmost bits**, 11110001 or -15**)

Note that: Ignore extended sign bit that go beyond 2n.

- EXAMPLE 2.24 Let’s look at the larger example of 53 X 126:

00110101 (53; for subtracting, we will add the complement of 53 or 11001011)

x 01111110 (126)

+0000000000000000 (00 = **simple shift**)

+111111111001011 (10 = **subtract** = add 11001011, sign extension)

+00000000000000(11 = **simple shift**)

+0000000000000 (11 = **simple shift**)

+000000000000 (11 = **simple shift**)

+00000000000 (11 = **simple shift**)

+0000000000 (11 = **simple shift**)

+000110101 (01 = **add** 00110101, sign extension)

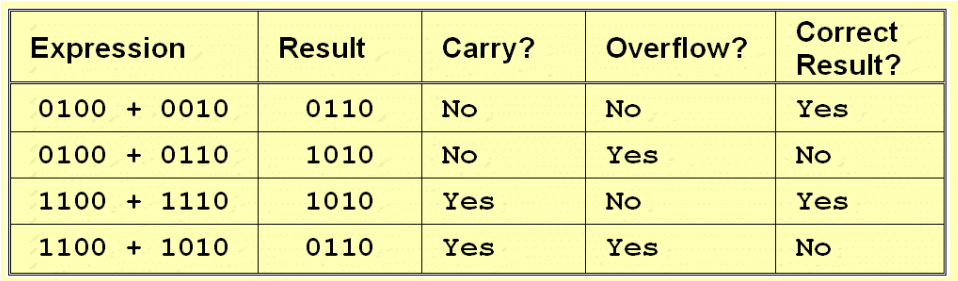
1000110100001011O (53 X 126 = **6678;** using the 16 rightmost bits)

Note that: Ignore extended sign bit that go beyond 2n.

- Booth’s algorithm not only allows multiplication to be performed **faster** in most cases, but it also has the added bonus in that it works correctly on **signed** numbers.

Carry Versus Overflow

- For **unsigned** numbers, a **carry** (out of the leftmost bit) indicates the total number of bits was not large enough to hold the resulting value, and overflow has occurred.
- For **signed** numbers, if the carry in to the sign bit and the carry (out of the sign bit) **differ**, then overflow has occurred.



Expression	Result	Carry?	Overflow?	Correct Result?
0100 + 0010	0110	No	No	Yes
0100 + 0110	1010	No	Yes	No
1100 + 1110	1010	Yes	No	Yes
1100 + 1010	0110	Yes	Yes	No

TABLE 2.2 Examples of Carry and Overflow in Signed Numbers

Binary Multiplication and Division Using Shifting:

- We can do binary multiplication and division by 2 very easily using an arithmetic shift operation
- A **left** arithmetic shift inserts a 0 in for the rightmost bit and shifts everything else left one bit; in effect, it **multiplies** by 2
- A **right** arithmetic shift shifts everything one bit to the right, but copies the sign bit; it **divides** by 2

- EXAMPLE 2.25: Multiply the value 11 (expressed using 8-bit signed two's complement representation) by 2.

  We start with the binary value for 11:
  00001011 (+11)
  We shift left one place, resulting in:
  00010110 (+22)
  The sign bit has not changed, so the value is valid.

  To multiply 11 by **4**, we simply perform a **left shift twice**.

- EXAMPLE 2.28: Divide the value 12 (expressed using 8-bit signed two's complement representation) by 2.

  We start with the binary value for 12:
  00001100 (+12)
  We shift left one place, resulting in:
  00000110 (+6)
  (Remember, we carry the sign bit to the left as we shift.) To

  divide 12 by **4**, we **right shift twice**.

6. Floating-Point Representation:

- In scientific notion, numbers are expressed in two parts: a **fractional** part call a mantissa, and an **exponential** part that indicates the power of ten to which the mantissa should be raised to obtain the value we need.

A Simple Model :

- In digital computers, floating-point number consist of three parts: a **sign** bit, an **exponent** part (representing the exponent on a power of 2), and a fractional part called a significand (which is a fancy word for a mantissa).

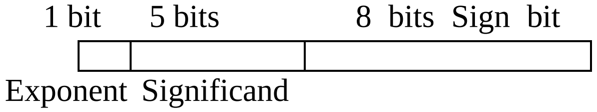


FIGURE 2.1 Simple Model Floating-Point Representation

- Unbiased Exponent

0	00101	10001000

$17_{10} = 0.10001_2 * 2^5$

0	10001	10000000

$65536_{10} = 0.1_2 * 2^{17}$

- Biased Exponent: We select 16 because it is **midway** between 0 and 31 (our exponent has 5 bits, thus allowing for 2^5 or 32 values). Any number **larger** than 16 in the exponent field will represent a positive value. Value **less** than 16 will indicate negative values.

0	10101	10001000

$17_{10} = 0.10001_2 * 2^5$

0	01111	10000000

The biased exponent is 16 + 5 = 21

- EXAMPLE 2.31

$0.25_{10} = 0.1_2 * 2^{-1}$

- A **normalized** form is used for storing a floating-point number in memory. A normalized form is a floating-point representation where the leftmost bit of the significand will always be a 1.
  Example: Internal representation of $(10.25)_{10}$

$(10.25)_{10} = (1010.01)_2$ (Un-normalized form)

$= (1010.01)_2 \text{ x } 2^0$.

$= (101.001)_2 \text{ x } 2^1$.

:

$= (.101001)_2 \text{ x } 2^4$ (**Normalized form**)

$= (.0101001)_2 \text{ x } 2^5$ (Un-normalized form)

$= (.00101001)_2 \text{ x } 2^6$.

Internal representation of $(10.25)_{10}$ is 0 10100 10100100

Floating-Point Arithmetic

- EXAMPLE 2.32: Add the following binary numbers as represented in a normalized 14-bit format, using the simple model with a bias of 16.

0	10010	11001000

+

0	10000	10011010

11.001000

+ 0.10011010

11.10111010

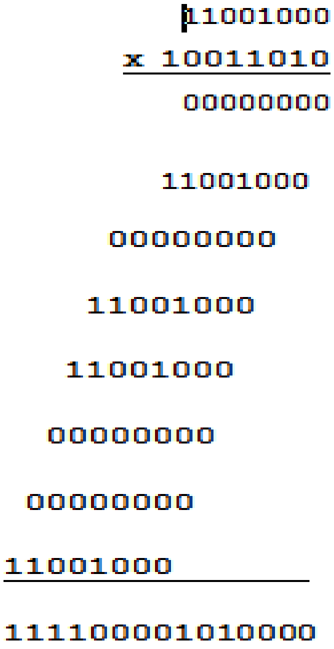
Renormalizing we retain the larger exponent and **truncate** the low-order bit.

0	10010	11101110

- EXAMPLE 2.33 Multiply:

X

0	10010	11001000	$= 0.11001000 \text{ X } 2^2$
0	10000	10011010	$= 0.10011010 \text{ X } 2^0$



```
          11001000
        x 10011010
          00000000
         11001000
       00000000
      11001000
     11001000
    00000000
   00000000
  11001000
  111100001010000
```

Renormalizing 0.0111100001010000 * 2^2 = 0.11110000**1010000** * 2^1 we retain the larger exponent and **truncate** the low-order bit.

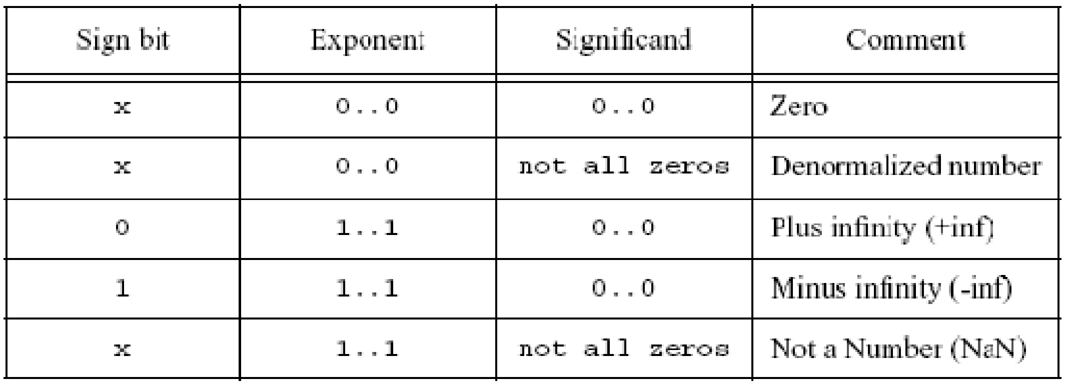
0	10001	11110000

Floating-Point Errors:

- We intuitively understand that we are working in the system of real number. We know that this system is **infinite**.
- Computers are **finite** systems, with **finite** storage. The more bits we use, the better the **approximation**. However, there is always some element of error, no matter how many bits we use.

The IEEE-754 Floating-Point Standard:

- The IEEE-754 **single** precision floating point standard uses bias of 127 over its **8-bit** exponent. An exponent of 255 indicates a special value.
- The **double** precision standard has a bias of 1023 over its **11-bit** exponent. The "special" exponent value for a double precision number is 2047, instead of the 255 used by the single precision standard.



Sign bit	Exponent	Significand	Comment
x	0..0	0..0	Zero
x	0..0	not all zeros	Denormalized number
0	1..1	0..0	Plus infinity (+inf)
1	1..1	0..0	Minus infinity (-inf)
x	1..1	not all zeros	Not a Number (NaN)

Special bit patterns in IEEE-754

Range, Precision, and Accuracy:

- The range of a numeric integer format is the difference between the largest and smallest values that is can express.
- The precision of a number indicates how much information we have about a value
- Accuracy refers to how closely a numeric representation approximates a true value.

Additional Problems with Floating-Point Numbers:

- Because of truncated bits, you cannot always assume that a particular floating point operation is commutative or distributive.

This means that we **cannot** assume: (a + b) + c = a + (b + c)

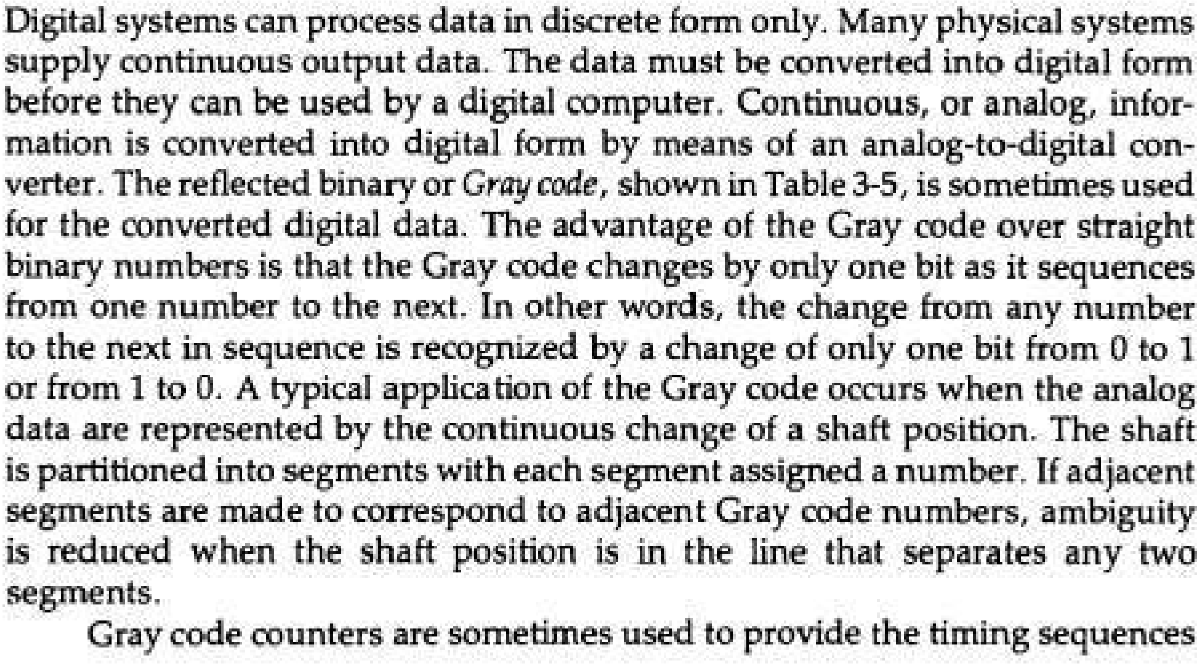
or

a * (b + c) = ab + ac

7.Other Binary Codes:

Other binary codes for decimal numbers and alphanumeric characters are sometimes used. Digital computers also employ other binary code for special applications.

Gray Code:



Digital systems can process data in discrete form only. Many physical systems supply continuous output data. The data must be converted into digital form before they can be used by a digital computer. Continuous, or analog, information is converted into digital form by means of an analog-to-digital converter. The reflected binary or *Gray code*, shown in Table 3-5, is sometimes used for the converted digital data. The advantage of the Gray code over straight binary numbers is that the Gray code changes by only one bit as it sequences from one number to the next. In other words, the change from any number to the next in sequence is recognized by a change of only one bit from 0 to 1 or from 1 to 0. A typical application of the Gray code occurs when the analog data are represented by the continuous change of a shaft position. The shaft is partitioned into segments with each segment assigned a number. If adjacent segments are made to correspond to adjacent Gray code numbers, ambiguity is reduced when the shaft position is in the line that separates any two segments.

Gray code counters are sometimes used to provide the timing sequences

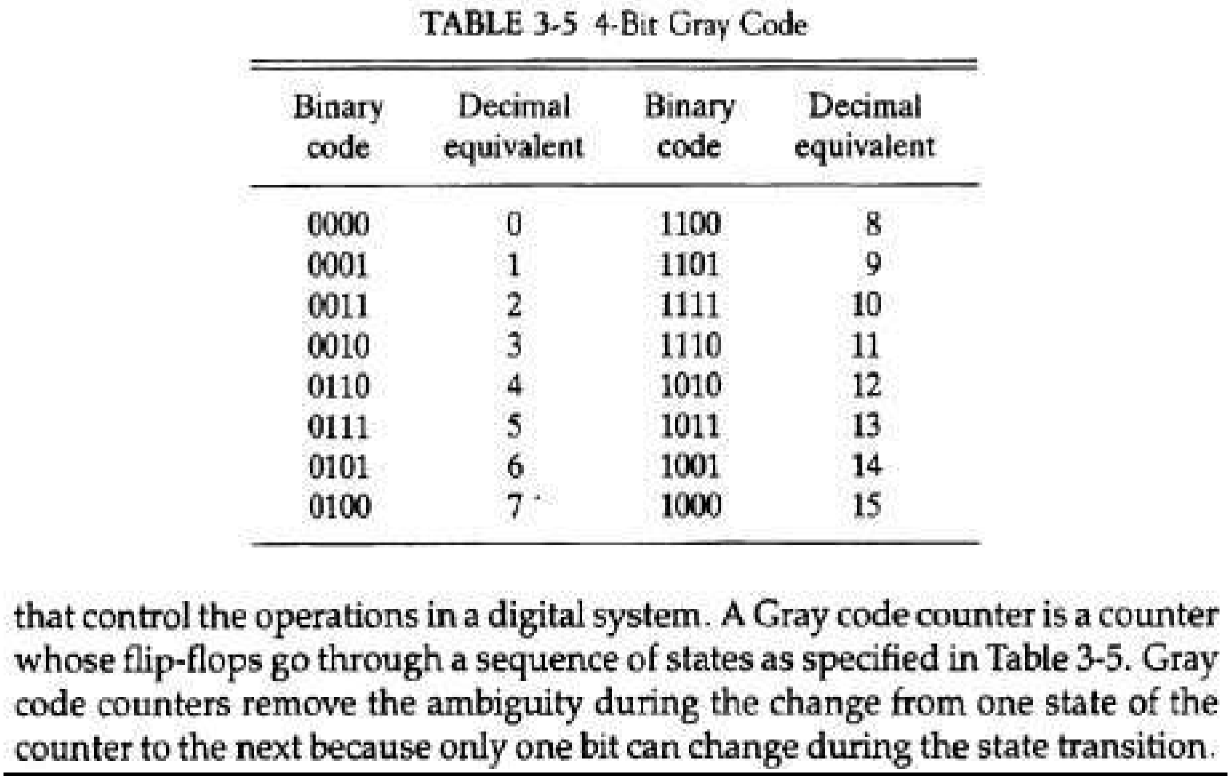
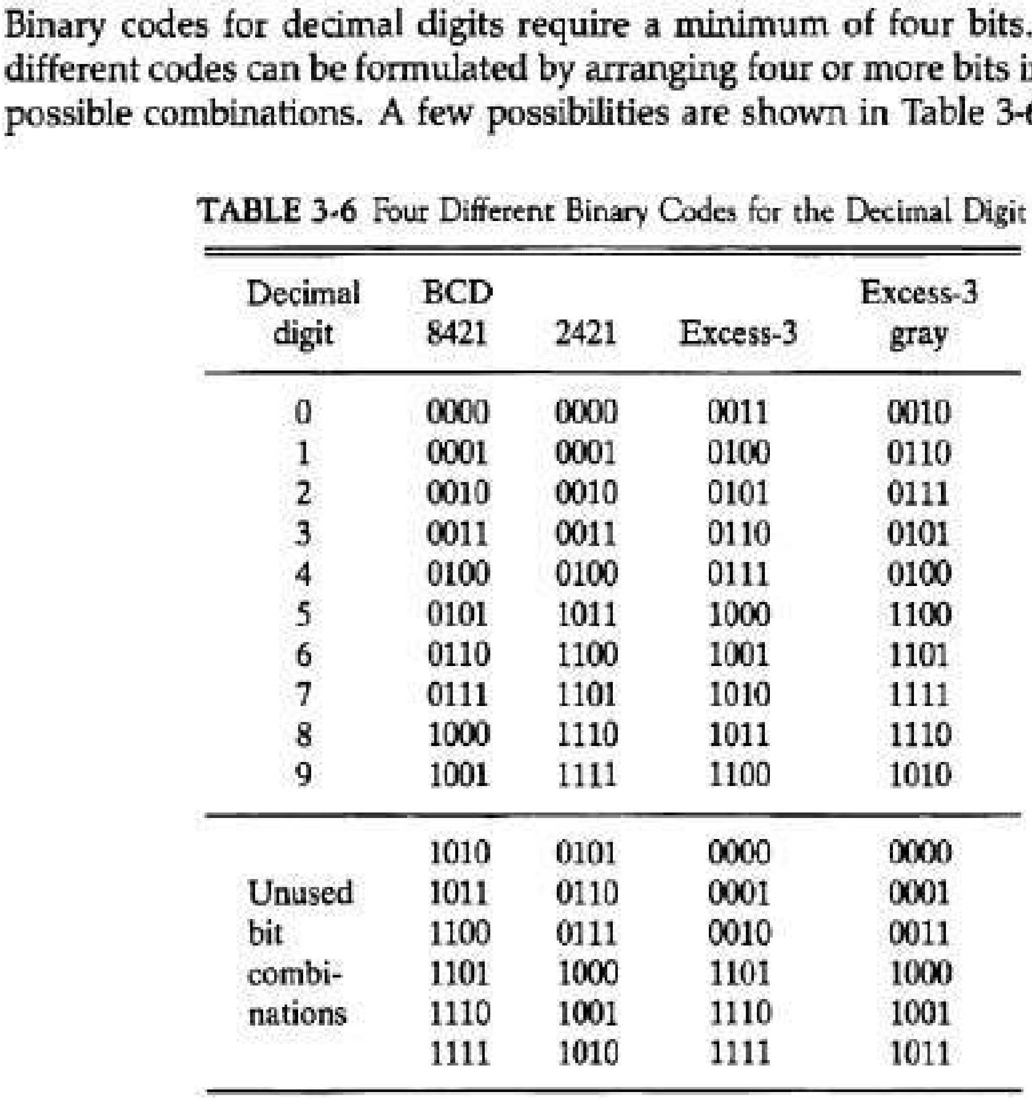


TABLE 3-5 4-Bit Gray Code

Binary code	Decimal equivalent	Binary code	Decimal equivalent
0000	0	1100	8
0001	1	1101	9
0011	2	1111	10
0010	3	1110	11
0110	4	1010	12
0111	5	1011	13
0101	6	1001	14
0100	7	1000	15

that control the operations in a digital system. A Gray code counter is a counter whose flip-flops go through a sequence of states as specified in Table 3-5. Gray code counters remove the ambiguity during the change from one state of the counter to the next because only one bit can change during the state transition.

Other Decimal Codes:



Binary codes for decimal digits require a minimum of four bits. Numerous different codes can be formulated by arranging four or more bits in 10 distinct possible combinations. A few possibilities are shown in Table 3-6.

TABLE 3-6 Four Different Binary Codes for the Decimal Digit

Decimal digit	BCD 8421	2421	Excess-3	Excess-3 gray
0	0000	0000	0011	0010
1	0001	0001	0100	0110
2	0010	0010	0101	0111
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1011	1000	1100
6	0110	1100	1001	1101
7	0111	1101	1010	1111
8	1000	1110	1011	1110
9	1001	1111	1100	1010
Unused bit combinations	1010	0101	0000	0000
	1011	0110	0001	0001
	1100	0111	0010	0011
	1101	1000	1101	1000
	1110	1001	1110	1001
	1111	1010	1111	1011

Other Alphanumeric codes:

EBCDIC:

- In 1964, BCD was extended to an 8-bit code, Extended Binary-Coded Decimal Interchange Code (EBCDIC).
- EBCDIC was one of the first widely-used computer codes that supported upper and lowercase alphabetic characters, in addition to special characters, such as punctuation and control characters.
- EBCDIC and BCD are still in use by IBM mainframes today. See Page 77 TABLE 2.6

ASCII

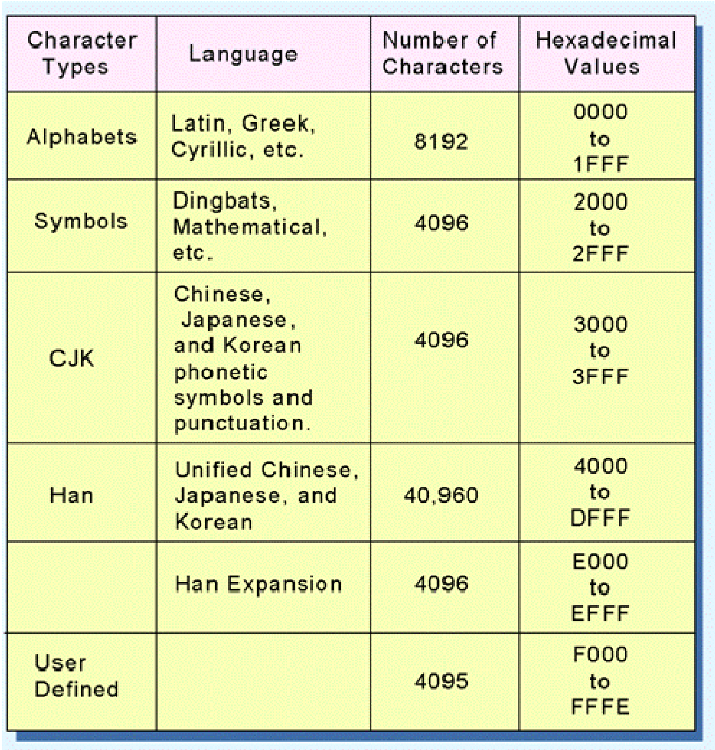
- ASCII: American Standard Code for Information Interchange
- In 1967, a derivative of this alphabet became the official standard that we now call ASCII.

Unicode

- Both EBCDIC and ASCII were built around the Latin alphabet.
- In 1991, a new international information exchange code called Unicode.
- Unicode is a **16-bit** alphabet that is downward compatible with ASCII and Latin-1 character set.
- Because the base coding of Unicode is 16 bits, it has the capacity to encode the majority of characters used

in every language of the world.

- Unicode is currently the default character set of the **Java** programming language.



Character Types	Language	Number of Characters	Hexadecimal Values
Alphabets	Latin, Greek, Cyrillic, etc.	8192	0000 to 1FFF
Symbols	Dingbats, Mathematical, etc.	4096	2000 to 2FFF
CJK	Chinese, Japanese, and Korean phonetic symbols and punctuation.	4096	3000 to 3FFF
Han	Unified Chinese, Japanese, and Korean	40,960	4000 to DFFF
	Han Expansion	4096	E000 to EFFF
User Defined		4095	F000 to FFFE

TABLE 2.8 Unicode Codespace

- The Unicode codespace is divided into six parts. The first part is for Western alphabet codes, including English, Greek, and Russian.
- The lowest-numbered Unicode characters comprise the ASCII code.
- The highest provide for user-defined codes.

8. Error Detection and Correction:

- **No** communications channel or storage medium can be completely error-free.

Cyclic Redundancy Check:

- Cyclic redundancy check (CRC) is a type of checksum used primarily in data communications that determines whether an error has occurred within a large block or stream of information bytes.
- Arithmetic Modulo 2The rules are as follows:

$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = 0$

- EXAMPLE 2.35 Find the sum of 1011_2 and 110_2 modulo 2.

  $1011_2 + 110_2 = 1101_2$ (mod 2)

- EXAMPLE 2.36 Find the quotient and remainder when 1001011_2 is divided by 1011_2.

Quotient 1010_2 and Remainder 101_2

- Calculating and Using CRC
  - o Suppose we want to transmit the information string: 1001011_2.
  - o The receiver and sender decide to use the (arbitrary) **polynomial pattern**, 1011.
  - o The information string is shifted left by one position less than the number of positions in the divisor. I = 1001011**000**$_2$
  - o The remainder is found through modulo 2 division (at right) and added to the information string: 1001011000_2 + **100$_2$** = 1001011100_2.
  - o If no bits are lost or corrupted, dividing the received information string by the agreed upon pattern will give a remainder of **zero**.
  - o We see this is so in the calculation at the right.
  - o Real applications use longer polynomials to cover larger information strings.
- A remainder other than **zero** indicates that an error has occurred in the transmission.
- This method work best when a large **prime** polynomial is used.
- There are four standard polynomials used widely for this purpose:
  - o CRC-CCITT (ITU-T): $X^{16} + X^{12} + X^5 + 1$
  - o CRC-12: $X^{12} + X^{11} + X^3 + X^2 + X + 1$
  - o CRC-16 (ANSI): $X^{16} + X^{15} + X^2 + 1$
  - o CRC-32: $X^{32} + X^{26} + X^{23} + X^{22} + X^{16} + X^{12} + X^{11} + X^{10} + X^8 + X^7 + X^6 + X^4 + X + 1$
- CRC-32 has been proven that CRCs using these polynomials can detect over **99.8%** of all single-bit errors.

<u>Hamming Codes:</u>

- Data communications channels are simultaneously more error-prone and more tolerant of errors than disk systems.
- Hamming code use parity bits, also called check bits or redundant bits.
- The final word, called a code word is an n-bit unit containing m data bits and r check bits.

  n = m + r
- The Hamming distance between two code words is the number of bits in which two code words differ.

  10001001
  10110001
  ***   Hamming distance of these two code words is 3
- The minimum Hamming distance, D(min), for a code is the smallest Hamming distance between all pairs of words in the code.
- Hamming codes can detect (D(min) - 1) errors and correct [(D(min) – 1) / 2] errors.
- EXAMPLE 2.37
- EXAMPLE 2.38 00000

  01011
  10110
  11101
  D(min) = 3. Thus, this code can detect up to **two** errors and correct **one** single bit error.

- We are focused on **single bit error**. An error could occur in any of the n bits, so each code word can be associated with n erroneous words at a Hamming distance of 1.
- Therefore, we have n + 1 bit patterns for each code word: **one valid** code word, and **n erroneous** words. With n-bit code words, we have 2^n possible code words consisting of 2^m data bits (where m = n + r).

  This gives us the inequality: $(n + 1) * 2^m < = 2^n$

Because m = n + r, we can rewrite the inequality as:

$(m + r + 1) * 2^m <= 2^{m + r}$ or

$\boldsymbol{(m + r + 1) <= 2^r}$

- EXAMPLE 2.39 Using the Hamming code just described and even parity, encode the 8-bit ASCII character K. (The high-order bit will be zero.) Induce a single-bit error and then indicate how to locate the error.

  m = 8, we have $(8 + r + 1) <= 2^r$ then We choose r = 4 Parity bit at 1, 2, 4, 8

Char K $75_{10} = 01001011_2$

1 = 1	5 = 1 + 4	9 = 1 + 8
2 = 2	6 = 2 + 4	10 = 2 + 8
3 = 1 + 2	7 = 1 + 2 + 4	11 = 1 + 2 + 8
4 = 4	8 = 8	12 = 4 + 8

We have the following code word as a result:

0	1	0	0	**1**	1	0	1	**0**	1	**1**	**0**
12	11	10	9	8	7	6	5	4	3	2	1

Parity b1 = b3 + b5 + b7 + b9+ b11 = 1 + 1 + 1 + 0 + 1 = **0**
Parity b2 = b3 + b6 + b7 + b10 + b11 = 1 + 0 + 1 + 0 + 1 = **1**
Parity b4 = b5 + b6 + b7 + b12 = 1 + 0 + 1 + 0 = **0**
Parity b8 = b9 + b10 + b11 + b12 = 0 + 0 + 1 + 0 = **1**

Let's introduce an error in bit position b9, resulting in the code word:

0	1	0	***1***	**1**	1	0	1	**0**	1	**1**	**0**
12	11	10	9	8	7	6	5	4	3	2	1

Parity b1 = b3 + b5 + b7 + b9+ b11 = 1 + 1 + 1 + 1 + 1 = **1** (Error, should be **0**) Parity b2 = b3 + b6 +
b7 + b10 + b11 = 1 + 0 + 1 + 0 + 1 = **1** (OK)
Parity b4 = b5 + b6 + b7 + b12 = 1 + 0 + 1 + 0 = **0** (OK)
Parity b8 = b9 + b10 + b11 + b12 = 1 + 0 + 1 + 0 = **0** (Error, should be **1**)

We found that parity bits 1 and 8 produced an error, and **1 + 8 = 9**, which in exactly where the error occurred.

Reed-Soloman:

- If we expect errors to occur in **blocks**, it stands to reason that we should use an error- correcting code that operates at a block level, as opposed to a Hamming code, which operates at the **bit** level.
- A Reed-Soloman (RS) code can be thought of as a CRC that operates over entire characters instead of only a few bits.
- RS codes, like CRCs, are systematic: The **parity bytes** are append to a block of information bytes.
- RS (n, k) code are defined using the following parameters:
  - s = The number of bits in a character (or "symbol").
  - k = The number of s-bit characters comprising the data block.
  - n = The number of bits in the code word.
- RS (n, k) can correct (n-k)/2 errors in the k information bytes.
- Reed-Soloman error-correction algorithms lend themselves well to implementation in computer **hardware**.

COMPUTER ARITHMETIC

Introduction:

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.

Addition and Subtraction :

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)

Addition and Subtraction of Signed-Magnitude Numbers

Computer Arithmetic 2 *Addition and Subtraction*

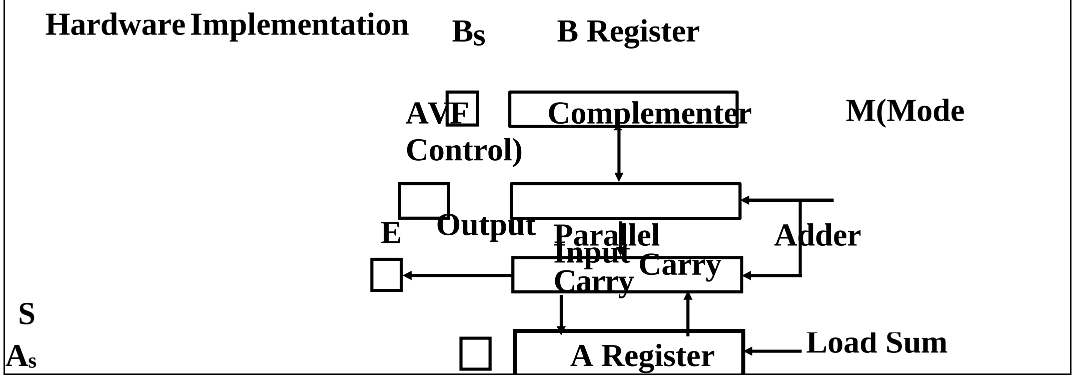
SIGNED MAGNITUDEADDITION AND SUBTRACTION

Addition: A + B ; A: Augend; B: Addend

Subtraction: A - B: A: Minuend; B: Subtrahend

Operation	Add Magnitude	Subtract Magnitude 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 - A)	+(A - B)
(+A) - (- B)	+(A + B)			
(- A) - (+B)	- (A + B)			
(- A) - (- B)		- (A - B)	+(B - A)	+(A - B)

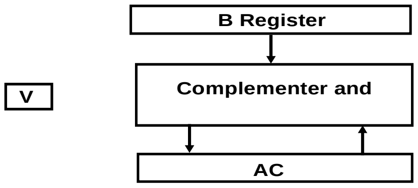
Hardware Implementation



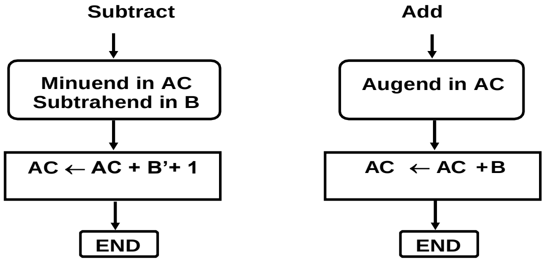
Computer Arithmetic 3 *Addition and Subtraction*

SIGNED 2'S COMPLEMENT ADDITION AND SUBTRACTION

Hardware



Algorithm



Algorithm:

- The flowchart is shown in Figure 7.1. The two signs A, and B, are compared by an exclusive-OR gate.

If the output of the gate is 0 the signs are identical; If it is 1, the signs are different.

- For an add operation, identical signs dictate that the magnitudes be added. For a subtract operation, different signs dictate that the magnitudes be added.

- The magnitudes are added with a microoperation EA A-+ B, where EA is a register that combines E and A. The carry in E after the addition constitutes an overflow if it is equal to 1. The value of E is transferred into the add-overflow flip-flop AVF.

- The two magnitudes are subtracted if the signs are different for an add operation or identical for a subtract operation. The magnitudes are subtracted by adding A to the 2's complemented B. No overflow can occur if the numbers are subtracted so AVF is cleared to 0.

- 1 in E indicates that A >= B and the number in A is the correct result. If this numbs is zero, the sign A must be made positive to avoid a negative zero.

- 0 in E indicates that A < B. For this case it is necessary to take the 2's complement of the value in A. The operation can be done with one microoperation A A' +1.

- However, we assume that the A register has circuits for microoperations complement and increment, so the 2's complement is obtained from these two microoperations.

- In other paths of the flowchart, the sign of the result is the same as the sign of A. so no change in A is required. However, when A < B, the sign of the result is the complement of the original sign of A. It is then necessary to complement A, to obtain the correct sign.

- The final result is found in register A and its sign in As. The value in AVF provides an overflow indication. The final value of E is immaterial.

- Figure 7.2 shows a block diagram of the hardware for implementing the addition and subtraction operations.

It consists of registers A and B and sign flip-flops As and Bs. Subtraction is done by adding A to the 2's complement of B.

- The output carry is transferred to flip-flop E , where it can be checked to determine the relative magnitudes of two numbers.

- The add-overflow flip-flop *AVF* holds the overflow bit when A and B are added.

- The A register provides other microoperations that may be needed when we specify the sequence of steps in the algorithm.

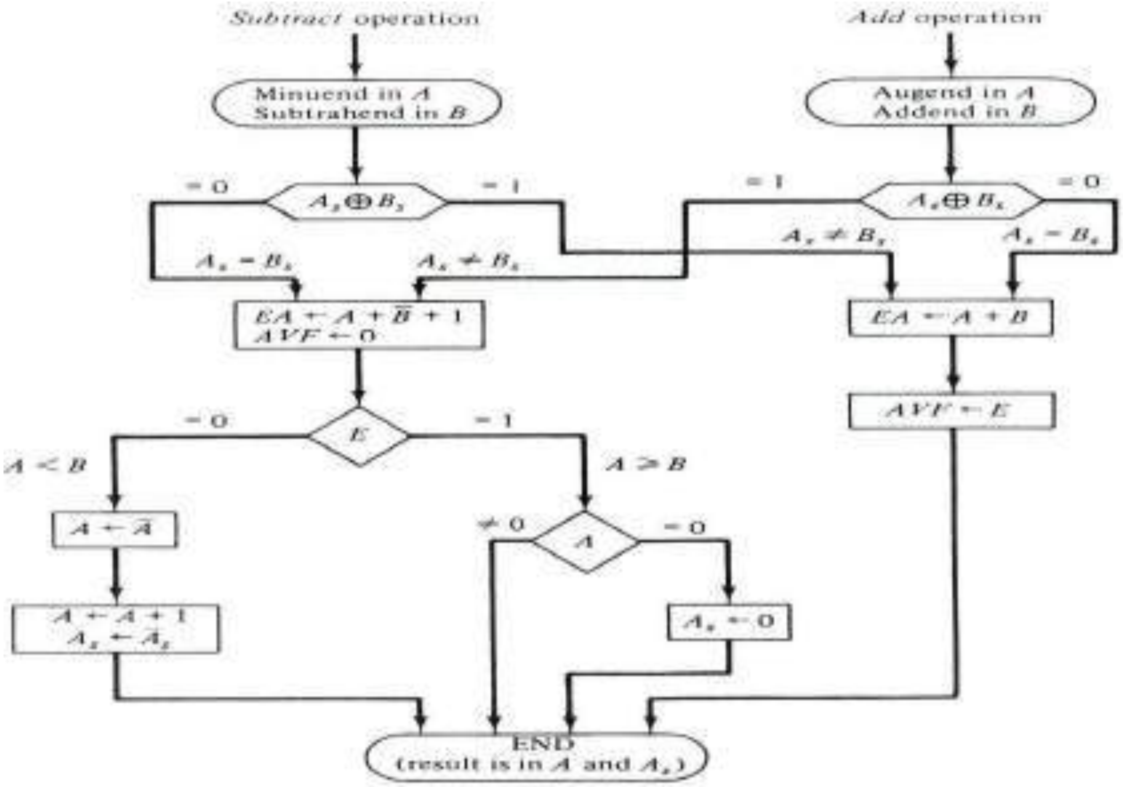
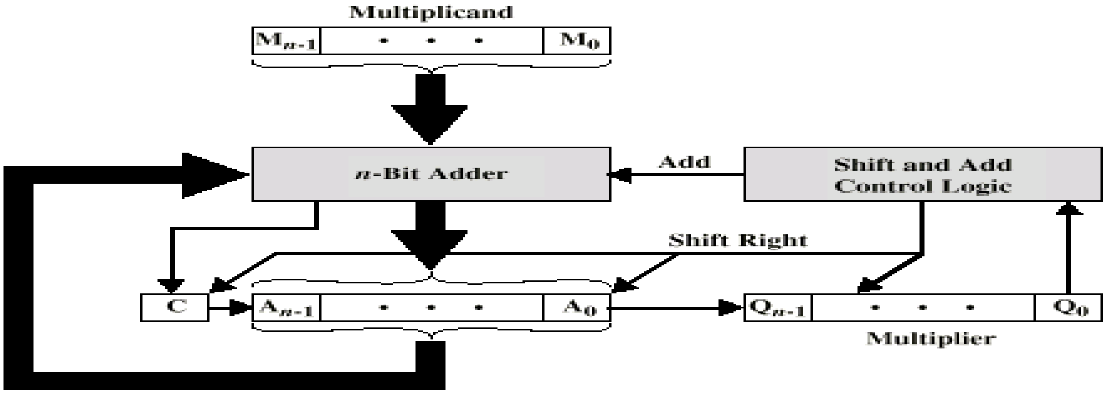


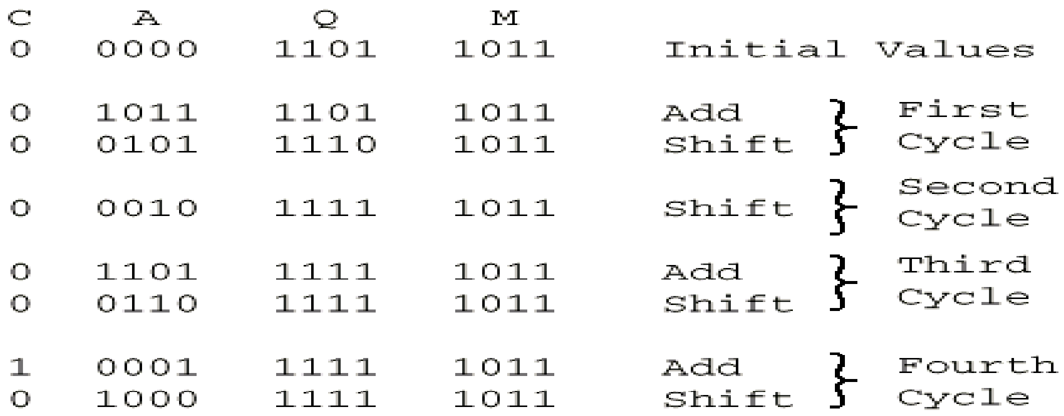
Figure 10-2 Flowchart for add and subtract operations.

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.





C	A	Q	M		
0	0000	1101	1011	Initial Values	
0	1011	1101	1011	Add	First Cycle
0	0101	1110	1011	Shift	
0	0010	1111	1011	Shift	Second Cycle
0	1101	1111	1011	Add	Third Cycle
0	0110	1111	1011	Shift	
1	0001	1111	1011	Add	Fourth Cycle
0	1000	1111	1011	Shift	

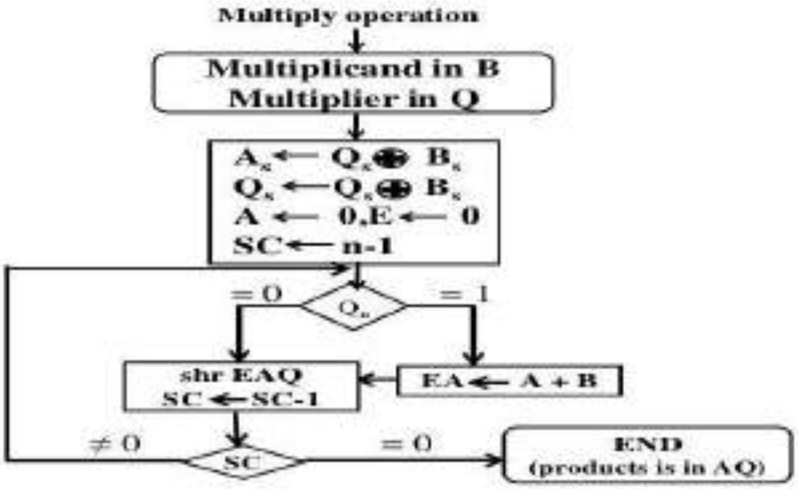


Figure: Flowchart for multiply operation.

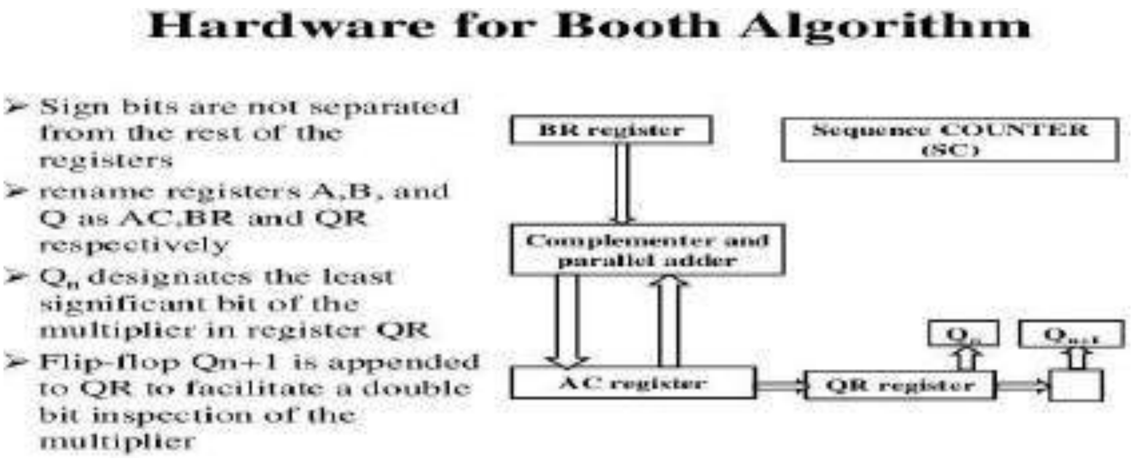
Booth's algorithm :

- Booth algorithm gives a procedure for multiplying binary integers in signed- 2's complement representation.

  It operates on the fact that strings of 0's in the multiplier require no addition but just shifting, and a string of 1’s in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} – 2^m$.

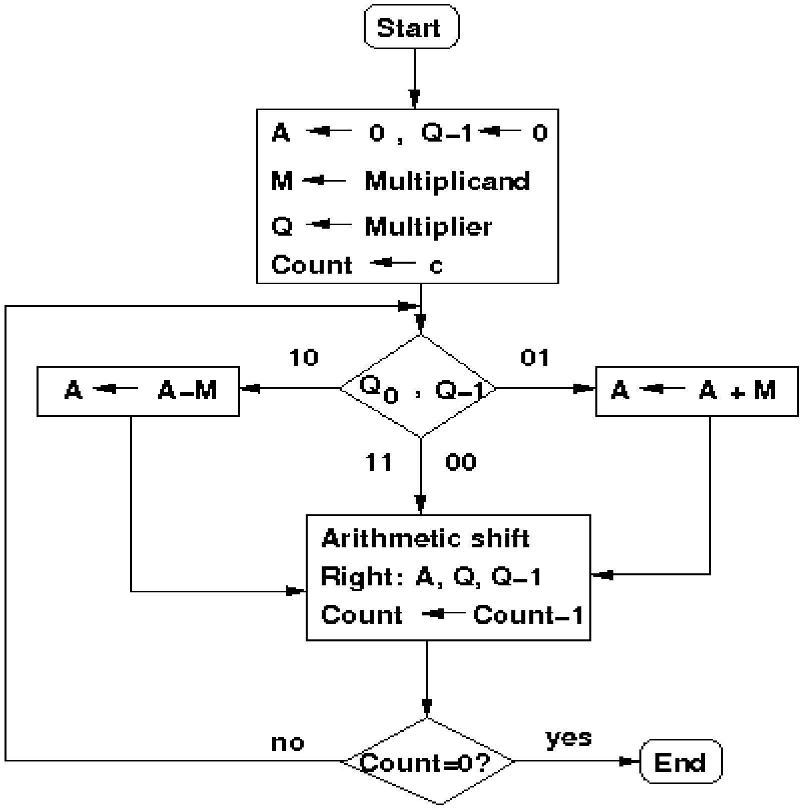
- For example, the binary number 001110 (+14) has a string 1’s from 2^3 to 2^1 (k=3, m=1). The number can be represented as $2^{k+1} – 2^m$. = $2^4 – 2^1$ = 16 – 2 = 14. Therefore, the multiplication M X 14, where M is the multiplicand and 14 the multiplier, can be done as M X 2^4 – M X 2^1.

- Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.



Hardware for Booth Algorithm

- Sign bits are not separated from the rest of the registers
- rename registers A,B, and Q as AC,BR and QR respectively
- Q_n designates the least significant bit of the multiplier in register QR
- Flip-flop Qn+1 is appended to QR to facilitate a double bit inspection of the multiplier



- As in all multiplication schemes, booth algorithm requires examination of the multiplier bits and shifting of partial product.

- Prior to the shifting, the multiplicand may be added to the partial product, subtracted from the partial, or left unchanged according to the following rules:

1. The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1’s in the multiplier.

2. The multiplicand is added to the partial product upon encountering the first 0 in a string of 0’s in the multiplier.

3. The partial product does not change when multiplier bit is identical to the previous multiplier bit.

- The algorithm works for positive or negative multipliers in 2’s complement representation.

- This is because a negative multiplier ends with a string of 1’s and the last operation will be a subtraction of the appropriate weight.

- The two bits of the multiplier in Qn and Qn+1 are inspected.

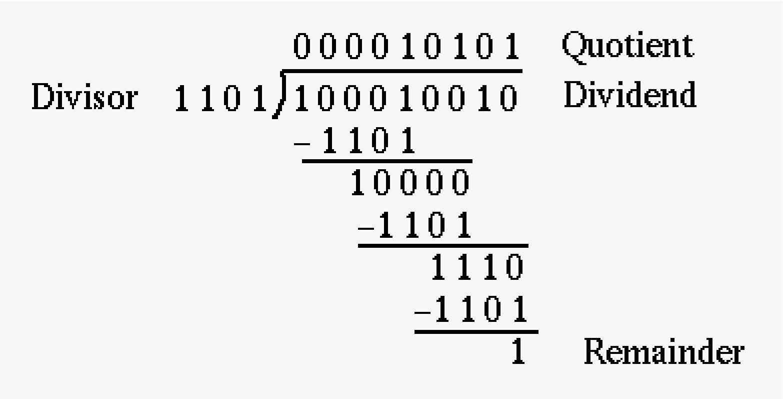
- If the two bits are equal to 10, it means that the first 1 in a string of 1 's has been encountered. This requires a subtraction of the multiplicand from the partial product in AC.

- If the two bits are equal to 01, it means that the first 0 in a string of 0's has been encountered. This requires the addition of the multiplicand to the partial product in AC.

- When the two bits are equal, the partial product does not change.

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



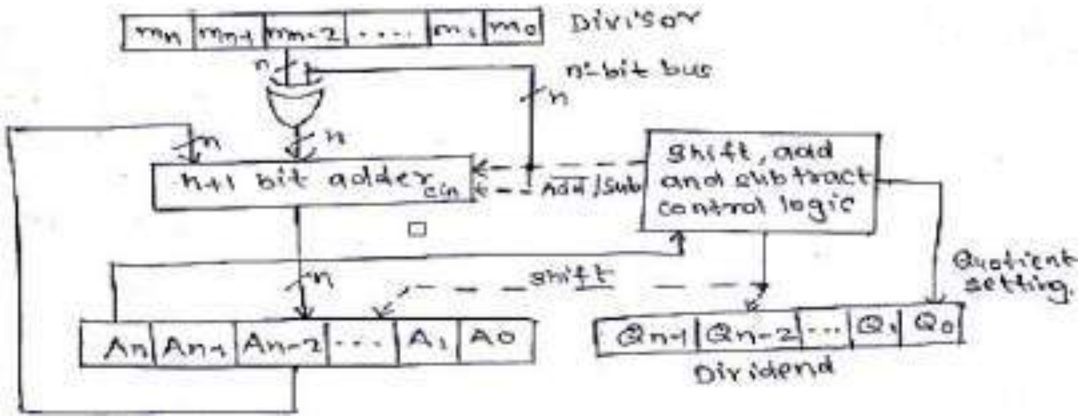
The devisor is compared with the five most significant bits of the dividend. Since the 5-bit number is smaller than B, we again repeat the same process. Now the 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend. Now we shift the divisor once 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.

Hardware Implementation for Signed-Magnitude Data

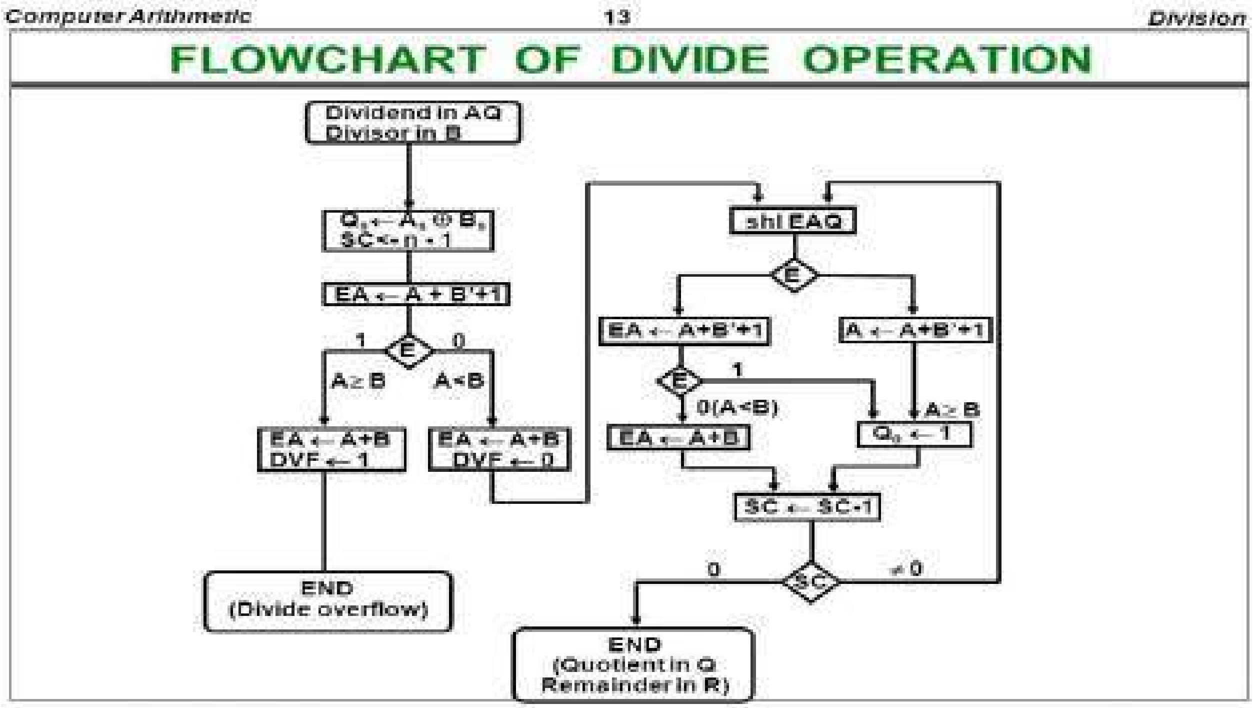
In hardware implementation for signed-magnitude data in a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, two dividends, or partial remainders, are shifted to the left, thus leaving the two numbers in the required relative position. Subtraction 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. E

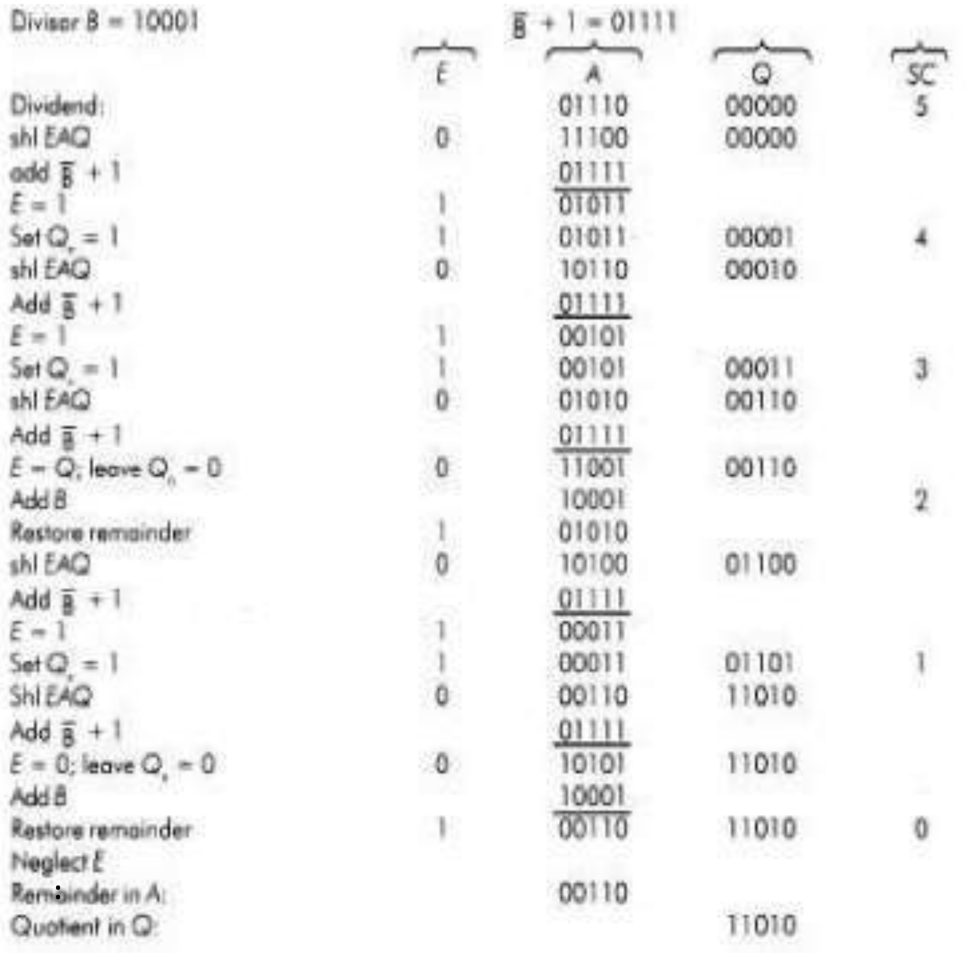


Hardware Implementation for Signed-Magnitude Data

Algorithm:



Example of Binary Division with Digital Hardware



Divisor B = 10001 $\bar{B} + 1 = 01111$

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

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.

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

$m \text{ x } r^{e}$

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

$.53725 \text{ x } 10^{3}$

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 + $(24^{7} – 1)$, which is approximately + 101^{4}. 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

$+ (1 – 2^{-35}) \text{ x } 2^{2047}$

This number is derived from a fraction that contains 35 1's, an exponent of 11 bits (excluding its sign), and because $2^{11}–1 = 2047$. The largest number that can be accommodated is approximately 10^{615}. 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 $(2^{35} –1)$. This is approximately equal to 10^{10}, 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:

$.5372400 \times 10^{2}$

$+ .1580000 \times 10^{-1}$

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.

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.

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.

FLOATING POINT ARITHMETIC OPERATIONS

F = m x r^e

where m: Mantissa

r: Radix

Registers for Floating Point Arithmetic

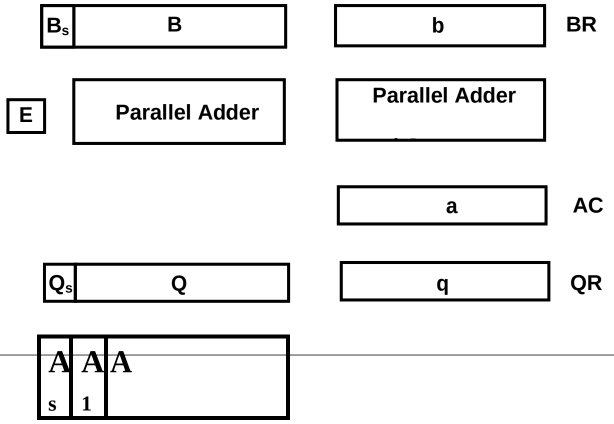


Figure 4.13: Registers for Floating Point arithmetic operations

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 are 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 a 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.

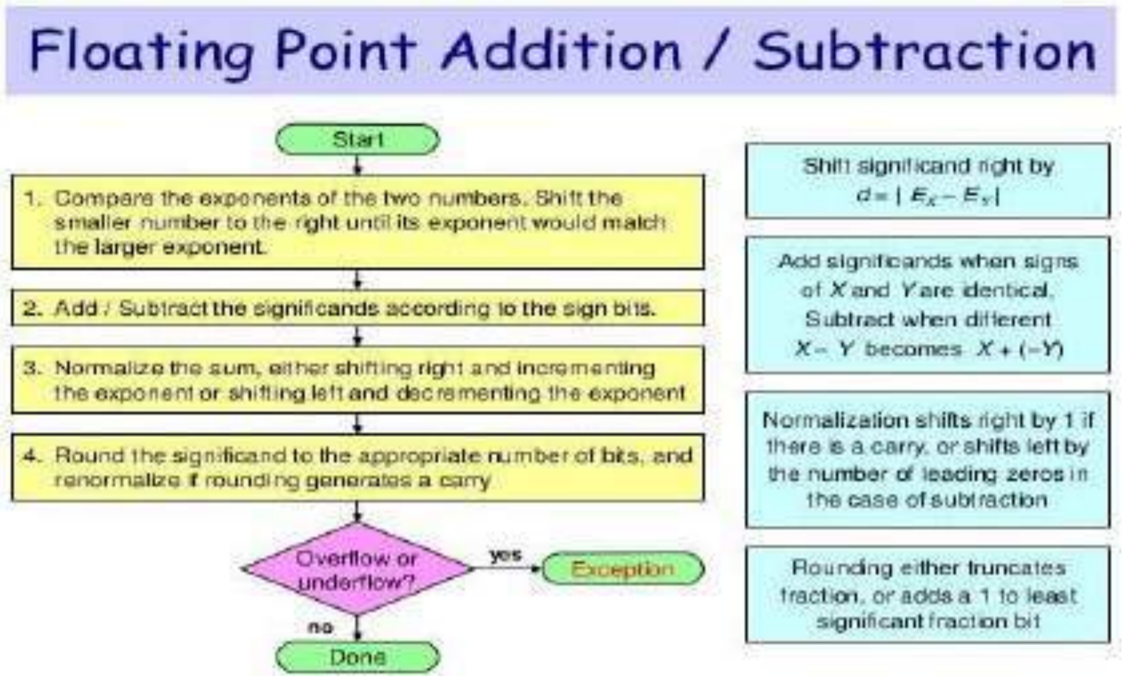
Addition and Subtraction of Floating Point Numbers:

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.

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.



Floating Point Addition / Subtraction

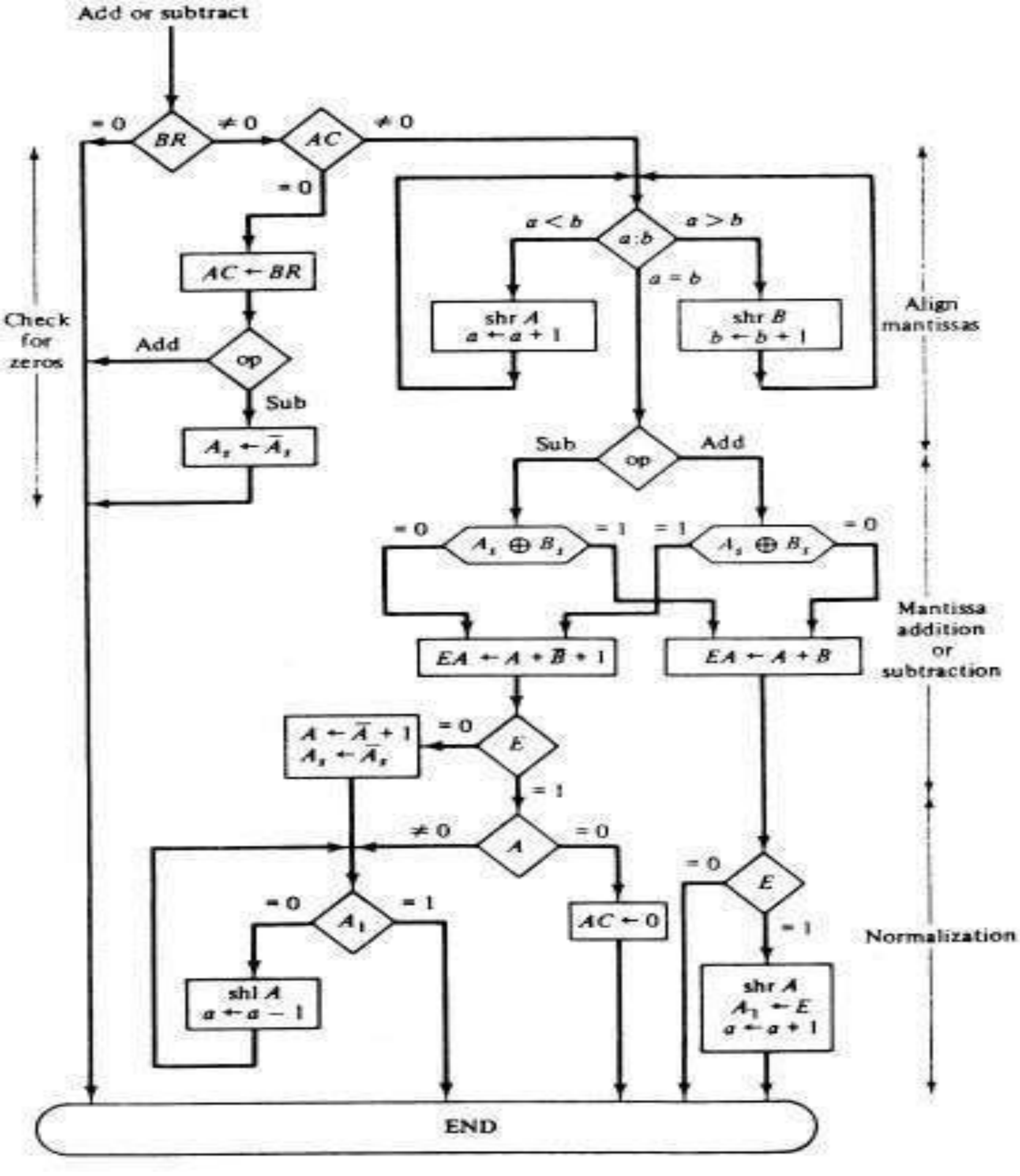
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent.
2. Add / Subtract the significands according to the sign bits.
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
4. Round the significand to the appropriate number of bits, and renormalize if rounding generates a carry

Shift significand right by $d = | E_X - E_Y |$

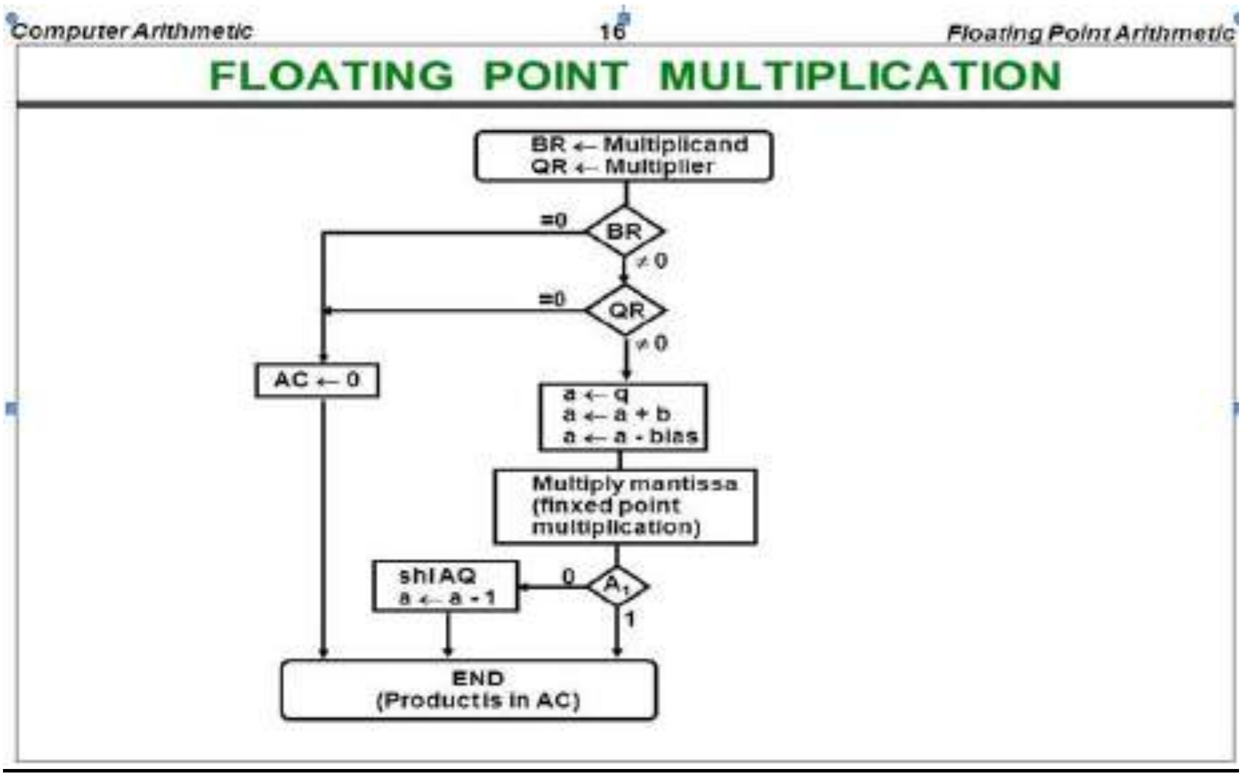
Add significands when signs of X and Y are identical, Subtract when different $X - Y$ becomes $X + (-Y)$

Normalization shifts right by 1 if there is a carry, or shifts left by the number of leading zeros in the case of subtraction

Rounding either truncates fraction, or adds a 1 to least significant fraction bit

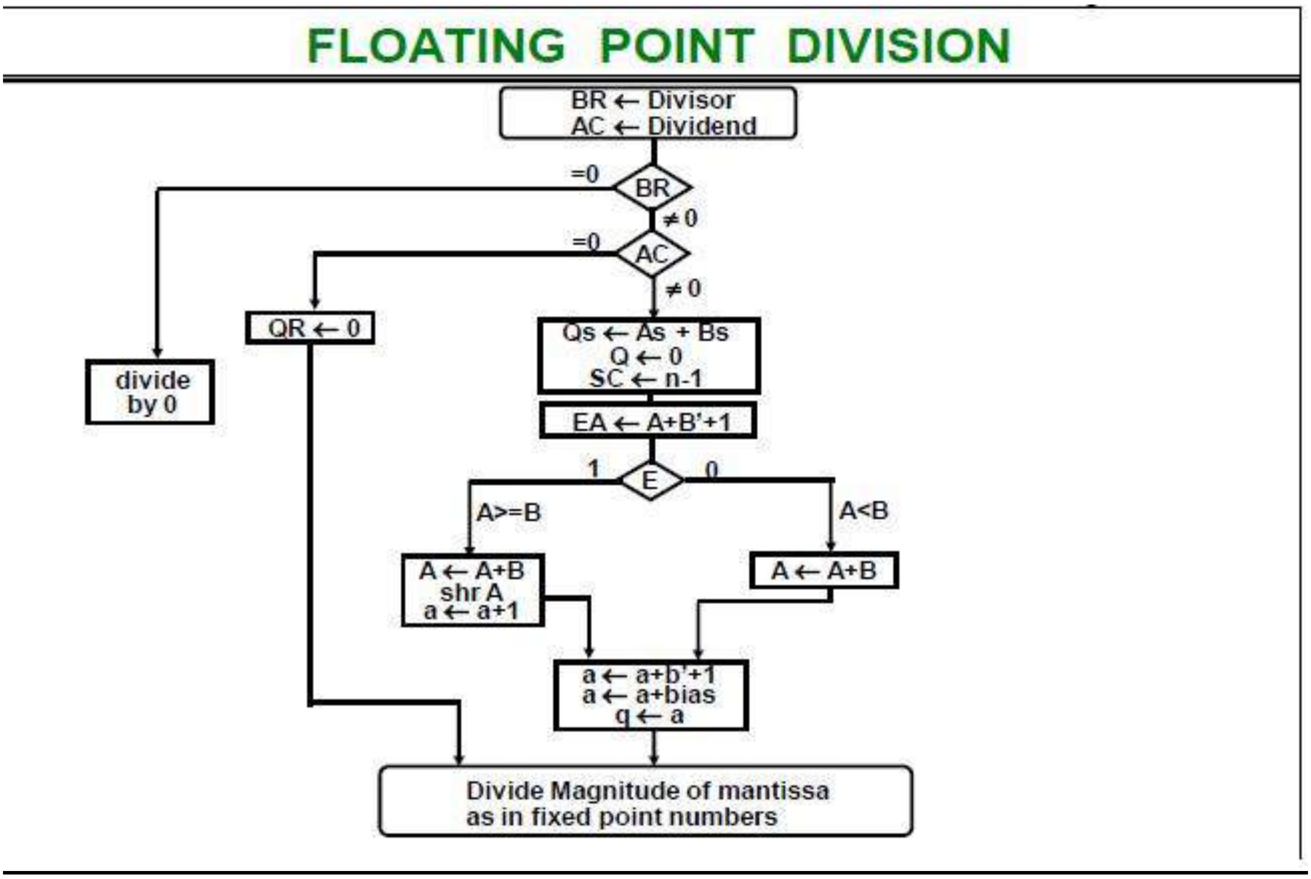


Algorithm for Floating Point Addition and Subtraction



Computer Arithmetic 16 Floating Point Arithmetic

FLOATING POINT MULTIPLICATION



FLOATING POINT DIVISION

COMPUTER ORGANIZATION – UNIT-2

UNIT II

Register Transfer Language and Micro-operations: Register Transfer language. Register Transfer Bus and Memory Transfers, Arithmetic Micro operations, Logic Micro Operations, Shift Micro Operations, Arithmetic Logic Shift Unit.

Basic Computer Organization and Design: Instruction Codes, Computer Register, Computer Instructions, Instruction Cycle, Memory – Reference Instructions. Input –Output and Interrupt, Complete Computer Description.

---

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 in the registers.
- The control that initiates the sequence of micro operations.

→Use symbols, rather than words, to specify the sequence of micro-operations.

Register transfer language:

→The symbolic notation used to describe the micro-operation transfers among registers is called a **register transfer language**. The term "register transfer" implies the availability of hardware logic circuits that can perform a stated micro-operation and transfer the result of the operation to the same or another register. The word "language" is borrowed from programmers, who apply this term to programming languages.

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

→A ***register transfer language*** is a system for expressing in symbolic form the micro-operation sequences among the registers of a digital module. It is a convenient tool for describing the internal organization of digital computers in concise and precise manner. It can also be used to facilitate the design process of digital systems.

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

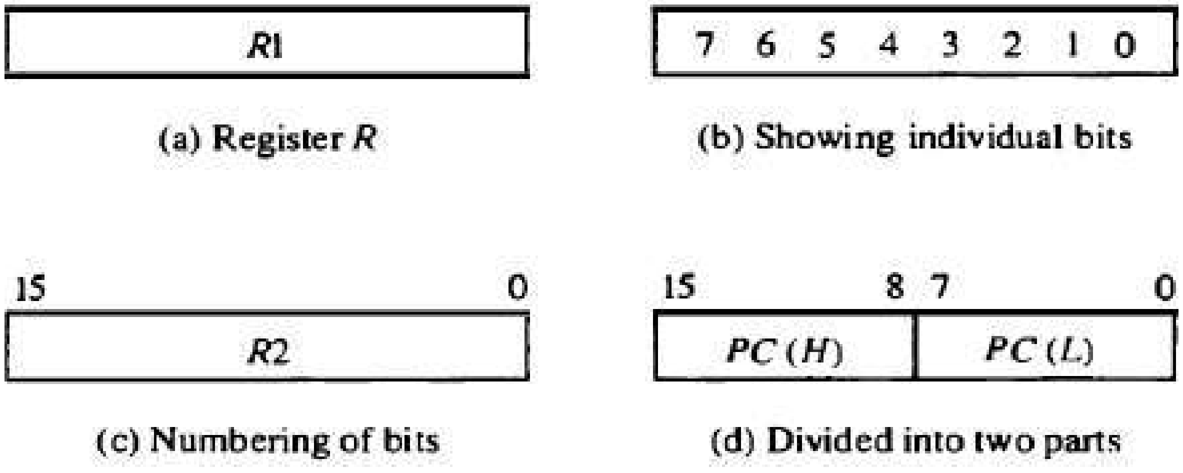
2. Register Transfer:

Registers:

→A register is a group of flip-flops with each flip-flop capable of storing one bit of information.

→Designate computer **registers** by capital letters to denote its function. The register that holds an address for the memory unit is called **Memory address register MAR.**. The *program counter register* is called **PC**. *Instruction register* **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, starting from 0 in the right most position and increasing the numbers towards the left.

Figure 4-1 Block diagram of register.



(a) Register *R*

(b) Showing individual bits

(c) Numbering of bits

(d) Divided into two parts

Figure: shows the representation of registers in block diagram form.

→The most common way to represent a register is by a rectangular box with the name of the register inside, as in **figure (a).** The individual bits can be distinguished as in **figure (b).** The numbering of bits in a 16-bit register can be marked on top of the box as shown in **figure (c).** A 16-bit register is partitioned into two parts in **figure (d).**

→Bits 0 through 7 are assigned the symbol **L (for low byte)** and bits 8 through 15 are assigned the symbol **H (for high bit).** The name of the 16-bit register is PC. The symbol **PC (0-7) or PC (L)** refers to the low order byte and **PC (8-15) or PC (H)** to high order byte.

Register transfer:

→Register Transfer is defined as copying the content of one register to another.

→ Information transfer from one register to another is designated in symbolic form by means of a replacement operator. The statement 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 signal generated in the control section. It is sometimes convenient to separate the control variables from the register transfer operation by specifying a **control function.**

→A **control function** is a Boolean variable that is equal to 1 or 0. The control function is included in the statement as follows:

P: R2 ← R1

The control condition is terminated with a colon. It symbolizes the requirement that the transfer operation be executed by the hardware only **if P = 1.**

→Every statement written in a register transfer notation implies a hardware construction for implementing the transfer.

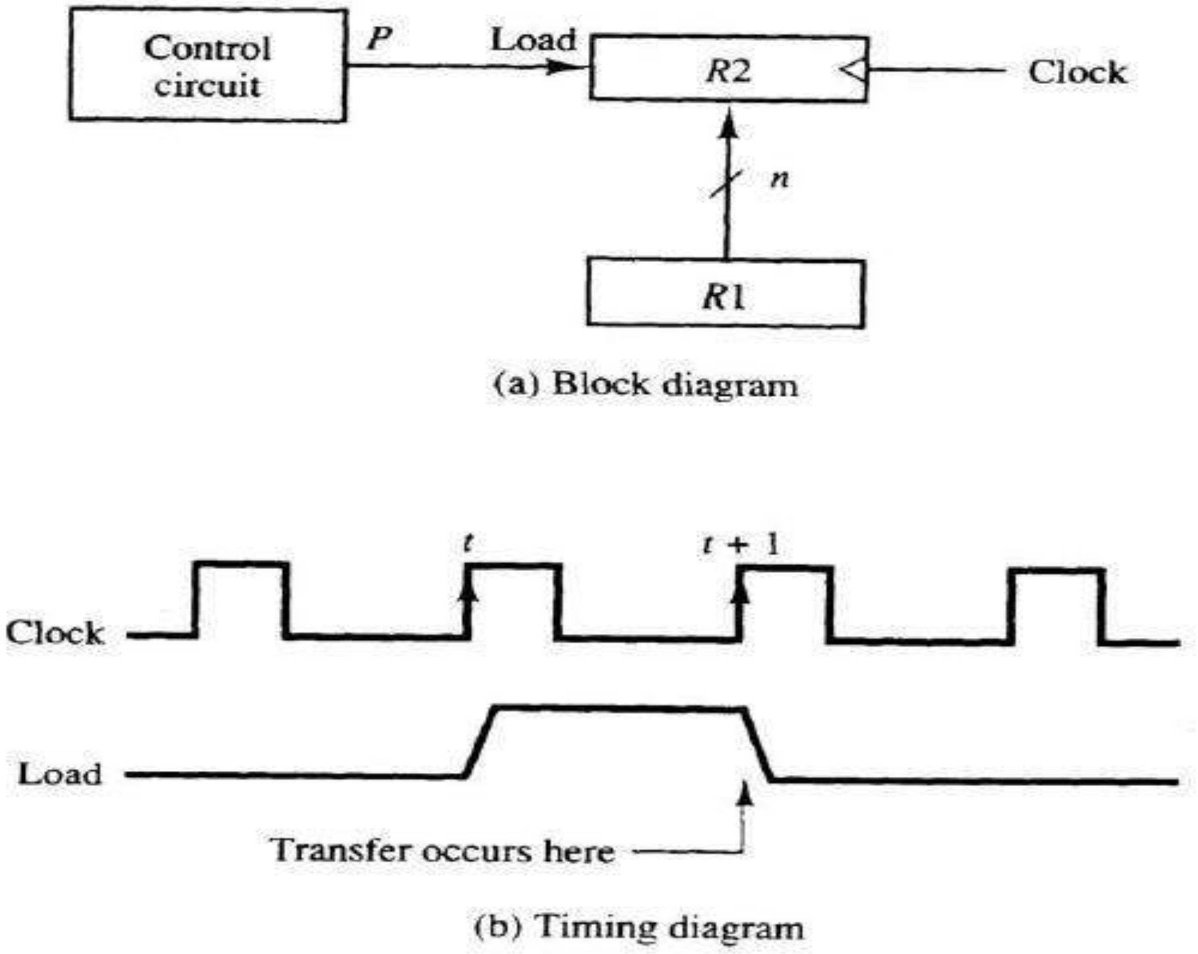
→**Figure 4-2 shows the block diagram that depicts the transfer from R1 to R2.** The n outputs of register R1 are connected to the n inputs of register R2. Register R2 has a load input activated by the control variable P. It is assumed that the control variable is synchronized with the same clock as the one applied to the register.

→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

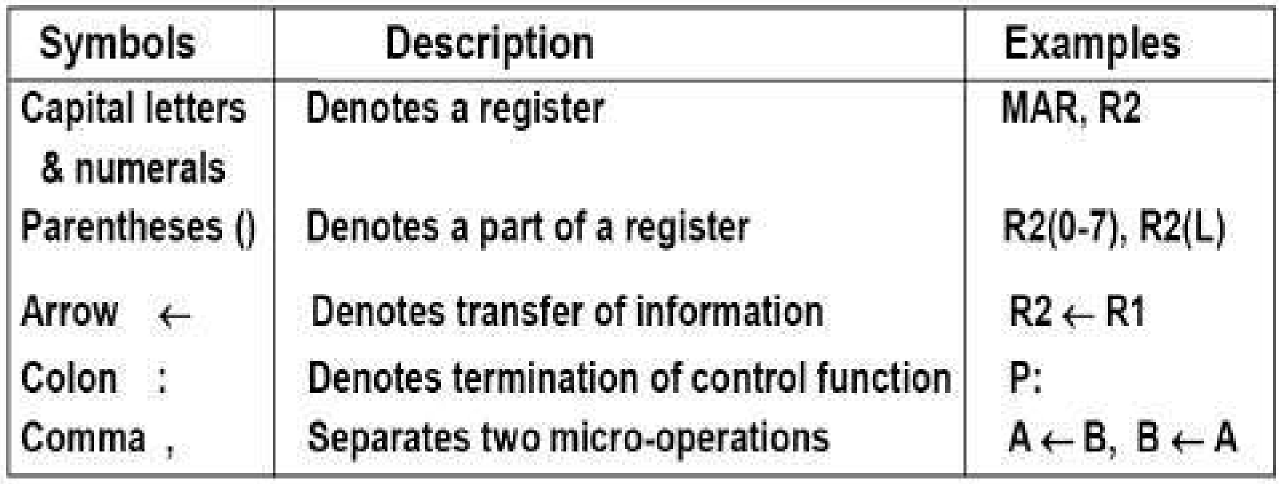
Figure 4-2 Transfer from $R1$ to $R2$ when $P = 1$.



(a) Block diagram

(b) Timing diagram

The basic symbols for register transfer are shown in table



Symbols	Description	Examples
Capital letters & numerals	Denotes a register	MAR, R2
Parentheses ()	Denotes a part of a register	R2(0-7), R2(L)
Arrow ←	Denotes transfer of information	R2 ← R1
Colon :	Denotes termination of control function	P:
Comma ,	Separates two micro-operations	A ← B, B ← A

Table: Basic Symbols for Register Transfers.

3. Bus and Memory Transfers:

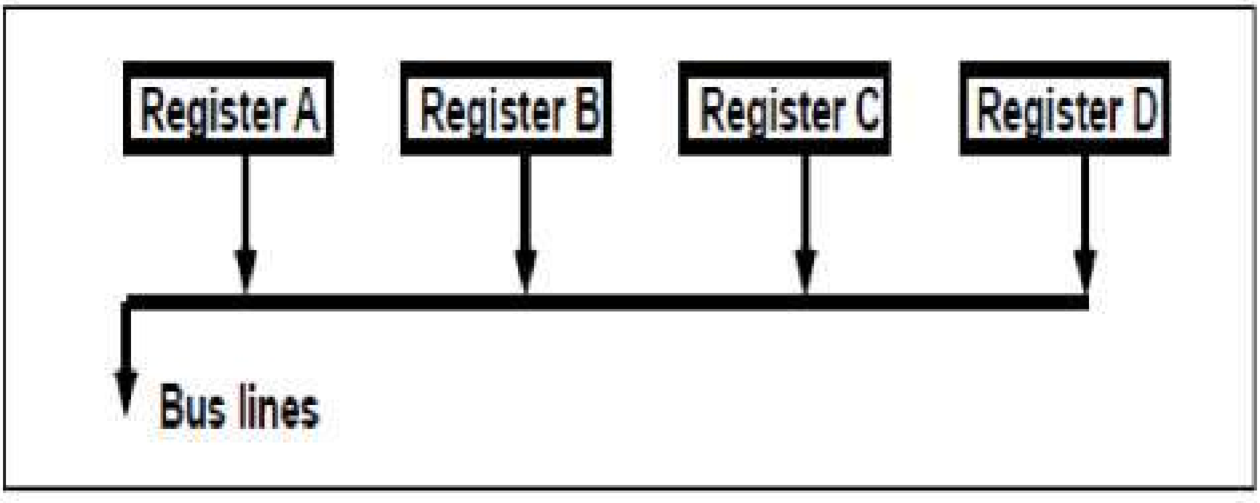
→Rather than connecting wires between all registers, a common bus is used.

→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.

→**The construction of a *bus system for four registers* is shown in Figure 4-3.**

→Each register has four bits, numbered 0 through 3. The bus consists of four 4 X 1 multiplexers each having four data inputs, 0 through 3, and two selection inputs, **S1 and S0.**

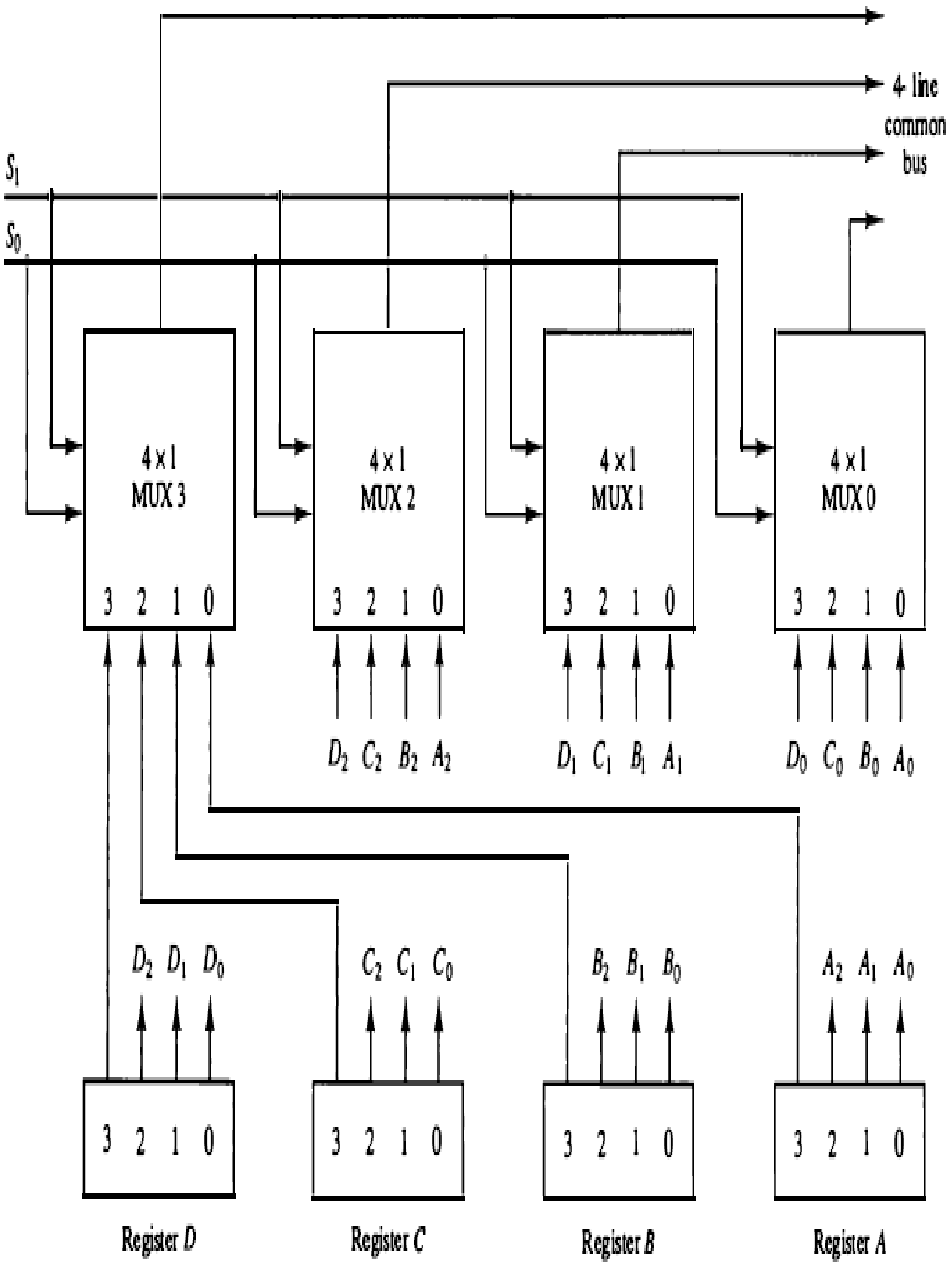
→**MUX 0** multiplexes the four 0 bits of the registers, **MUX 1** multiplexes the four 1 bits of the registers, and similarly for the other two bits.



→The two selection lines S_1 and S_0 are connected to selection inputs of all four multiplexers. The selection lines choose the four bits of one register and transfer them into the four-line common bus.

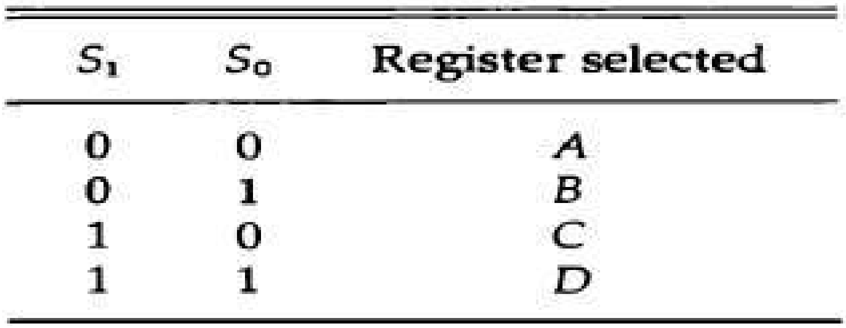
→When **S_1S_0 =00**, the **0** data inputs of all four multiplexers are selected and applied to the outputs that from the bus. This causes the bus lines to receive the content of **register A** since the outputs of this register are connected to the **0** data inputs of the multiplexers. Similarly, register B is selected if **S_1S_0 =01**, and so on.

Figure 4-3 Bus system for four registers.



→**Table:** shows the register that is selected by the bus for each of the four possible binary values of the selection lines**.**

TABLE 4-2 Function Table for Bus of Fig. 4-3



S_1	S_0	Register selected
0	0	A
0	1	B
1	0	C
1	1	D

→In general, a bus 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.

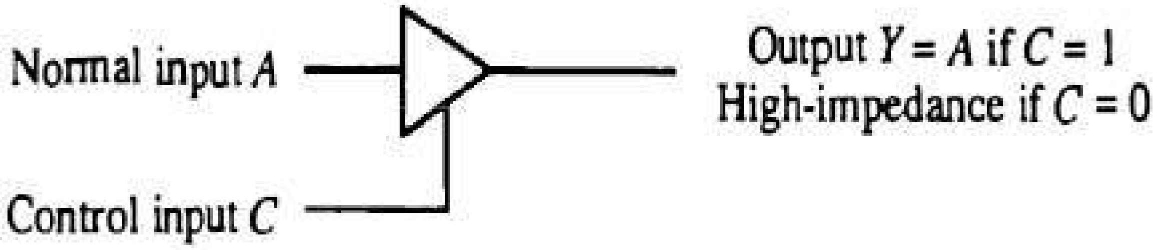
Three State Bus Buffers:

→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 logic significance.

→The graphic symbol of a **three-state buffer** gate is shown in **figure.4-4.**

Figure 4-4 Graphic symbols for three-state buffer.



→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.

→Decoders are used to ensure that no more than one control input is active at any given time.

→ The construction of a **bus system** with three-state buffers is demonstrated in **Figure 4-5.**

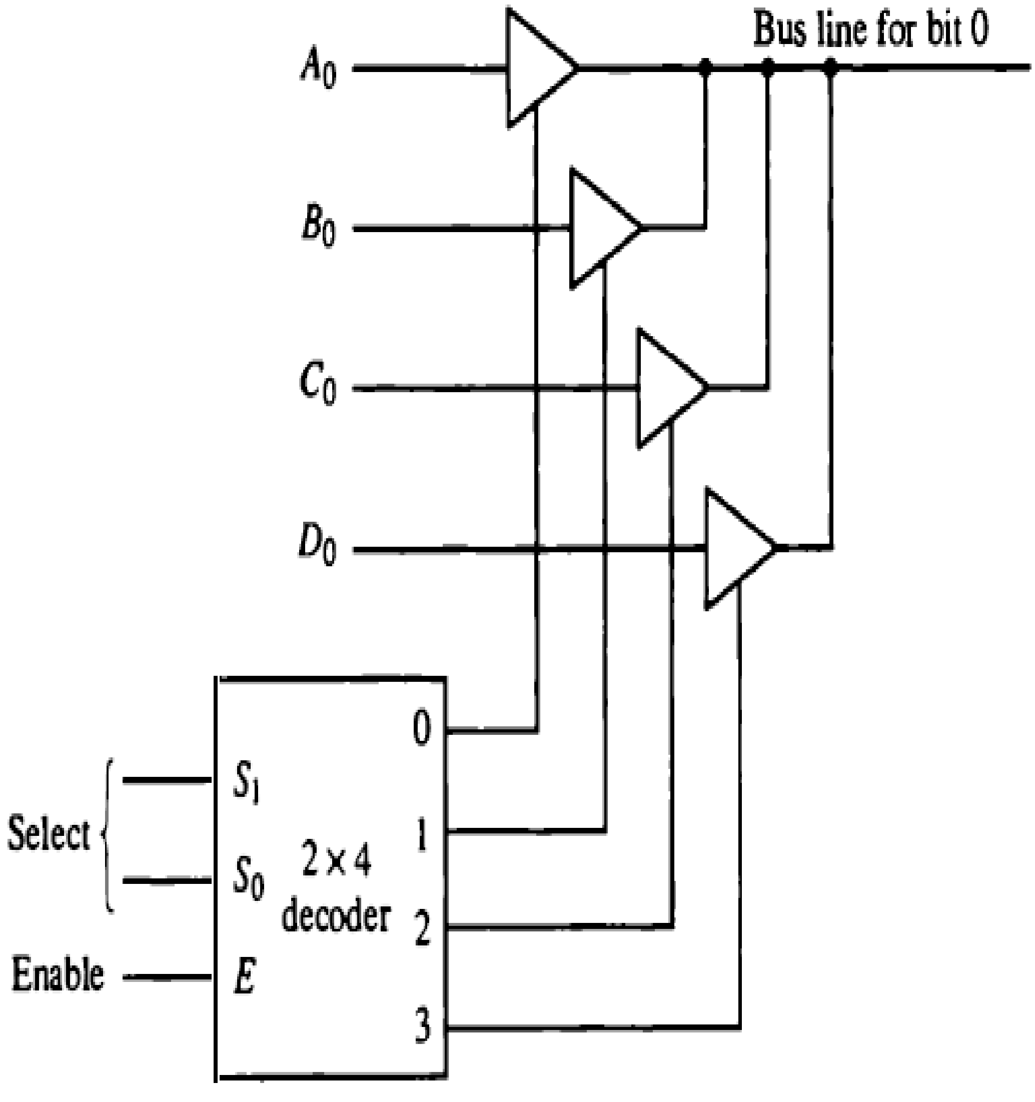


Figure 4-5 Bus line with three state-buffers.

→The outputs of four buffers are connected together to form a signal bus line. The control inputs to the buffers determine which of the four normal inputs will communicate with the bus line. No more than one buffer may be in the active state at any given time. The connected buffers must be controlled so that only one three state buffer has access to the bus line while all other buffers are maintained in a **high-impedance state.**

→One way to ensure that no more than one control input is active at any given time is to use a decoder, as shown in the diagram. When the **enable input of the decoder is 0,** all of its four outputs are 0, and the bus line is in a high impedance state because all four buffers are disabled. When the enable input is active, one of the three state buffers will be active, depending on the binary value in the select inputs of the decoder.

→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.

Memory Transfer:

→The transfer of information from a memory word to the outside environment is called a **read operation.**The transfer of new information to be stored into the memory is called a **write operation.**

→Designate a memory word by the letter M. It is necessary to specify the address of M when writing memory transfer operations.

→Consider a memory unit that receives the address from a register, called the ***address register.***

→The data are transferred to another register, called the **data register.**

→Designate the address register by AR and the data register by DR.

Read Operation:

→The ***read operation*** can be stated as:

Read: DR ← M[AR].

This causes a transfer of information into DR from the memory word M selected by the address in AR.

Write Operation:

→The **write operation** transfers the content of a data register to a memory word M selected by the address. Assume that the input data are in register R1 and the address is in AR.

The **write operation** can be stated as:

Write: M [AR] ← R1.

→This causes a transfer of information from R1 into the memory word M selected by the address in AR.

4. Arithmetic Micro Operation:

Micro-operation:

→A **micro operation** is an elementary operation performed with data stored in register. They are classified into:

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

Register transfer micro-operations:

→This type of micro-operation does not change the information content when the binary information moves from the source register to the destination register.

→Registers are designated by capital letters, sometimes followed by numbers (e.g., A, R13, IR). Often the names indicate function:

MAR	**memory address register**
PC	**program counter**
IR	**instruction register**

→Information transfer from one register to another is described in symbolic form by replacement operator. The statement "R2 ←R1" denotesa transfer of the content of the R1 into resister R2.

Control Function:

→Often actions need to only occur if a certain condition is true. In digital systems, this is often done via a control signal, called a control function.

Example: **P: R2← R1 i.e. if (P = 1) then (R2 ← R1)**

→Which means "if **P = 1**, then load the contents of register R1 into register R2". If two or more operations are to occur simultaneously, they are separated with commas.

Example: **P: R3 ←R5, MAR ← IR**

Arithmetic micro-operations:

→Basic arithmetic micro-operations are **addition, subtraction, increment, decrement and Shift.**

→The additional arithmetic micro-operations are **Add with carry, Subtract with borrow and Transfer/Load.**

→Example of ***addition*****: R3 ← R1 +R2.**

→It states that the contents of register R1 are added the contents of register R2 and the sum transferred to register R3. Usually it is implemented using hardware full adders.

→***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 R1-R2.

R3 □□R1 + R2 +1 or (R1 – R2)

→The **increment and decrement** micro-operations are implemented with a combinational circuit or with a binary Up and Down Counter.

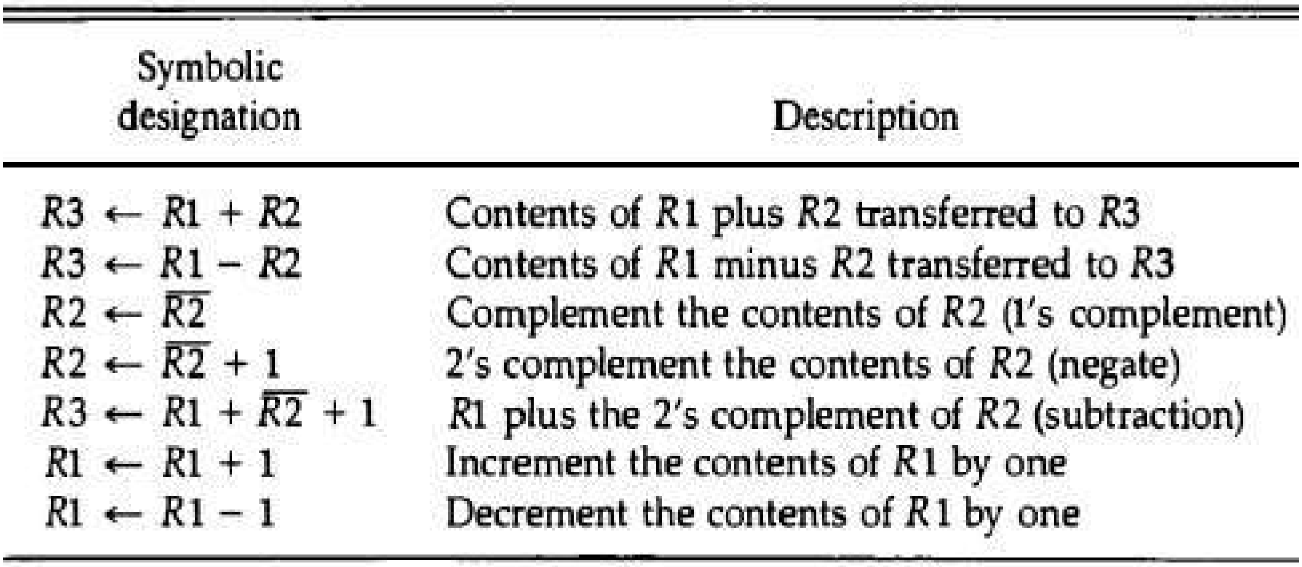
→Multiply and divide are not included as micro-operations.

→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).

→Summary of typical arithmetic micro-operations:

TABLE 4-3 Arithmetic Microoperations



Symbolic designation	Description
$R3 \leftarrow R1 + R2$	Contents of $R1$ plus $R2$ transferred to $R3$
$R3 \leftarrow R1 - R2$	Contents of $R1$ minus $R2$ transferred to $R3$
$R2 \leftarrow \overline{R2}$	Complement the contents of $R2$ (1's complement)
$R2 \leftarrow \overline{R2} + 1$	2's complement the contents of $R2$ (negate)
$R3 \leftarrow R1 + \overline{R2} + 1$	$R1$ plus the 2's complement of $R2$ (subtraction)
$R1 \leftarrow R1 + 1$	Increment the contents of $R1$ by one
$R1 \leftarrow R1 - 1$	Decrement the contents of $R1$ by one

Binary Adder (Ripple Carry Adder):

→To implement the add micro-operation with hardware, we need the resisters that hold the data and the digital component that performs the arithmetic addition. The digital circuit that generates the arithmetic sum of two binary numbers of any lengths is called Binary adder.

→The binary adder is constructed with the **full-adder circuit connected in cascade**, with the output carry from one full-adder connected to the input carry of the next full-adder.

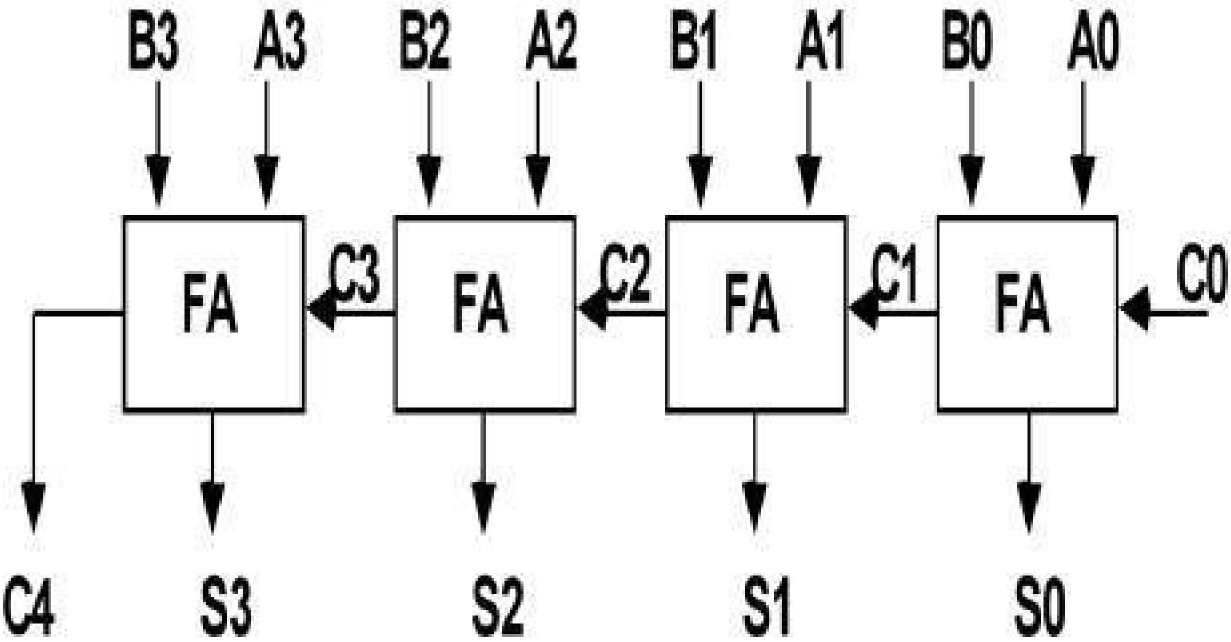
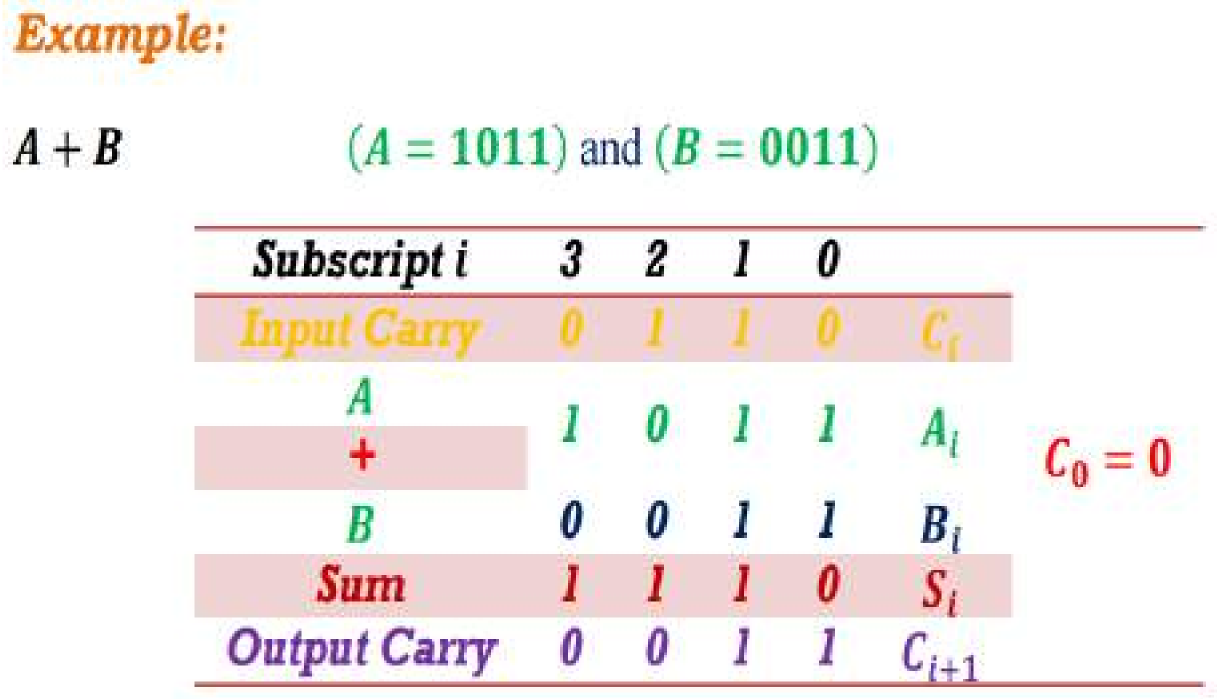


Figure 4-6 4-bit binary adder.

→**Figure 4-6** shows the interconnections of four full-adders (FA) to provide a 4-bit binary adder. The augend bits of A and the addend bits of B are designated by subscript numbers from right to left, with subscript 0 denoting the low-order bit. The carries are connected in a chain through the full-adders. The input carry to the binary adder is C_0 and the output carry is C_4. The S outputs of the full-adders generate the required sum bits.

→An n-bit binary adder requires n full-adders. The output carry from each full-adder is connected to the input carry of the next-high-order-full-adder. Inputs A and B come from two registers R1 and R2.



Example:

$A + B$ $(A = 1011)$ and $(B = 0011)$

Subscript i	3	2	1	0		
Input Carry	0	1	1	0	C_i	
A +	1	0	1	1	A_i	$C_0 = 0$
B	0	0	1	1	B_i	
Sum	1	1	1	0	S_i	
Output Carry	0	0	1	1	C_{i+1}	

Binary Adder-Subtractor:

→The subtraction A – B can be done by taking the 2's complement of B and adding to A. It means if we use the inverters to make 1’s complement of B (connecting each Bi to an inverter) and then add 1 to the least significant bit (by setting carry C_0 to 1) of binary adder, then we can make a **binary subtractor**.

→The addition and subtraction operations can be combined into one common circuit by including an **exclusive-OR** gate with each full adder.

→A 4-bit adder-subtractor circuit is shown in **Figure 4-7**. The mode input M controls the operation. When M=0 the circuit is an **adder** and when **M=1** the circuit becomes a **subtractor**. Each exclusive-OR gate receives input M and one of the inputs of B.

→When **M=0**, we have **$B \oplus 0= B$**. The full-adders receive the value of B, the input carry is 0, and circuit performs **A plus B.**

→When **M=1,** we have **$B \oplus 1 = B^1$** and **C_0=1**. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2’s complement of B. For unsigned numbers, this gives A - B if A ≥ B or the2's complement of (B - A) if A < B. For signed numbers, the result is A – B provided that there is no overflow.

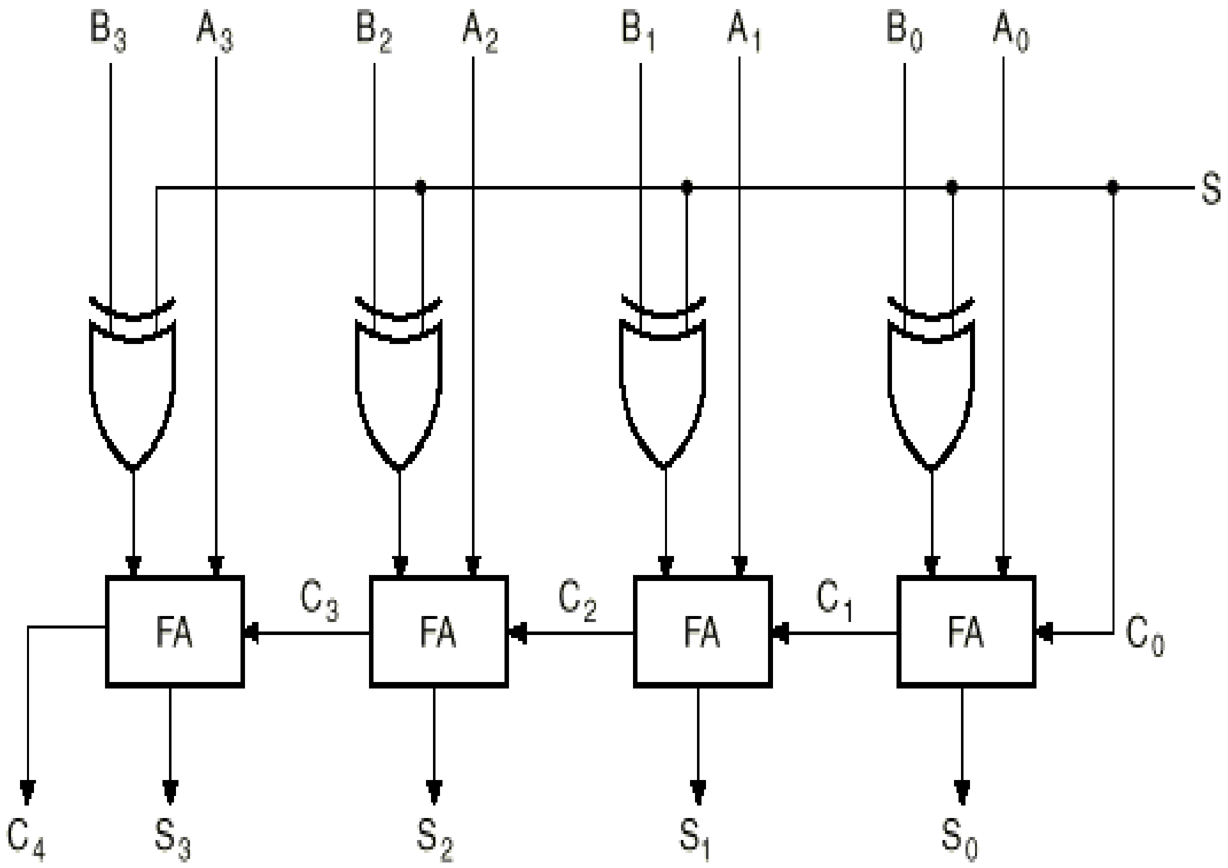
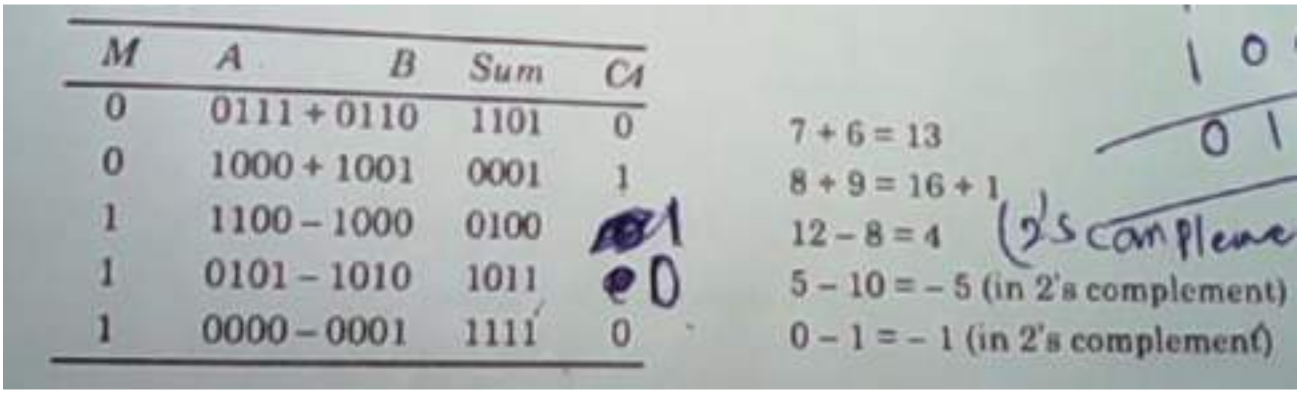


Figure 4-7 4-bit adder-subtractor.

Example:



M	A B	Sum	C4
0	0111 + 0110	1101	0
0	1000 + 1001	0001	1
1	1100 – 1000	0100	1
1	0101 – 1010	1011	0
1	0000 – 0001	1111	0

7 + 6 = 13

8 + 9 = 16 + 1

12 – 8 = 4 (2's compleme

5 – 10 = – 5 (in 2's complement)

0 – 1 = – 1 (in 2's complement)

1 0

0 1

Binary Incrementor:

→Increment micro-operation adds one to a number in a register. For example, if a 4-bit register has a binary value 0110, it will go to 0111 after it is incremented.

→The diagram of a 4-bit combinational circuit incrementer is shown in **Figure 4-8.** One of the inputs to the least significant **half-adder (HA)** is connected to logic-1 and the other input is connected to the least significant bit of the number to be incremented. The output carry from one-half adder is connected to one of the inputs of the next-higher-order half adder. The circuit receives the four bits from A_0 through A_3adds one to it, and generates the incremented output in **S_0 through S_3.** The output carry C_4 will be 1 only after incrementing binary 1111. This is also causes outputs **S_0 through S_3 to go to 0.**

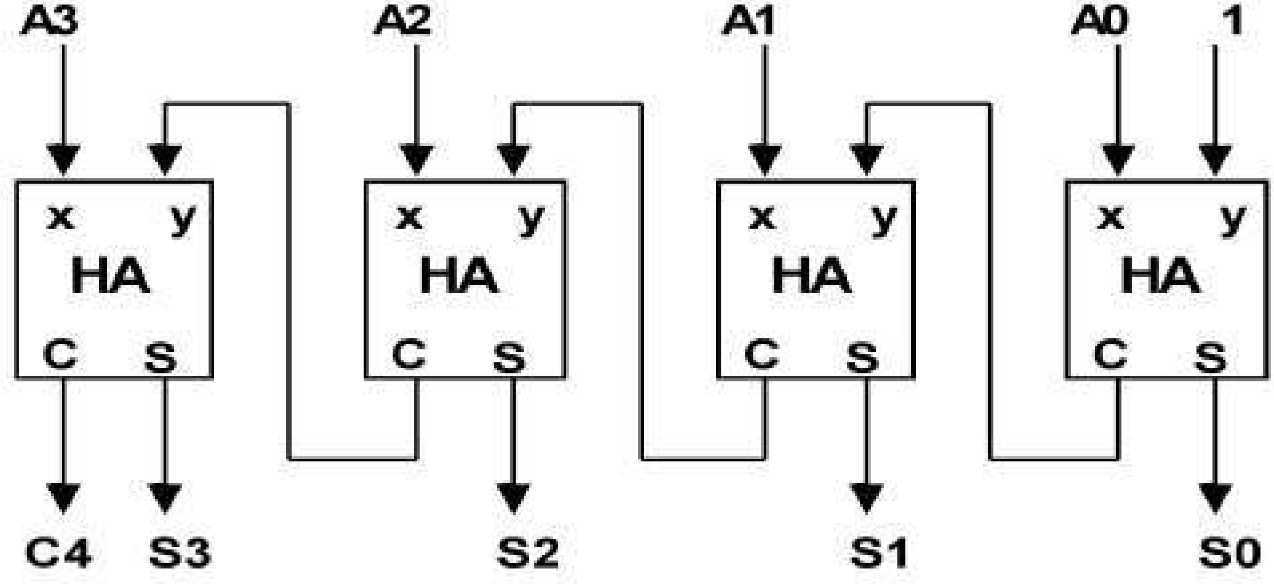


Figure 4-8 4-bit binary incrementer.

Arithmetic Circuit:

→We can implement 7 arithmetic micro-operations (add, add with carry, subtract, subtract with borrow, increment, decrement and transfer) with one circuit.

→The diagram of a 4-bit arithmetic circuit is shown in **Figure 4-9**. It has four full adder circuits that constitute the 4-bit adder and four multiplexers for choosing different operations. There are two 4-bit inputs A and B and a 4-bit output D. The four inputs from A go directly to the X inputs of the binary adder. Each of the four inputs from B is connected to the data inputs of the multiplexers. The multiplexer's data inputs also receive the complement of B. The other two data inputs are connected to logic-0 and logic-1.

→The four multiplexers are controlled by two selection inputs, S_1 and S_0. The input carry C_{in} goes to the carry input of the FA in the least significant position. The other carries a connected from one stage to the next.

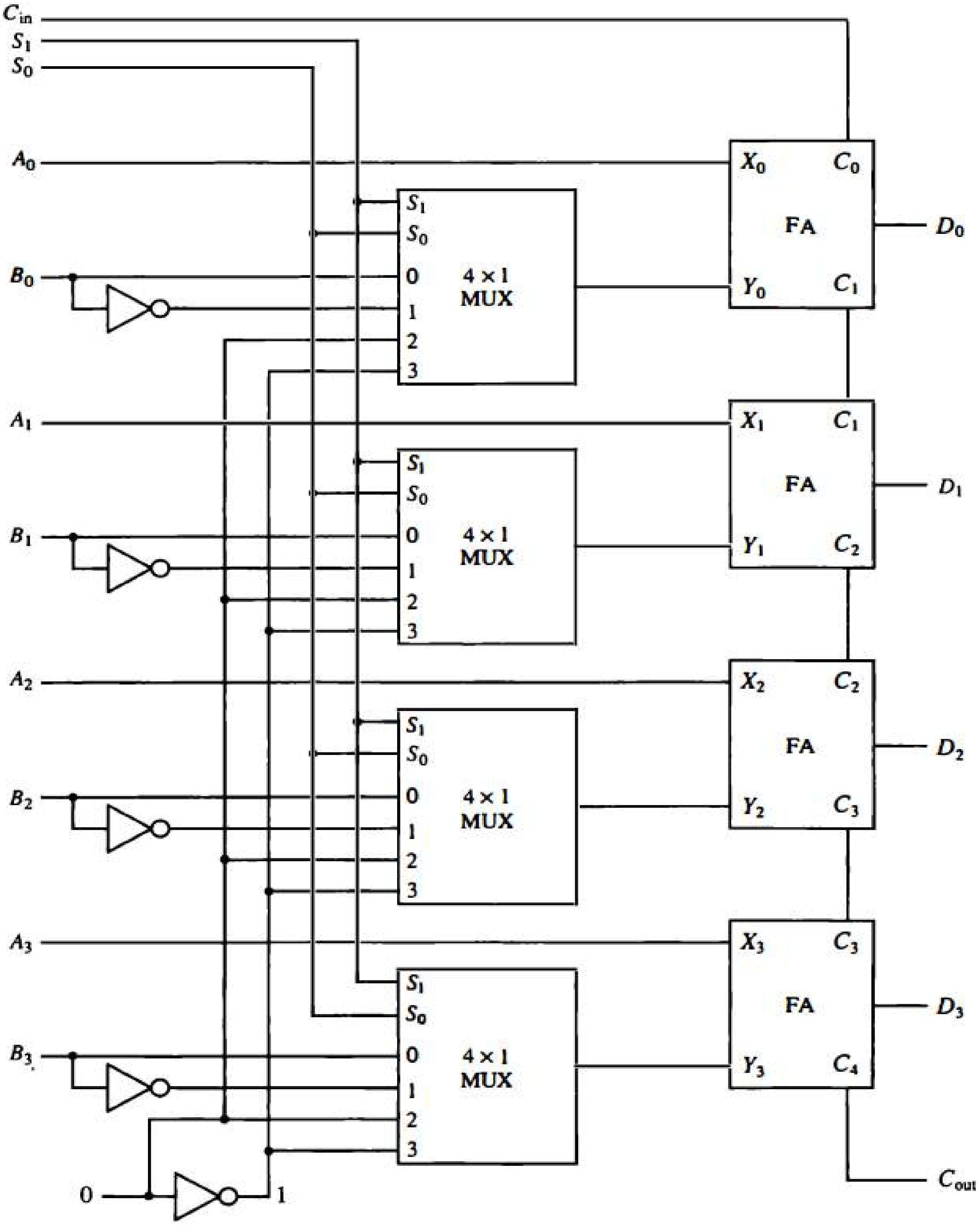


Figure 4-9 4-bit arithmetic circuit.

→The output of the binary adder is calculated from the following arithmetic sum:

D = A + Y + C_{in}

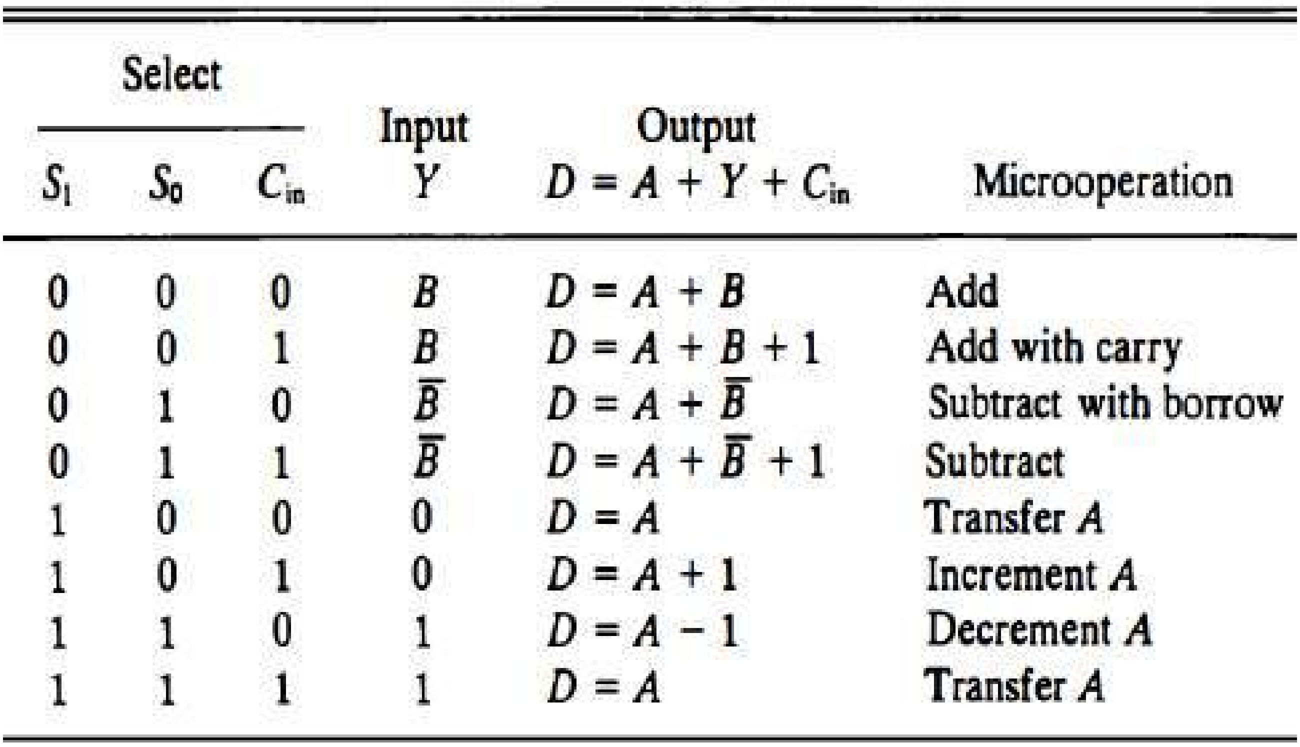
Where

→A is the 4-bit binary number at the X inputs and Y is the 4-bit binary number at the Y inputs of the binary adder.

→C_{in} is the input carry, which can be equal to 0 or 1.

→By controlling the value of Y with the two selection inputs S_1 and S_0 and making C_{in} equal to 0 or 1, it is possible to generate the eight arithmetic micro-operations listed in **Table 4-4.**

TABLE 4-4 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

→When **S_1S_0 = 00**, the value of B is applied to the Y inputs of the adder. If **C_{in} = 0,** the output **D =A+ B**. If **C_{in} = 1**, output **D =A+ B + l**. Both cases perform the add micro-operation **with or without adding the input carry**.

→When **S_1S_0 = 01**, the complement of B is applied to the Y inputs of the adder. If **C_{in} = 1**, then **D =A + B + 1.** This produces a plus the 2's complement of B, which is equivalent to a subtraction of a **- B.** When **C_{in} = 0,** then **D = A + B**. This is equivalent to a **subtract with borrow,** that is, **A - B - 1.**

→When S_1S_0 = 10, the inputs from B are neglected, and instead, all O's are inserted into the Y inputs. The output becomes **D = A + 0 + C_{in}**. This gives D = A when **C_{in} = 0** and **D = A + 1** when **C_{in} = 1**. In the first case we have a direct **transfer from input A** to output D. In the second case, the value of **A is incremented by 1**.

→When **S_1S_0 = 11**, all 1's are inserted into the Y inputs of the adder to produce the **decrement operation D = A - 1** when **C_{in} = 0**. This is because a number with all 1's is equal to the 2's complement of 1 (the 2's complement of binary 0001 is 1111). Adding a number A to the 2's complement of 1 produces **F =A + 2's complement of 1 = A - 1.** When **C_{in} = 1**, then **D =A - 1 + 1 = A**, which causes a direct **transfer from input A** to output D.

5. Logic Micro-operations:

→Logic micro operations specify binary operations for strings of bits stored in registers. These operations consider each bit of register separately and treat them as binary variables. For example:

→The Exclusive-OR of R1 and R2 is symbolized by P: R1 ← R1 ⊕ R2

→It specifies a logic micro-operation to be executed on the individual bits of the registers provided that the control variable P = 1.

Example: R1 = 1010 and R2 = 1100

1010 Content of R1
1100 Content of R2
0110 Content of R1 after P = 1

→Special 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

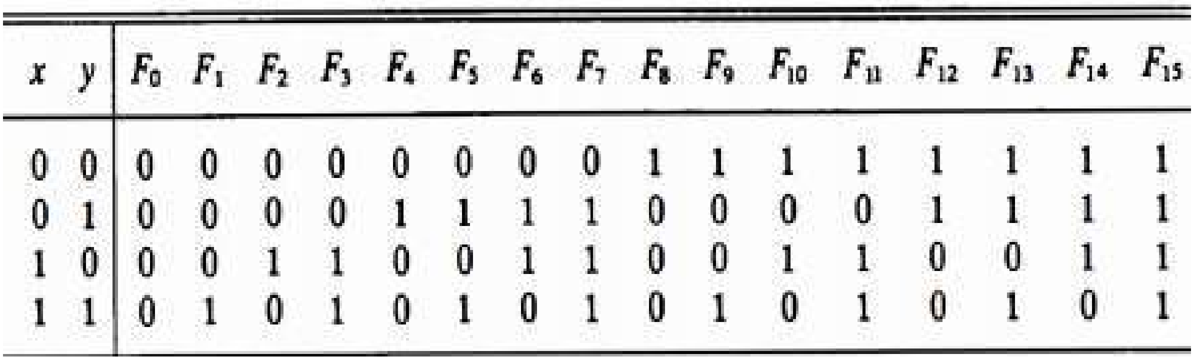
→ Example: P + Q: R1 ← R2 + R3, R4 ← R5 ∨ R6

List of Logic Micro-operations:

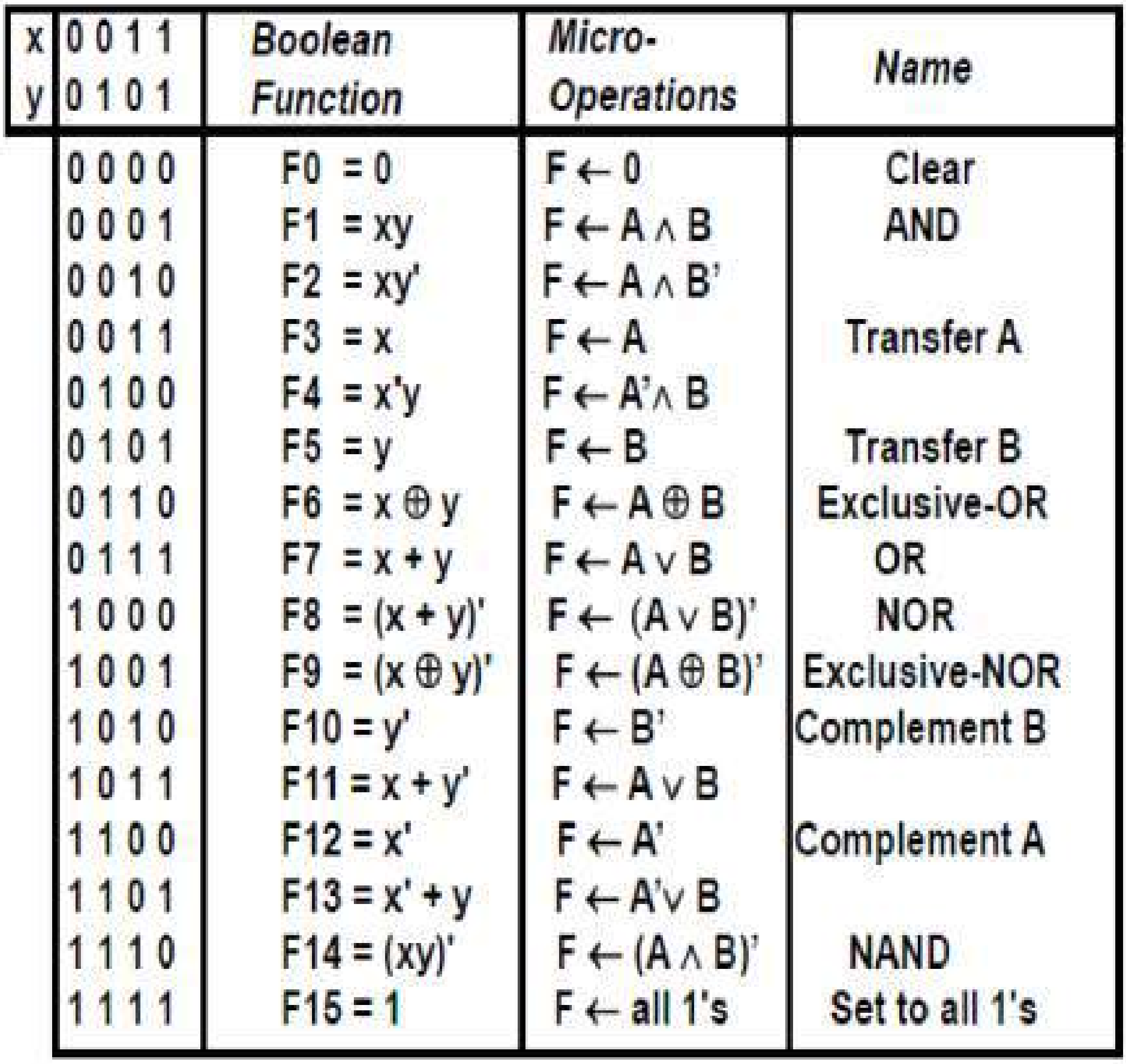
→There are 16 different logic operations that can be performed with two binary variables. They can be determined from all possible truth tables obtained with two binary variables as shown in **Table 4-5**.

→In this table, each of the 16 columns F0 through F15 represents a truth table of one possible Boolean function for the two variables x and y. Note that the functions are determined from the 16 binary combinations that can be assigned to F.

TABLE 4-5 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



x y	0 0 1 1 0 1 0 1	Boolean Function	Micro-Operations	Name
	0 0 0 0	$F0 = 0$	$F \leftarrow 0$	Clear
	0 0 0 1	$F1 = xy$	$F \leftarrow A \wedge B$	AND
	0 0 1 0	$F2 = xy'$	$F \leftarrow A \wedge B'$	
	0 0 1 1	$F3 = x$	$F \leftarrow A$	Transfer A
	0 1 0 0	$F4 = x'y$	$F \leftarrow A' \wedge B$	
	0 1 0 1	$F5 = y$	$F \leftarrow B$	Transfer B
	0 1 1 0	$F6 = x \oplus y$	$F \leftarrow A \oplus B$	Exclusive-OR
	0 1 1 1	$F7 = x + y$	$F \leftarrow A \vee B$	OR
	1 0 0 0	$F8 = (x + y)'$	$F \leftarrow (A \vee B)'$	NOR
	1 0 0 1	$F9 = (x \oplus y)'$	$F \leftarrow (A \oplus B)'$	Exclusive-NOR
	1 0 1 0	$F10 = y'$	$F \leftarrow B'$	Complement B
	1 0 1 1	$F11 = x + y'$	$F \leftarrow A \vee B$	
	1 1 0 0	$F12 = x'$	$F \leftarrow A'$	Complement A
	1 1 0 1	$F13 = x' + y$	$F \leftarrow A' \vee B$	
	1 1 1 0	$F14 = (xy)'$	$F \leftarrow (A \wedge B)'$	NAND
	1 1 1 1	$F15 = 1$	$F \leftarrow$ all 1's	Set to all 1's

TABLE 4-6 Sixteen Logic Micro-operations.

→The **16 Boolean functions** of two variables x and y are expressed in algebraic form in the first column of **Table 4-6**. The 16 logic micro-operations are derived from these functions by replacing variable x by the binary content of register A and variable y by the binary content of register B.

Hardware Implementation:

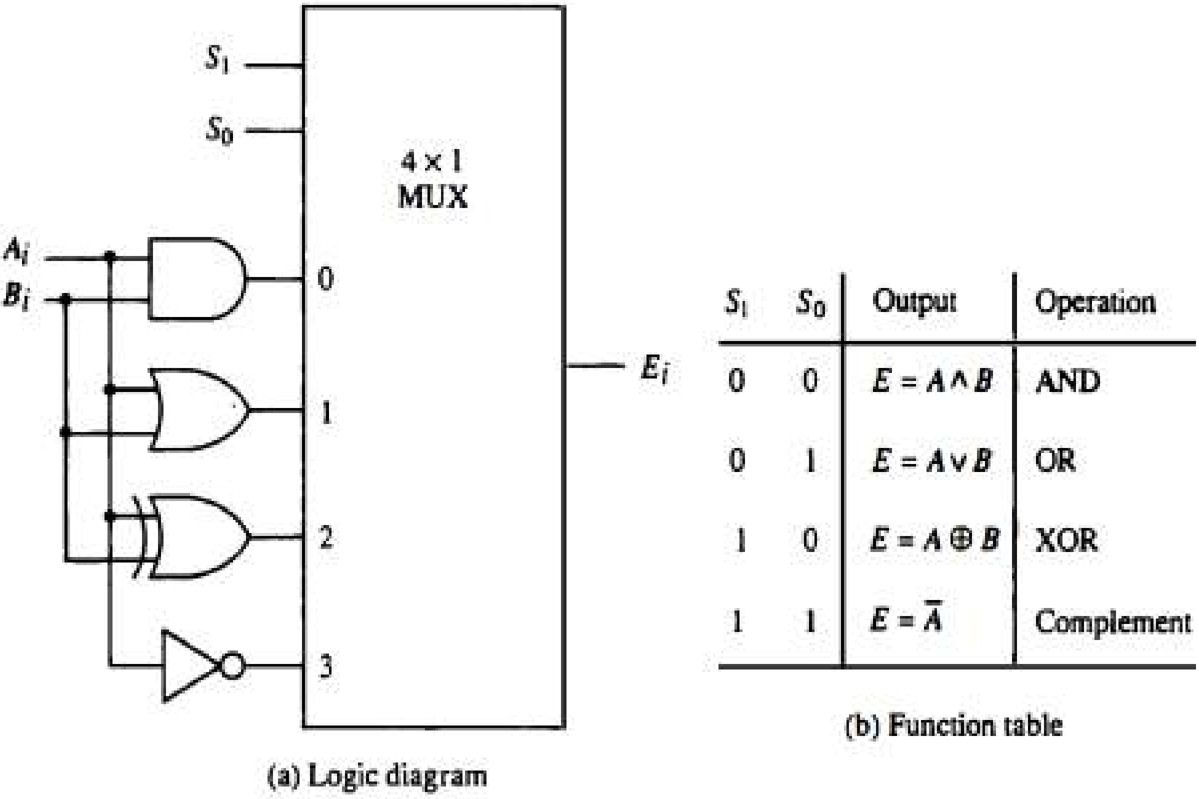
→The hardware implementation of logic micro-operations requires that logic gates be inserted for each bit or pair of bits in the registers to perform the required logic function.

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

→**Figure 4-10** shows one stage of a circuit that generates the four basic logic micro-operations. It consists of four gates and a multiplexer. Each of the four logic operations is generated through a gate that performs the required logic. The outputs of the gates are applied to the data inputs of the multiplexer. The two selection inputs S_1 and S_0 choose one of the data inputs of the multiplexer and direct its value to the output. The diagram shows one typical stage with subscript i. For a logic circuit with n bits, the diagram must be repeated n times for i = 0, 1, 2, ...,n - 1.

→ The selection variables are applied to all stages. The **function table in fig 4-10(b)** lists the logic micro-operations obtained for each combination of the selection variables.

Figure 4-10 One stage of logic circuit.



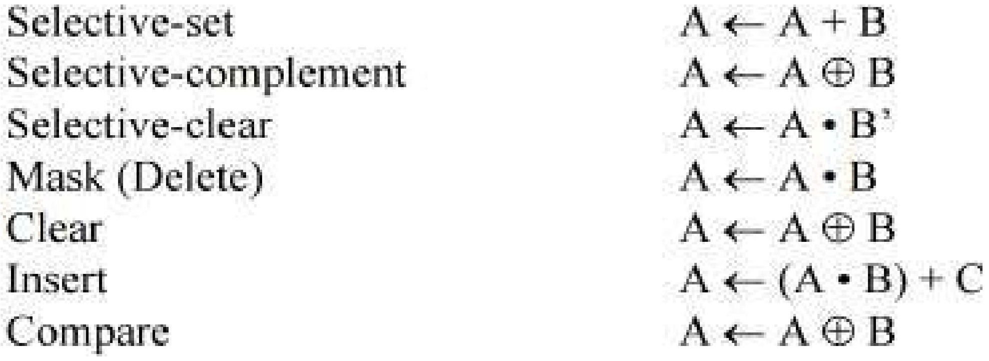
S_1	S_0	Output	Operation
0	0	$E = A \wedge B$	AND
0	1	$E = A \vee B$	OR
1	0	$E = A \oplus B$	XOR
1	1	$E = \bar{A}$	Complement

(a) Logic diagram

(b) Function table

Some Applications:

→Logic micro-operations can be used to manipulate individual bits or portions of a word in a register. Consider the data in a register A. In another register B, is bit data that will be used to modify the contents of A.



Selective-set	$A \leftarrow A + B$
Selective-complement	$A \leftarrow A \oplus B$
Selective-clear	$A \leftarrow A \bullet B'$
Mask (Delete)	$A \leftarrow A \bullet B$
Clear	$A \leftarrow A \oplus B$
Insert	$A \leftarrow (A \bullet B) + C$
Compare	$A \leftarrow A \oplus B$

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

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

A ← A ∨ B

If a bit in register B is set to 1, that same position in register A gets set to 1, otherwise that bit in register A keeps its previous value.

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

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

A ← A ⊕ B

If a bit in B is set to 1, that same position in A gets complemented from its original value, otherwise it is unchanged.

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

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

A ← A ∧~~B~~

If a bit in register B is set to 1 that same position in register A gets set to 0 otherwise it is unchanged.

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

1010 A before

1100 B (logic operand)

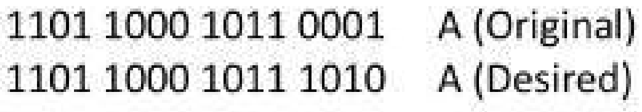
1000 A after

A ← A ∧ B

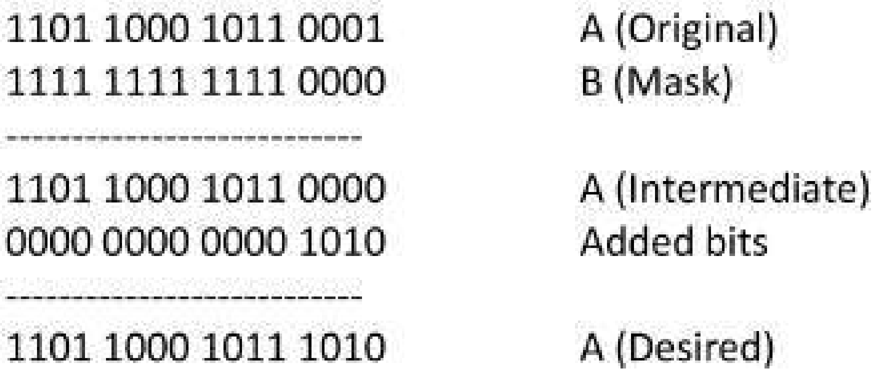
If a bit in register B is set to 0, that same position in register A gets set to 0, otherwise it is unchanged.

→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

Example » Suppose you wanted to introduce 1010 into the low order four bits of A:



```
1101 1000 1011 0001     A (Original)
1101 1000 1011 1010     A (Desired)
```



```
1101 1000 1011 0001              A (Original)
1111 1111 1111 0000              B (Mask)
--------------------------
1101 1000 1011 0000              A (Intermediate)
0000 0000 0000 1010              Added bits
--------------------------
1101 1000 1011 1010              A (Desired)
```

The mask operation is an AND micro-operation and the insert operation is an OR micro-operation.

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

1010 A

1010 B

0000 **A ← A ⊕B**

This operation is achieved by exclusive-OR micro-operation.

6. Shift Micro-operations:

→Shift micro-operations are used for serial transfer of data and are used in conjunction with arithmetic and logic operations. The register contents can be shifted to the left or to the right.

→There are three types of shift operations:

- **Logical shift:** Logical shifts transfers 0 through the serial input, with all the bits involved in the shifting.
- **Circular shift:** A circular shift circulates the bits of the register around the two ends with no loss of information.
- **Arithmetic shift:** Arithmetic shifts multiplies (or divides) a signed number by 2.

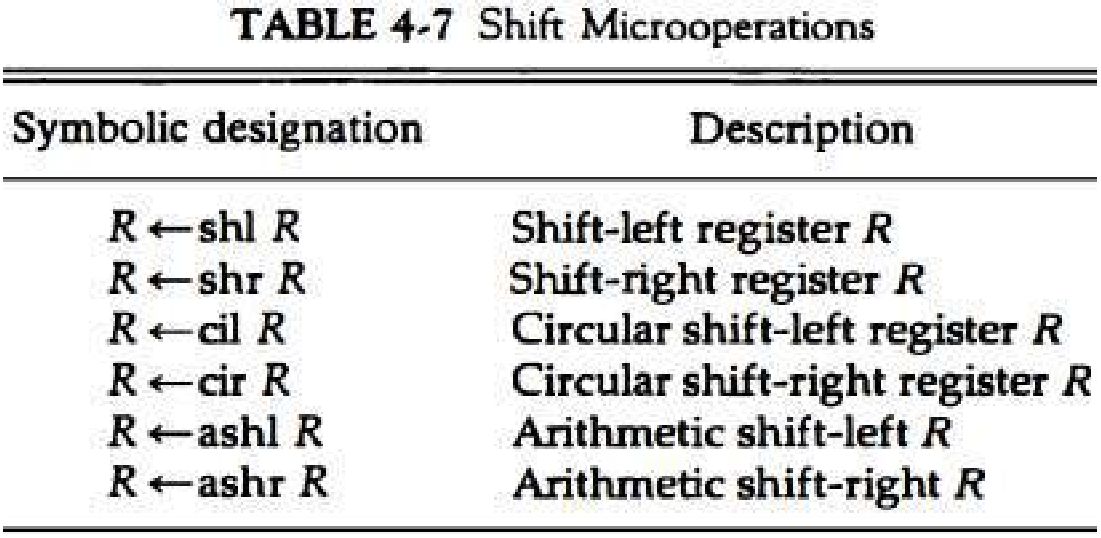
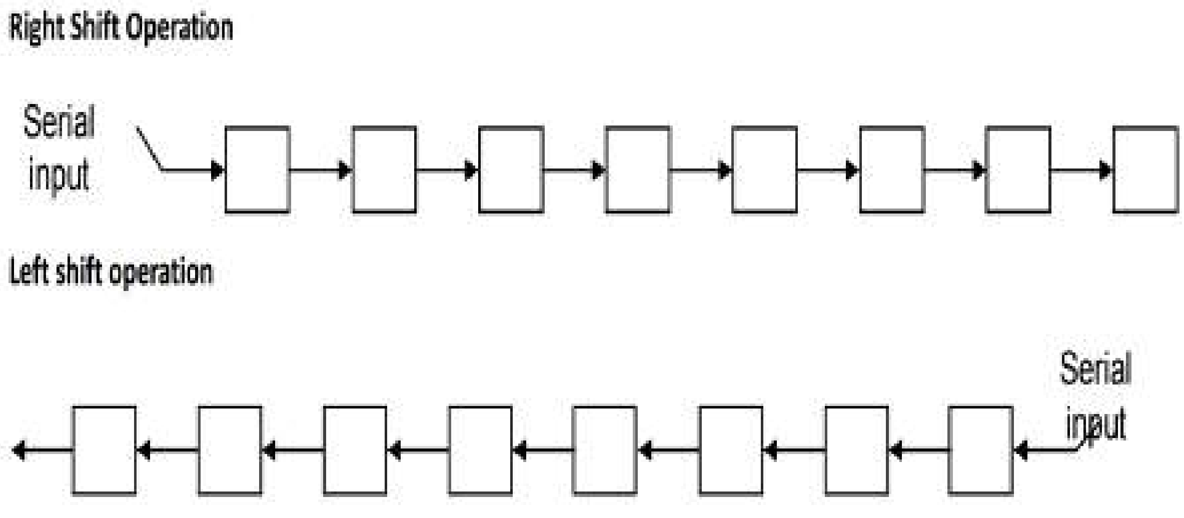


TABLE 4-7 Shift Microoperations

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

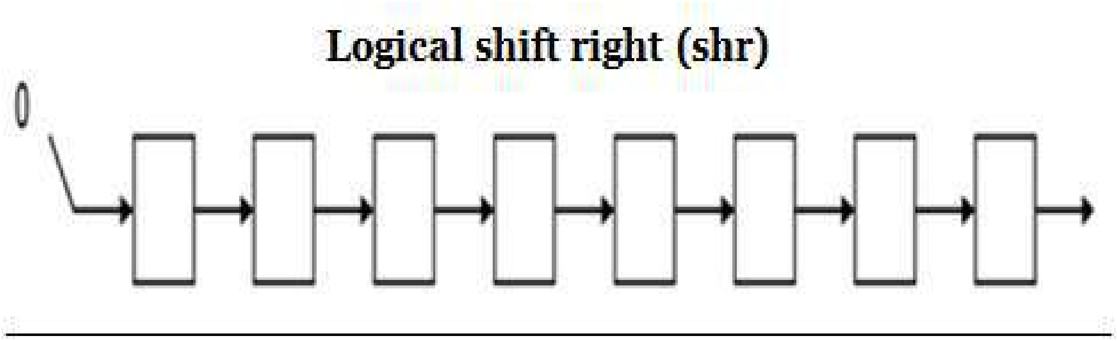


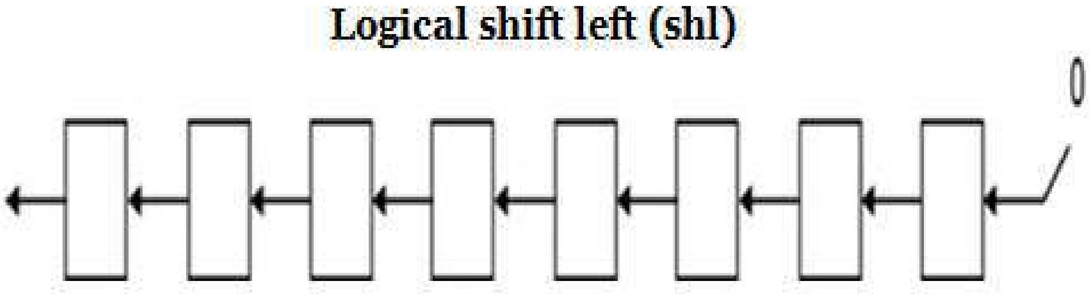
1. Logical shift: A logical shift is one that transfers 0 through the serial input. In a Register Transfer Language (RTL), the following notation is used.

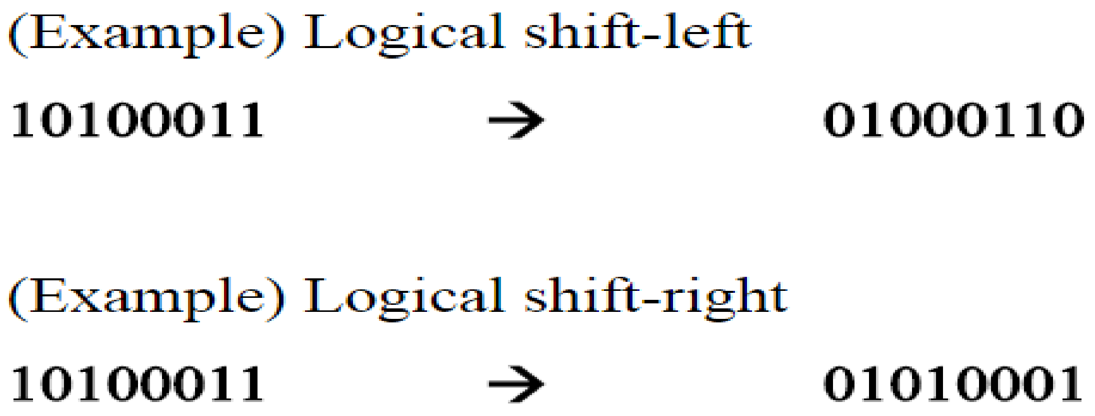
- **shl** for a logical shift left
- **shr** for a logical shift right

For example, **R1🡐shl R1**

R2← shr R2


Logical shift right (shr)


Logical shift left (shl)


(Example) Logical shift-left

10100011 → 01000110

(Example) Logical shift-right

10100011 → 01010001

2. Circular Shift (rotate operation): A Circular-shift circulates the bits of the resister around the two ends without the loss of information.

In a RTL, the following notation is used:

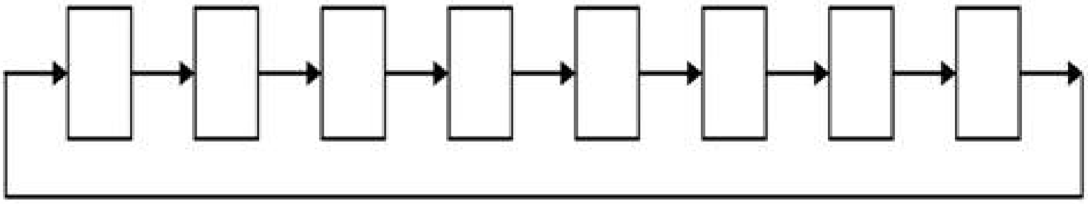
- **cil**for a circular shift left.
- **cir** for a circular shift right.

For example**,**

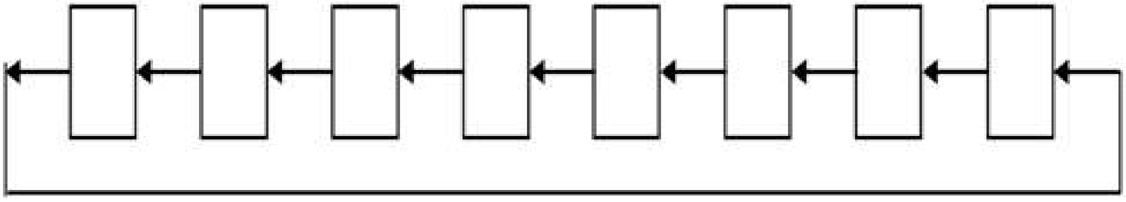
cir R2 ← R2

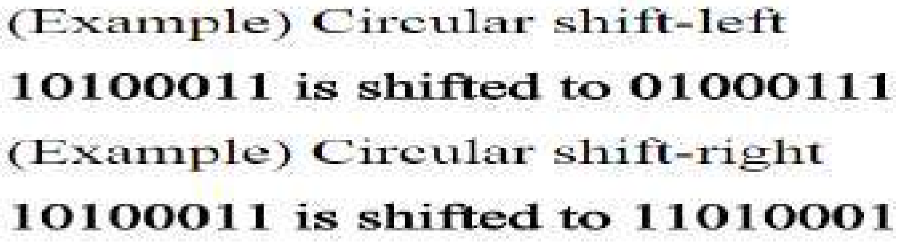
cil R3 ← R3

Right circular shift operation



Left circular shift operation:





(Example) Circular shift-left

10100011 is shifted to 01000111

(Example) Circular shift-right

10100011 is shifted to 11010001

3. Arithmetic shift: An arithmetic shift is meant for signed binary numbers (integer). An arithmetic left shift multiplies a signed number by 2 and an arithmetic right shift divides a signed number by 2. Arithmetic shifts must leave the sign bit unchanged because the sign of the number remains the same when it is multiplied or divided by 2. The left most bit in a resister holds a sign bit and remaining hold the number. Negative numbers are in 2's complement form.

In a Resister Transfer Language, the following notation is used:

ashl for an arithmetic shift left.

ashr for an arithmetic shift right.

For example,

ashr R2 ← R2

ashl R3 ← R3

Arithmetic shift right: Arithmetic shift-right leaves the sign bit unchanged and shift the number (including a sign bit) to the right. Thus $\mathbf{R}_{\mathbf{n-1}}$remains same; $\mathbf{R}_{\mathbf{n-2}}$receives input from $\mathbf{R}_{\mathbf{n-1}}$and **so on**.

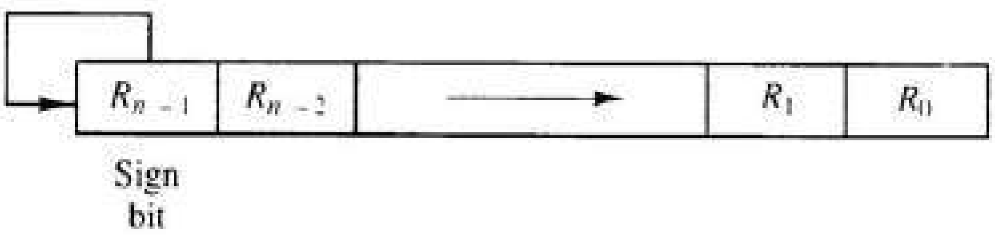
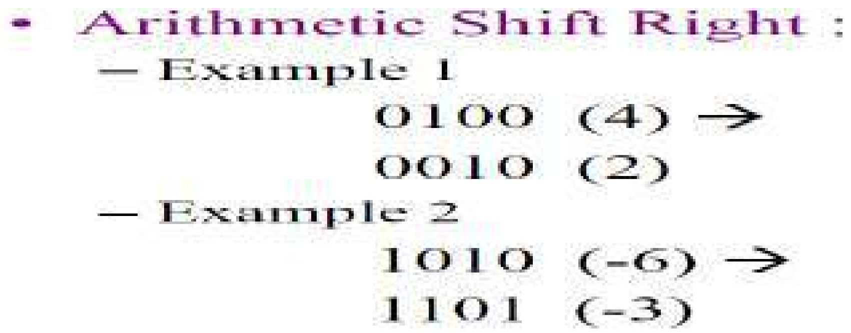


Figure 4-11 Arithmetic shift right.



- Arithmetic Shift Right :
  - Example 1

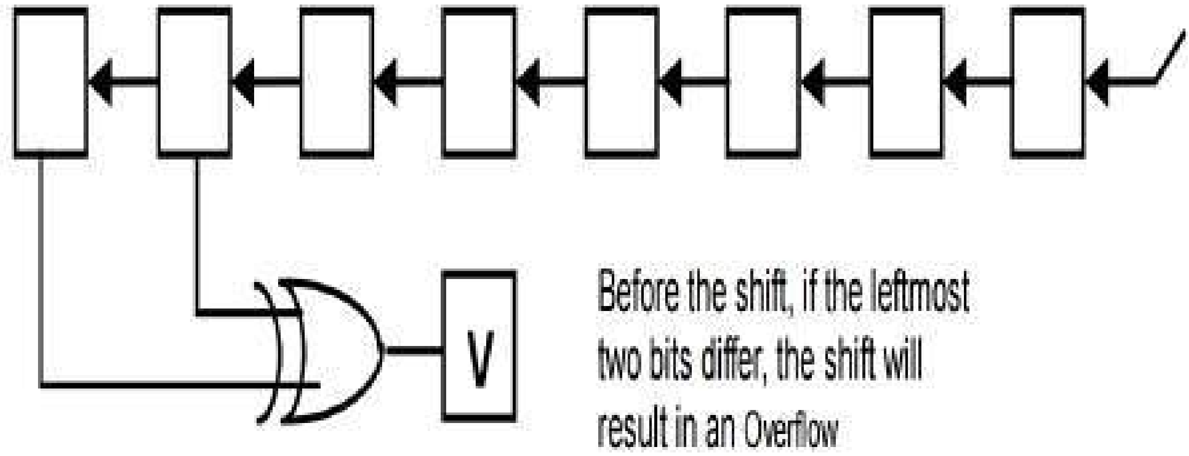
    0100 (4) →
    0010 (2)
  - Example 2

    1010 (-6) →
    1101 (-3)

Arithmetic shift-left: Arithmetic shift-left inserts a **0 into** $\mathbf{R}_{\mathbf{0}}$and shifts all other bits to left. Initial bit of $\mathbf{R}_{\mathbf{n-1}}$is lost and replaced by the bit from $\mathbf{R}_{\mathbf{n-2}}$.

Overflow case during arithmetic shift-left: If a bit in $\mathbf{R}_{\mathbf{n-1}}$changes in value after the shift, sign reversal occurs in the result. This happens if the multiplication by 2 causes an overflow. Thus, left arithmetic shift operation must be checked for the overflow: an overflow occurs after an arithmetic shift-left if before shift $\mathbf{R}_{\mathbf{n-1}} \neq \mathbf{R}_{\mathbf{n-2}}$.

→ Before the shift, if the left most two bits differ, the shift will result in an Overflow.

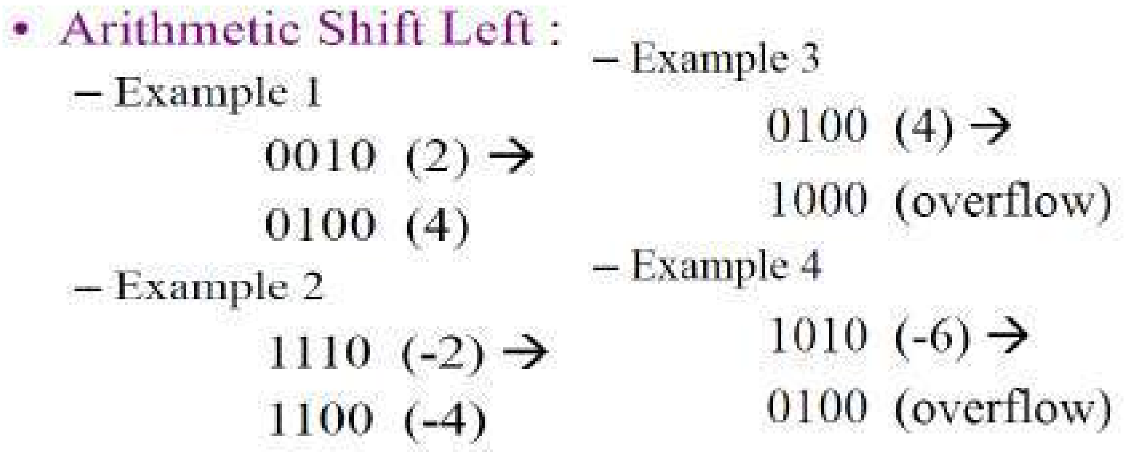


Before the shift, if the leftmost two bits differ, the shift will result in an Overflow

→An overflow flip-flop **V_s** can be used to detect an arithmetic shift-left overflow.

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

→If **V=0**, there is **no overflow** but if **V=1**, **overflow** is detected.



- Arithmetic Shift Left :
  - Example 1
    0010 (2) →
    0100 (4)
  - Example 2
    1110 (-2) →
    1100 (-4)
  - Example 3
    0100 (4) →
    1000 (overflow)
  - Example 4
    1010 (-6) →
    0100 (overflow)

Hardware Implementation of shift micro-operations:

→A bi-directional shift unit with parallel load could be used to implement this.

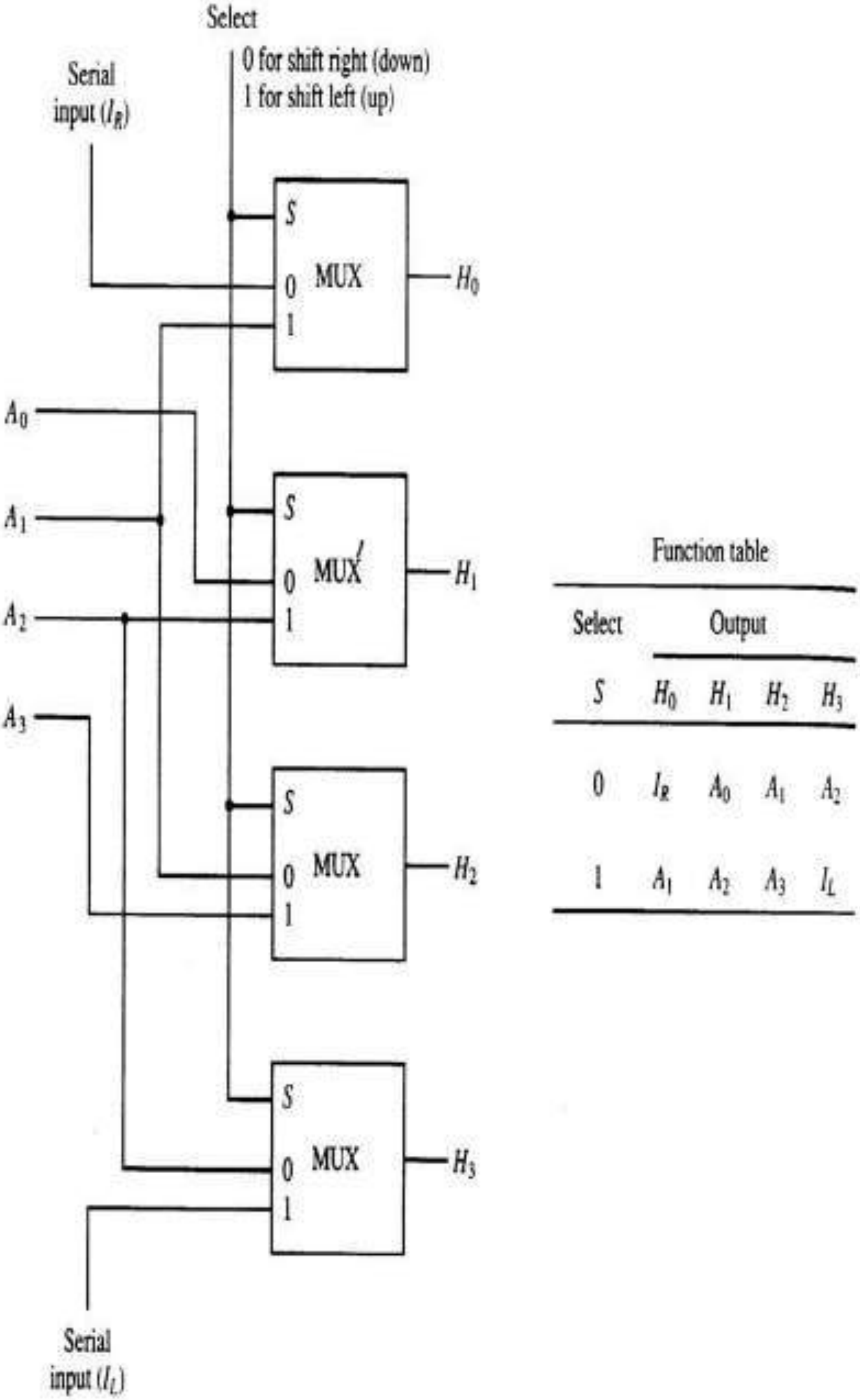
→Two clock pulses are necessary with this configuration: one to load the value and another to shift.

→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.

→A combinational circuit shifter can be constructed with multiplexers as shown in **Figure 4-12.** The 4-bit shifter has four data inputs, A_0 through A_3, and four data outputs, H_0 through H_3. There are two serial inputs, one for shift left (I_L) and the other for shift right (I_R). When the selection input **S = 0**, the input data are shifted right (**down in the diagram**). When **S = 1**, the input data are shifted left (**up in the diagram**).

The **function table in Figure 4-12** shows which input goes to each output after the shift. A shifter with n data inputs and outputs requires n multiplexers. The two serial inputs can be controlled by another multiplexer to provide the three possible types of shifts.



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

Figure 4-12 4-bit combinational circuit shifter.

7. Arithmetic Logic Shift Unit:

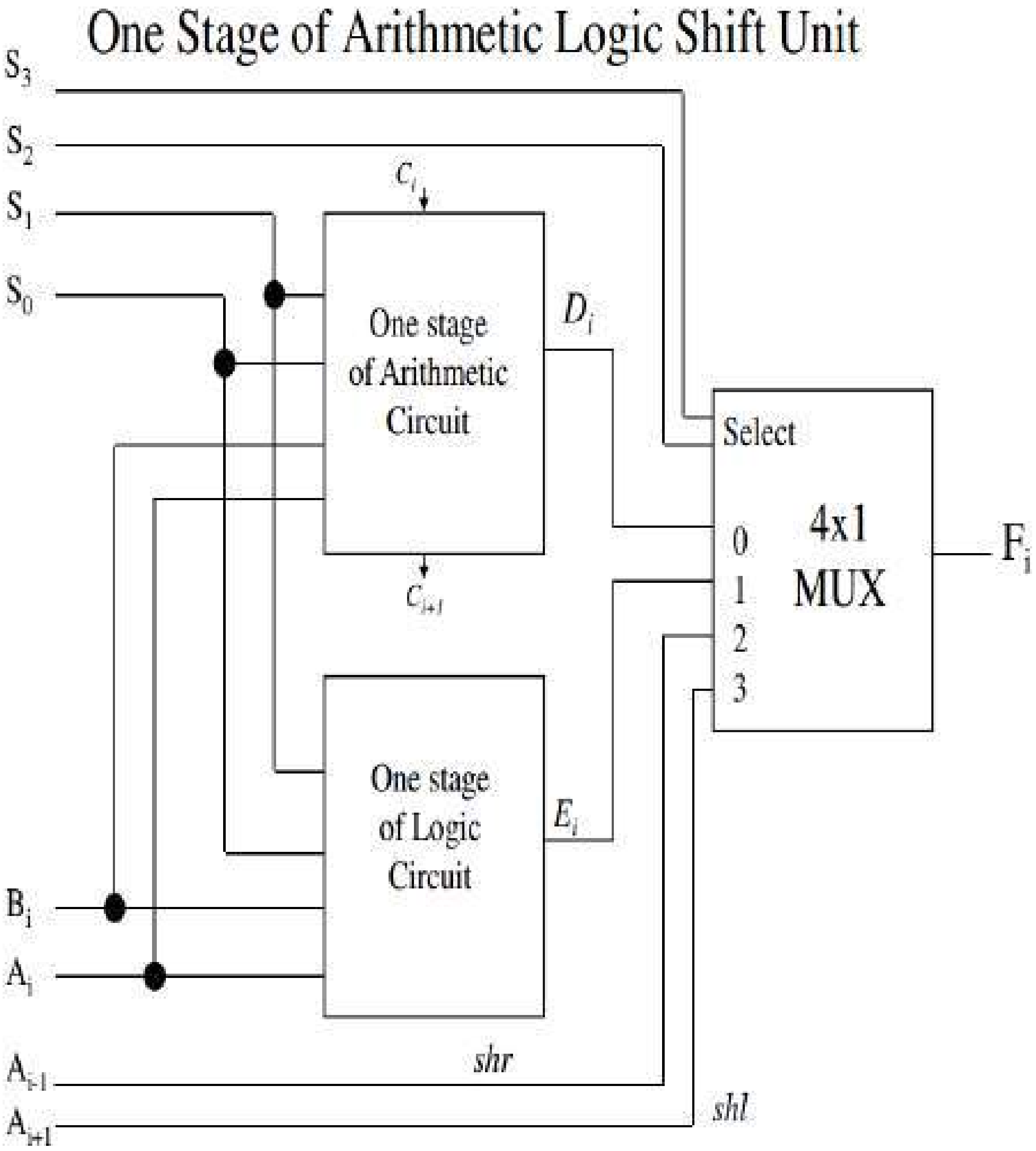

One Stage of Arithmetic Logic Shift Unit
S3
S2
Ci
S1
S0
One stage of Arithmetic Circuit
Di
Ci+1
Select
4x1 MUX
0
1
2
3
Fi
One stage of Logic Circuit
Ei
Bi
Ai
shr
Ai-1
Ai+1
shl

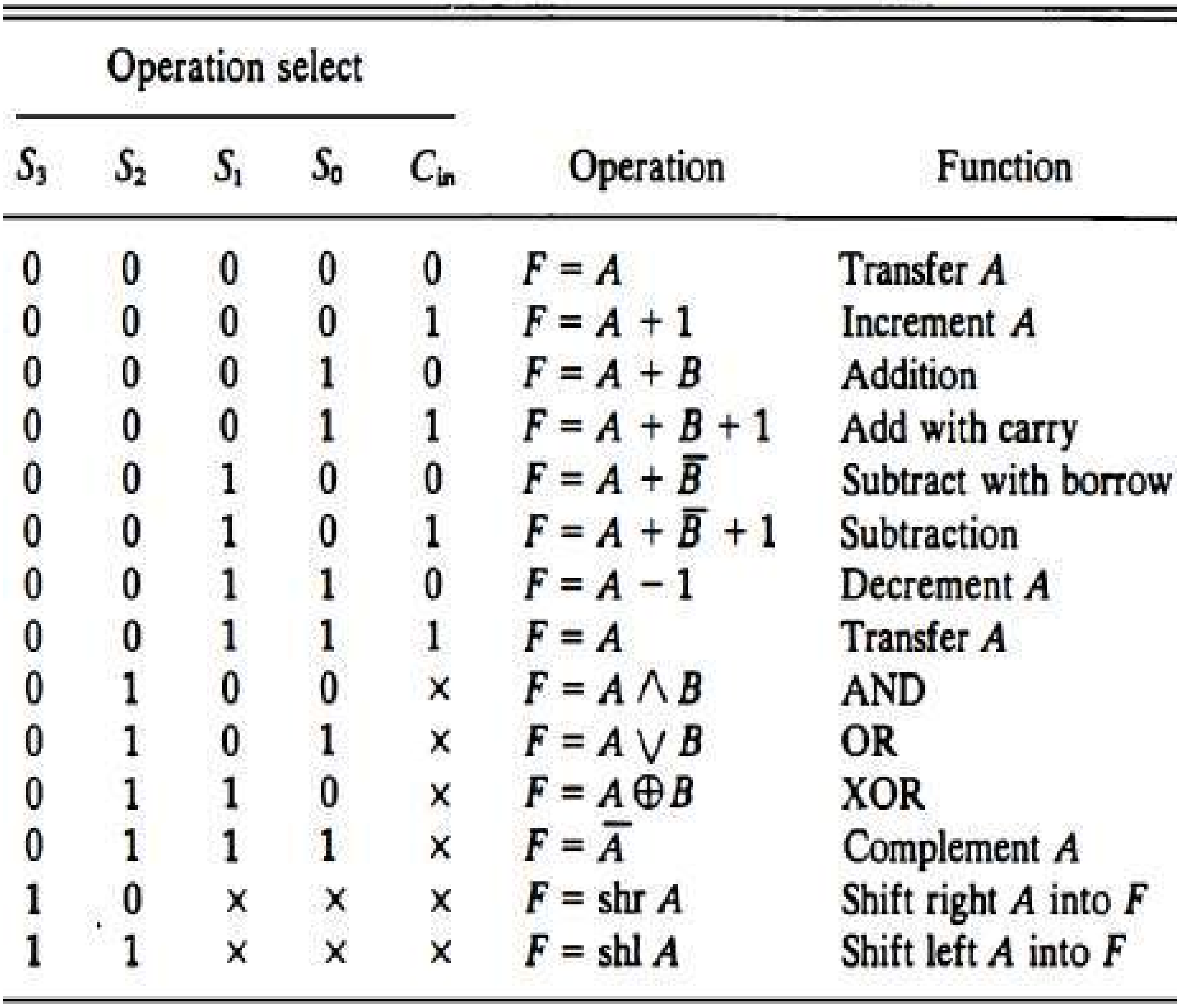
Figure 4-13.One stage of arithmetic logic shift unit.

→This is a common operational unit called **arithmetic logic unit (ALU)**. To perform a micro-operation, the contents of specified registers are placed in the inputs of the common ALU. The ALU performs the operation and transfer result to destination resister.

→A particular micro-operation is selected with inputs **s_1 and s_0.** A 4x1 MUX at the output chooses between an arithmetic output in D_i and logic output E_i. Other two inputs to the MUX receive inputs A_{i-1} for right-shift operation and A_{i+1} for left-shift operation. The diagram shows just one typical stage. The circuit must be repeated n times for an n-bit ALU.

→This circuit provides 8 arithmetic operations, 4 logic operations and 2 shift operations. Each operation is selected with five variables **S_3, S_2, S_1, S_0 and C_{in}**. The input carry **C_{in}** is used for arithmetic operations only.

TABLE 4-8 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*

→**Table 4-8** lists the 14 operations of the ALU. The first eight are arithmetic operations and are selected with **S_3S_2 = 00**. The next four are logic operations and are selected with **S_3S_2 = 01**. The input carry has no effect during the logic operations and is marked with don't-care

X's. The last two operations are shift operations and are selected with **S_3S_2= 10 and 11.** The other three selection inputs have no effect on the shift.

8. Instruction Codes:

→The internal organization of a digital system is defined by the sequence of micro-operations it performs on data stored in its registers. By executing several micro-operations in specified sequence, then a computer instruction can be executed.

Program: A set of instructions that specifies operation, operands, and sequence of processing has to occur

- The instructions of a program, along with any needed data are stored in memory. The CPU reads the next instruction from memory.
- It is placed in an Instruction Register(IR)
- Control circuitry in control unit then translates the instruction into the sequence of micro-operations necessary to implement it

→A **Computer Instruction** is a binary code that specifies a sequence of micro-operations for the computer. Every computer has its own unique instruction set

→Instruction code is a group of bits that instruct the computer to perform a specific operation.

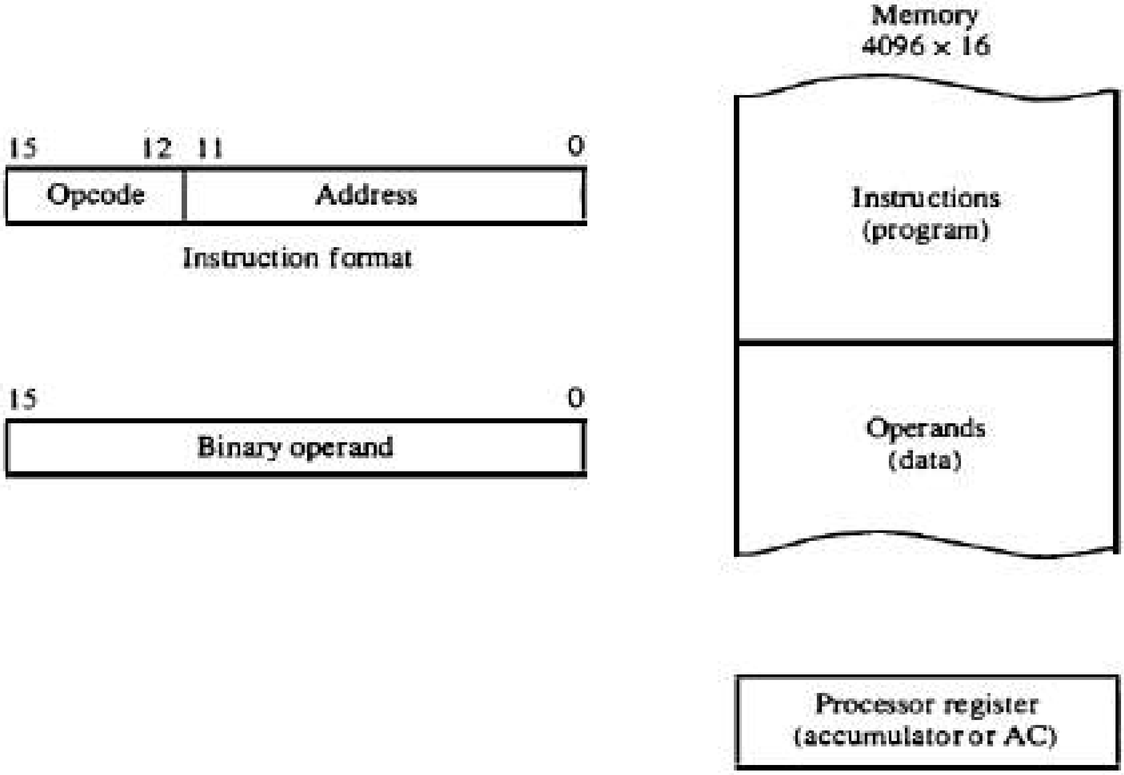
→**Operation code**of an instruction is a group of bits that define such operations as add, subtract, multiply, shift, and complement.

Stored Program Organization:

→The simplest way to organize a computer is to have one processor register andan instruction code format with two parts. The first part specifies the operation to be performed and the second specifies an address. The memory address tells the control where to find an operand in memory. This operand is read from memory and used as the data to be operated on together with the data stored in the processor register.

→**Figure 5-1** depicts this type of organization. Instructions are stored in one section of memory and data in another. For a memory unit with 4096 words we need 12 bits to specify an address since 2^{12} = 4096. If we store each instruction code in one 16-bit memory word, we have available four bits for the *operation code* (op-code) to specify one out of 16 possible operations, and 12 bits to specify the address of an operand. The control reads a 16-bit instruction from the program portion of memory. It uses the 12-bit address part of the instruction to read a 16-bit operand from the data portion of memory. It then executes the operation specified by the operation code.

Figure 5-1 Stored program organization.



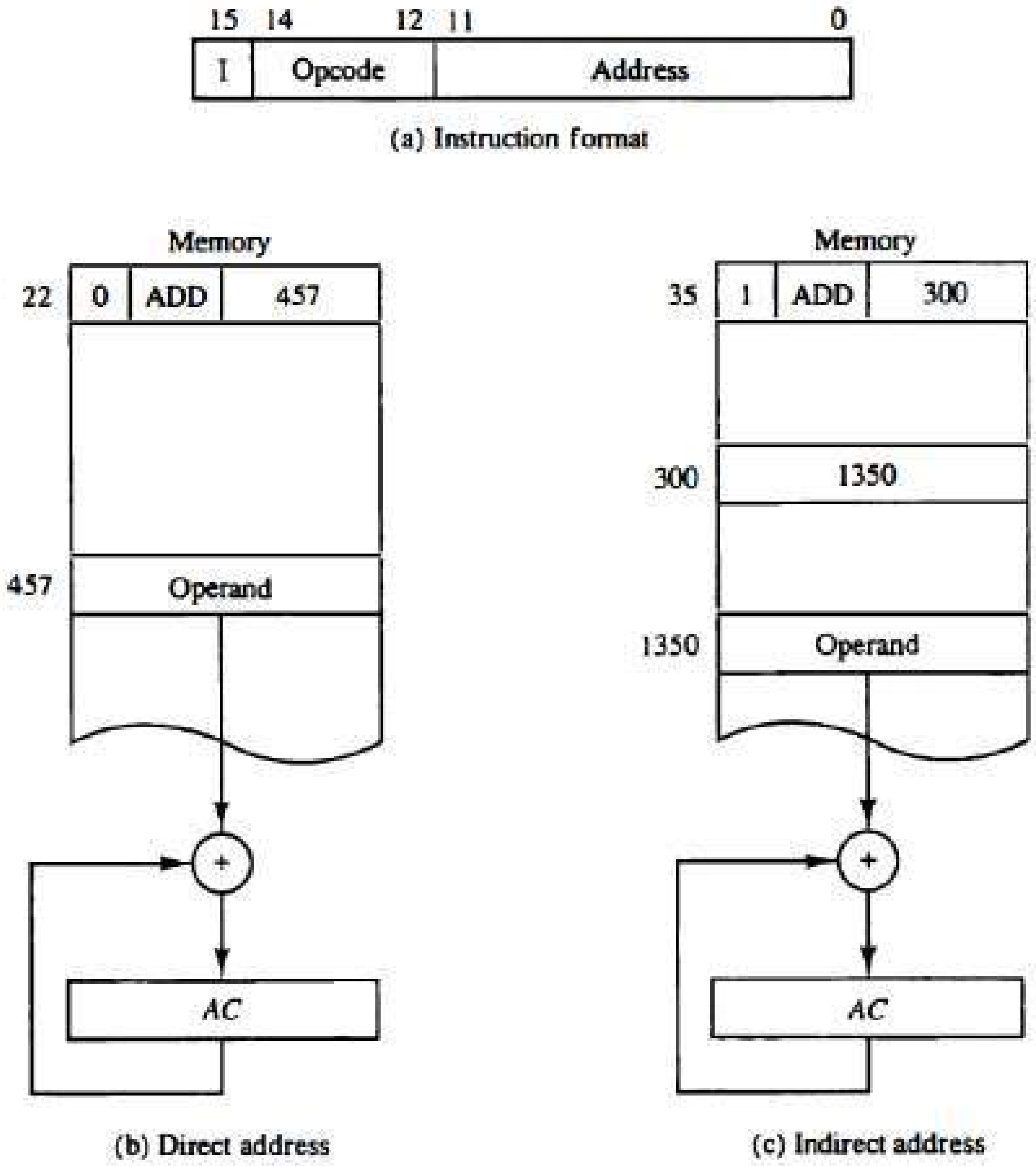
→**Accumulator Register (AC):** Computers that have a single register usually assign to it the name accumulator and label it AC. The operation is performed with the memory operand and the content of AC.

Indirect Address:

→One bit of the instruction code is used to distinguish between a direct and an indirect address.

→As an illustration of this configuration, consider the instruction code format shown in **Figure 5-2(a).** It consists of a 3-bit operation code, a 12-bit address, and an indirect address mode bit designated by **I**. The mode bit is 0 for a direct address and 1 for an indirect address.

→A direct address instruction is shown in **Figure 5-2(b).**It is placed in address 22 in memory. The I bit is 0, so the instruction is recognized as a direct address instruction. The op-code specifies an ADD instruction, and the address part is the binary equivalent of 457. The control finds the operand in memory at address 457 and adds it to the content of AC.



(a) Instruction format

(b) Direct address

(c) Indirect address

Figure 5-2 Demonstration of direct and indirect address.

→The instruction in address 35 shown in **Figure 5-2(c)** has a mode bit I = 1. Therefore, it is recognized as an indirect address instruction. The address part is the binary equivalent of 300. The control goes to address 300 to find the address of the operand. The address of the operand in this case is 1350. The operand found in address 1350 is then added to the content of AC.

→**Effective address**: Address where an operand is physically located. The address that can be directly used without modification to access an operand for a computation-type instruction, or as the target address for a branch-type instruction

→Thus the effective address in the instruction of **Figure 5-2(b)** is **457** and in the instruction of **Figure 5-2(c)** is **1350.**

9. Computer Registers:

A register is a very small amount of very fast memory that is built into the CPU (central processing unit). Contents can be accessed at extremely high speeds. Registers are used to store data temporarily during the execution of a program. Different processors have different register sizes. Registers are normally measured by the number of bits they can hold, for example, an 8-bit register means it can store 8 bits of data or a 32-bit register means it can store 32 bit of data.

→The computer needs processor registers for manipulating data and a register for holding a memory address. These requirements dictate the register configuration shown in **Figure 5-3.** The registers are also listed in **Table 5-1** together with a brief description of their function and the number of bits that they contain.

→The memory unit has a capacity of 4096 words and each word contains 16 bits. Twelve bits of an instruction word are needed to specify the address of an operand.

ACCUMULATOR (AC): The processor register AC consists of 16-bits. It is used to hold the results or partial results of arithmetic and logical operations. An accumulator is a register in which intermediate arithmetic and logic results are stored.

DATA REGISTER (DR): The register DR consists of 16-bits and it is used to hold memory operands (data). This register contains the data to be written into memory or receives the data read from memory.

TEMPORARY REGISTER (TR): Temporary register have 16-bits and it provides temporary storage of variables or results.

INSTRUCTION REGISTER (IR): The instruction register consists of 16-bits. The purpose of the instruction register is to hold a copy of the instruction which the processor is to execute. In our basic computer, instruction register (IR) holds instruction code which is read from memory.

ADDRESS REGISTER (AR): This register specifies the address in memory for next read or writes operations. The address register consists of 12-bits.

PROGRAM COUNTER (PC): Program counter has 12-bits and it holds the address of the next instruction to be read from memory after the current execution is executed. The instructions are read sequentially because the Program Counter (PC) automatically increments after fetching the current instruction.

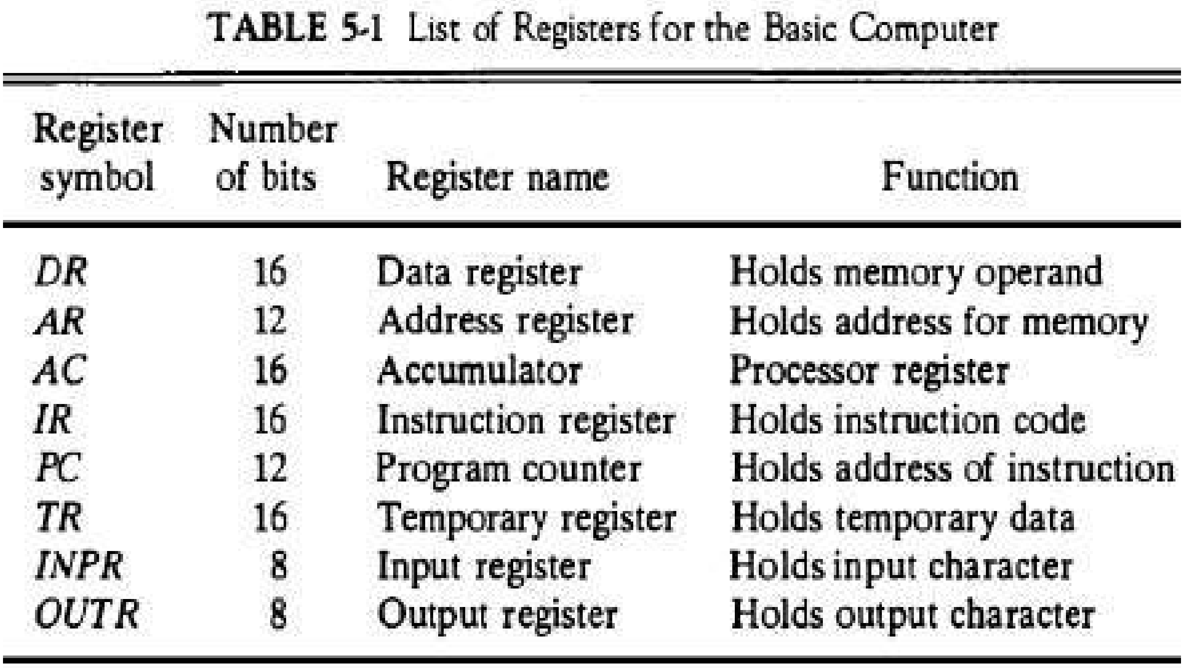


TABLE 5-1 List of Registers for the Basic Computer

Register symbol	Number of bits	Register name	Function
DR	16	Data register	Holds memory operand
AR	12	Address register	Holds address for memory
AC	16	Accumulator	Processor register
IR	16	Instruction register	Holds instruction code
PC	12	Program counter	Holds address of instruction
TR	16	Temporary register	Holds temporary data
INPR	8	Input register	Holds input character
OUTR	8	Output register	Holds output character

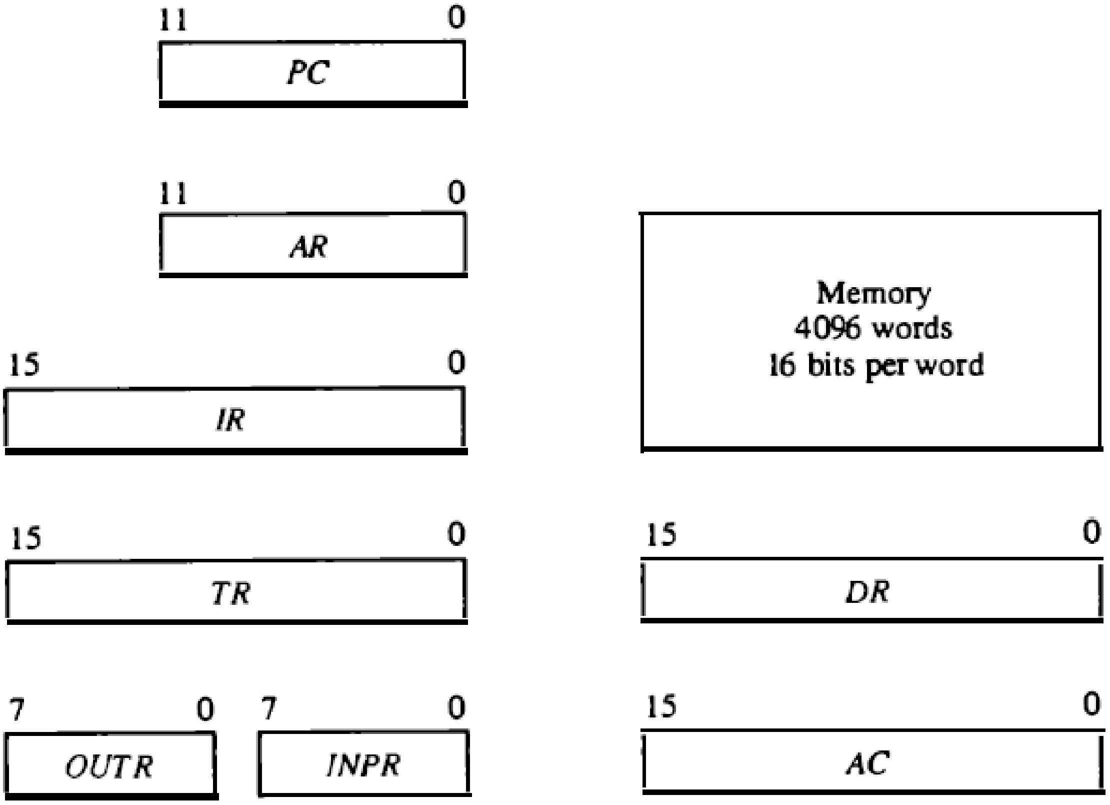


Figure 5-3 Basic computer registers and memory.

→Two registers are used for input and output. The **input register (INPR)** receives an 8-bit character from an input device. The **output register (OUTR)** holds an 8-bitcharacter for an output device.

Common Bus System:

BUS: A wire or a collection of wires that carry some multi-bit information is known as bus. Main purpose of bus is to transfer information form one system to another.

DESCRIPTION: The basic computer has eight registers (AC, PC, DR, AC, IR, TR, INPR, and OUTR), a memory unit and a control unit. Path must be provided to transfer information from one register to another and between memory and registers. The number of wires will be excessive if connections are made between the output of each register and input of other registers. A more efficient scheme is to use a common bus. Thus common bus provides a path between memory unit and registers. The connection of the registers and memory of the basic computer to a common bus system is shown in **Figure 5-4.**

→Five registers have three control inputs: **LD (load), INR (increment) and CLR (clear).** Two registers have only a **LD input.**

Load (LD): The lines from the common bus are connected to the inputs of each register and the data inputs of the memory. The particular register whose LD input is enabled receives the data from the bus.

Increment (INR)) and Clear (CLR): The contents of the particular register are incremented when its INR signal is enabled and cleared when its CLR signal is enabled.

Memory Unit: The memory receives the 16-bit information from the bus when its write input is enabled and the memory places its 16-bit information onto the bus when its read input is activated and $S_2S_1S_0$ = 111.

Address Register (AR): This register specifies the address in memory for next read or writes operations. The address register consists of 12 bits. When selection inputs $S_2S_1S_0$ =001 is applied to the bus, the address register AR receives or transfers address from or to the bus when its LD input is enable. The address is incremented or clear by the inputs INR or CLR.

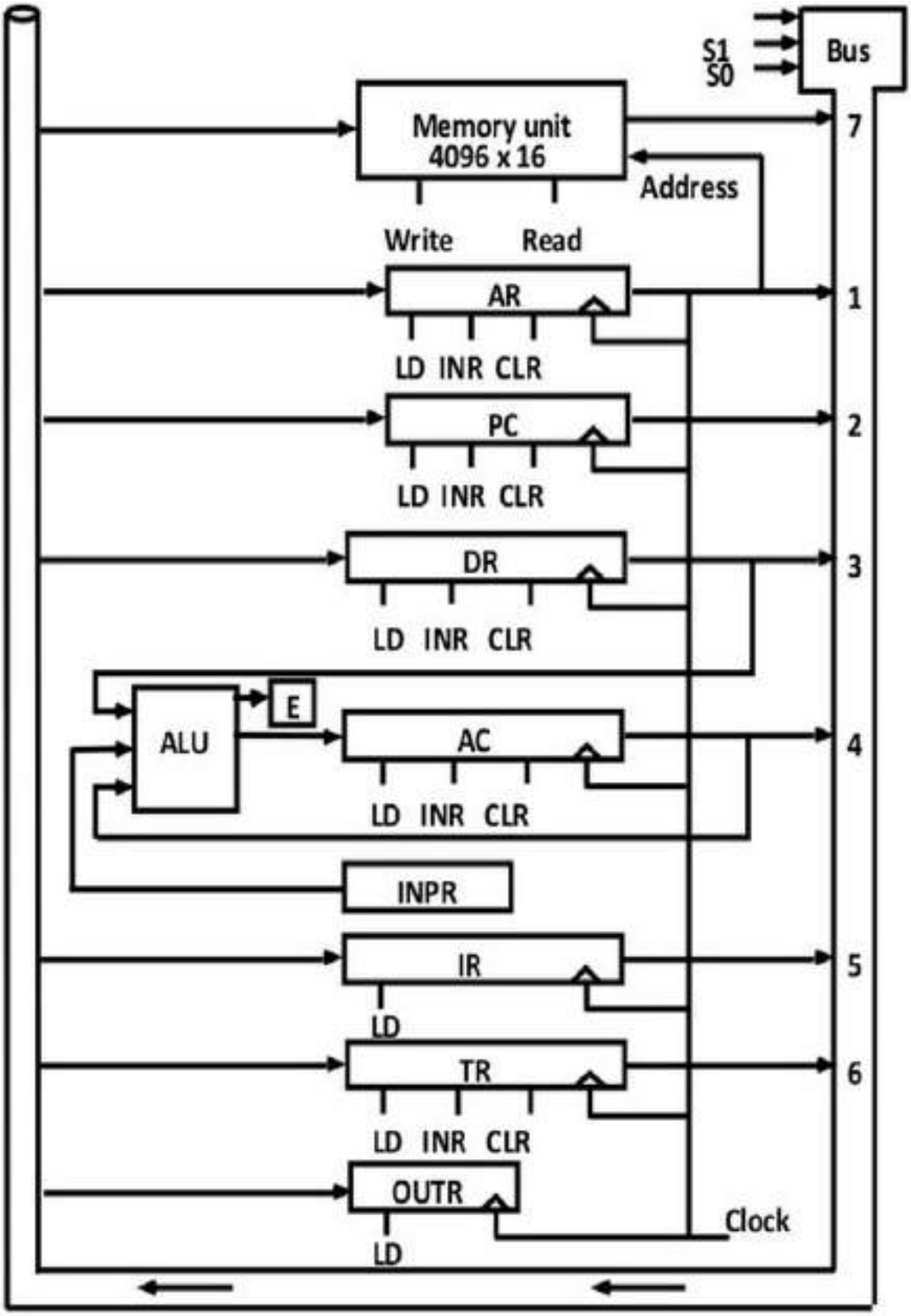


Figure 5-4. The connections of registers and memory of the basic computer to a common bus system.

Program Counter (PC): Program counter has 12 bits and it holds the address of the next instruction to be read from memory after the current execution is executed. When selection inputs $S_2S_1S_0$ = 010 is applied to the bus, the program counter (PC) receives or transfers address from or to the bus when its LD input is enable. The address is incremented or clear by the inputs INR or CLR.

Data Register (DR): The register DR consists of 16-bits and memory operands (data). This register contains the data to be written into memory or receives the data read from memory. When selection inputs $S_2S_1S_0$ = 011 is applied to the bus, the data register DR receives or transfers data from or to the bus when its LD input is enable. The data is incremented or clear by the inputs INR or CLR.

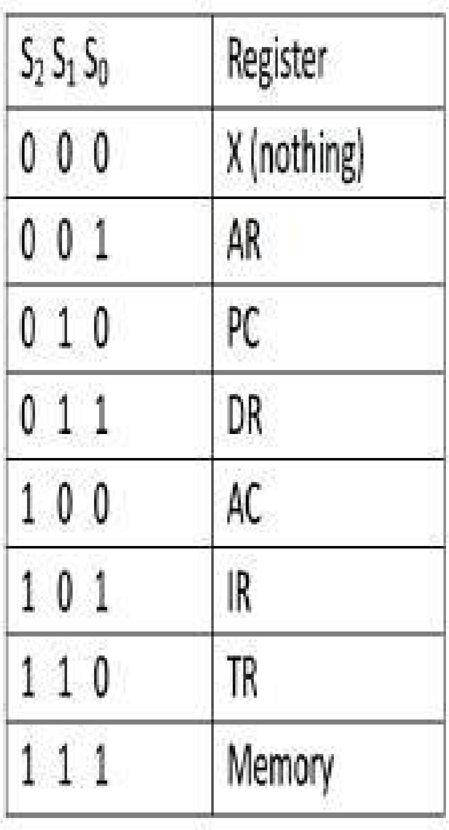
Accumulator (AC): The processor register AC consists of 16 bits. The 16-bit inputs to the Adder / logic circuit come from the outputs of AC. They are used to implement register micro operation such as complement and shift the contents of AC. The results of these micro operations are again transferred to AC. So an accumulator is a register in which intermediate arithmetic and logic results are stored. When selection inputs $S_2S_1S_0$ = 100 is applied to the bus, the processor register AC receives or transfers its data to the bus by enabling the LD input of DR, it transfers the contents of DR through the adder / logic circuit into AC when its LD input is enable. The data of AC is incremented or clear by the inputs INR or CLR.

Instruction Register (IR): The instruction register consists of 16-bits. The purpose of the instruction register is to hold a copy of the instruction which the processor is to execute. The instruction read from memory is placed in the IR. When selection inputs $S_2S_1S_0$ = 101 is applied to the bus, the instruction register IR receives or transfers instruction code from or to the bus when its LD input is enable.

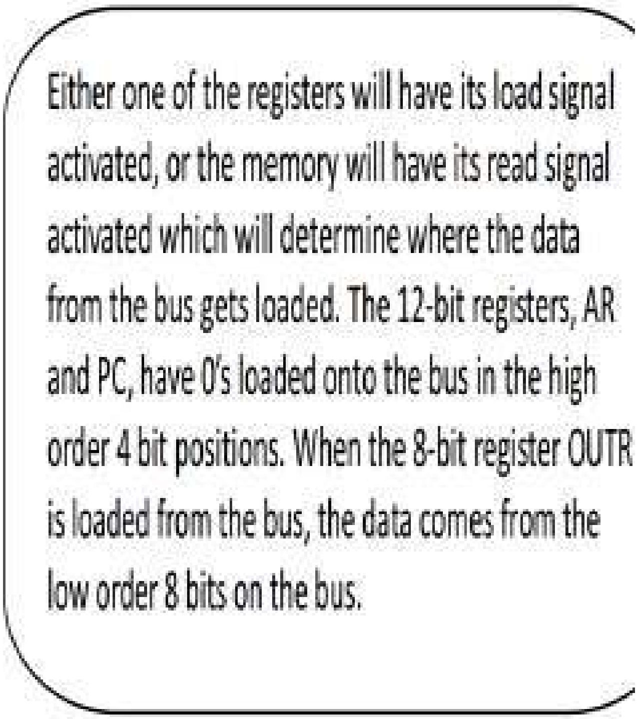
Temporary Register (TR): Temporary registers have 16 bits. It provides temporary storage of variables or results. When selection inputs $S_2S_1S_0$ = 111 is applied to the bus, the temporary register TR receives or transfers temporary data from or to the bus when its LD input is enable. The data is incremented or clear by the inputs INR or CLR.

Input Register (INPR): The Input Register INPR consists of 8-bits and hold alphanumeric input information. The serial information from the input device is shifted into input of 8-bit register INPR. When LD input of AC is an enable, the 8-bit information of INPR is transferred to the AC via Adder/logic circuit.

Output Register (OUTR): The output OUTR receives information from AC and transfers it to the output device.



$S_2 S_1 S_0$	Register
0 0 0	X (nothing)
0 0 1	AR
0 1 0	PC
0 1 1	DR
1 0 0	AC
1 0 1	IR
1 1 0	TR
1 1 1	Memory



Either one of the registers will have its load signal activated, or the memory will have its read signal activated which will determine where the data from the bus gets loaded. The 12-bit registers, AR and PC, have 0's loaded onto the bus in the high order 4 bit positions. When the 8-bit register OUTR is loaded from the bus, the data comes from the low order 8 bits on the bus.

10. Computer Instructions:

→The Basic Computer has 3 instruction code formats as shown in **Figure5-5**.

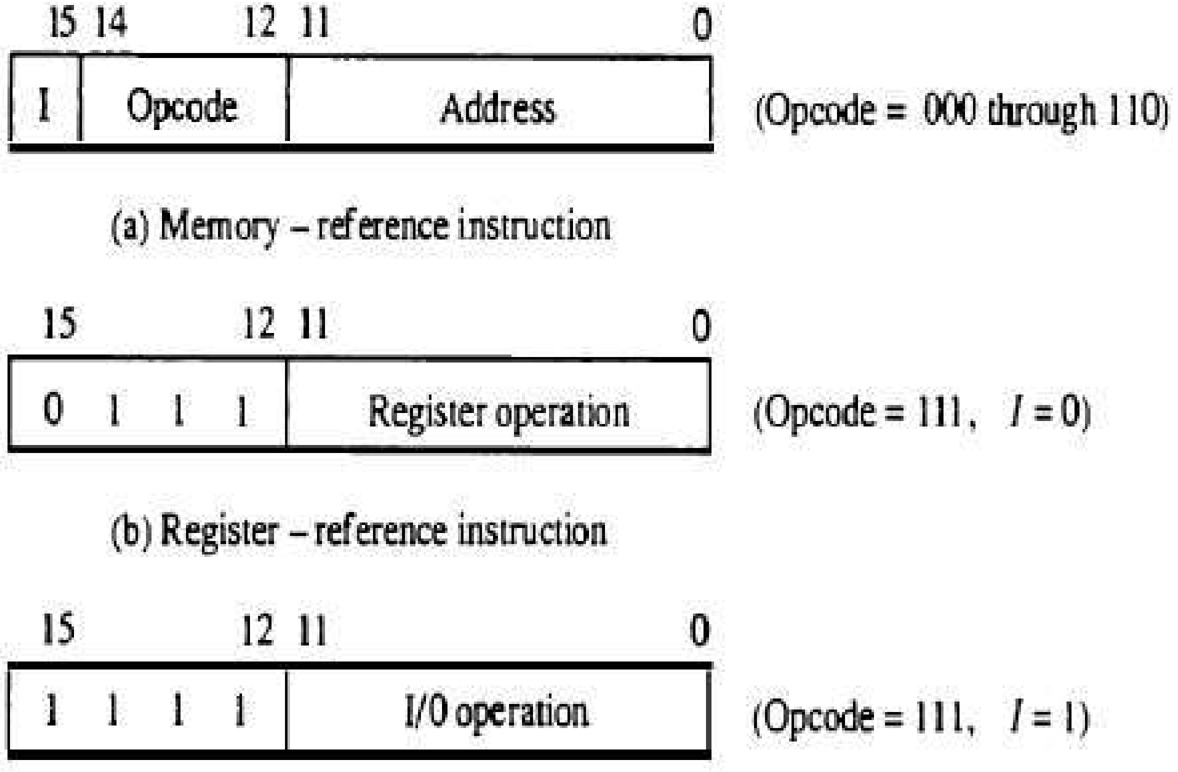
→Each format has 16 bits. The operation code (op-code) of the instruction contains 3 bits and the meaning of the remaining 13 bits depends on the operation code encountered.

→A **Memory reference instruction** uses 12 bits to specify an address and one bit to specify the addressing mode *I*. Addressing mode *I* is equal to 0 for direct address and to 1 for indirect address.

→The **register reference instruction** are recognized by operation code 111 with a 0 in left most bit (Bit 15) of the instruction. The 12 bits are used in to specify the operation done with AC register.

→**Input-Output instruction** is recognized by operation code 111 with a 1 in the left most bit of the instruction. The remaining 12 bits are used to specify the type of input –output operation or test performed.

Figure 5-5 Basic computer instruction formats.



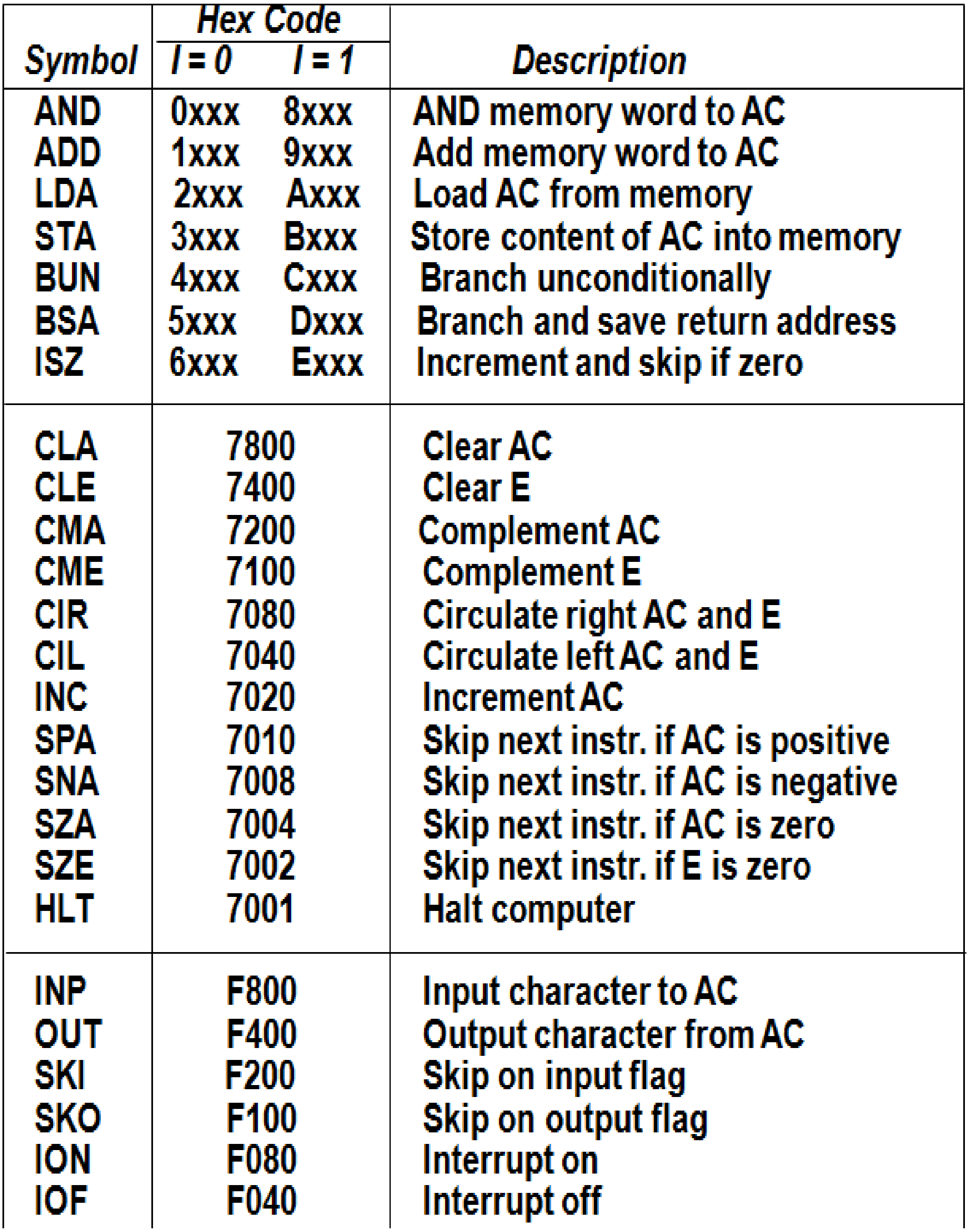
(a) Memory – reference instruction

(b) Register – reference instruction

(c) Input – output instruction

→The type of instruction is recognized by the computer control from the four bits in positions 12 through 15 of the instruction. If the three op-code bits in positions 12 through 14 are not equal to 111, the instruction is a **memory-reference type** and the bit in position 15 is taken as the addressing mode *I*. If the 3-bit op-code is equal to 111, control then inspects the bit in position 15. If this bit is 0, the instruction is a **register-reference type**. If the bit is 1, the instruction is an ***input-output type.***

→The instructions for the computer are listed in **Table** ***5-2*.**The hexadecimal is equal to the equivalent number of binary code used for the instruction. When *I*=0, the last four bits of an instruction have a hexadecimal digit equivalent from 0 to 6 since the last bit is 0. When *I*=1, the hexadecimal digit equivalent of the last four bits of the instruction ranges from 8 to E since the last bit is 1



Symbol	*Hex Code*		*Description*
	I = 0	*I = 1*	
AND	0xxx	8xxx	AND memory word to AC
ADD	1xxx	9xxx	Add memory word to AC
LDA	2xxx	Axxx	Load AC from memory
STA	3xxx	Bxxx	Store content of AC into memory
BUN	4xxx	Cxxx	Branch unconditionally
BSA	5xxx	Dxxx	Branch and save return address
ISZ	6xxx	Exxx	Increment and skip if zero
CLA	7800		Clear AC
CLE	7400		Clear E
CMA	7200		Complement AC
CME	7100		Complement E
CIR	7080		Circulate right AC and E
CIL	7040		Circulate left AC and E
INC	7020		Increment AC
SPA	7010		Skip next instr. if AC is positive
SNA	7008		Skip next instr. if AC is negative
SZA	7004		Skip next instr. if AC is zero
SZE	7002		Skip next instr. if E is zero
HLT	7001		Halt computer
INP	F800		Input character to AC
OUT	F400		Output character from AC
SKI	F200		Skip on input flag
SKO	F100		Skip on output flag
ION	F080		Interrupt on
IOF	F040		Interrupt off

TABLE 5-2 Basic Computer instructions.

Note: The presented code is for any instruction that has 16 bits. The xxx represents don't care (any data for the first 12 bits). Example 7002 for is a hexadecimal code equivalent to 0111 0000 0000 0010 which means B1 (Bit 1) is set to 1 and the rest of the first 12 bits are set to zeros.

Timing andControl:

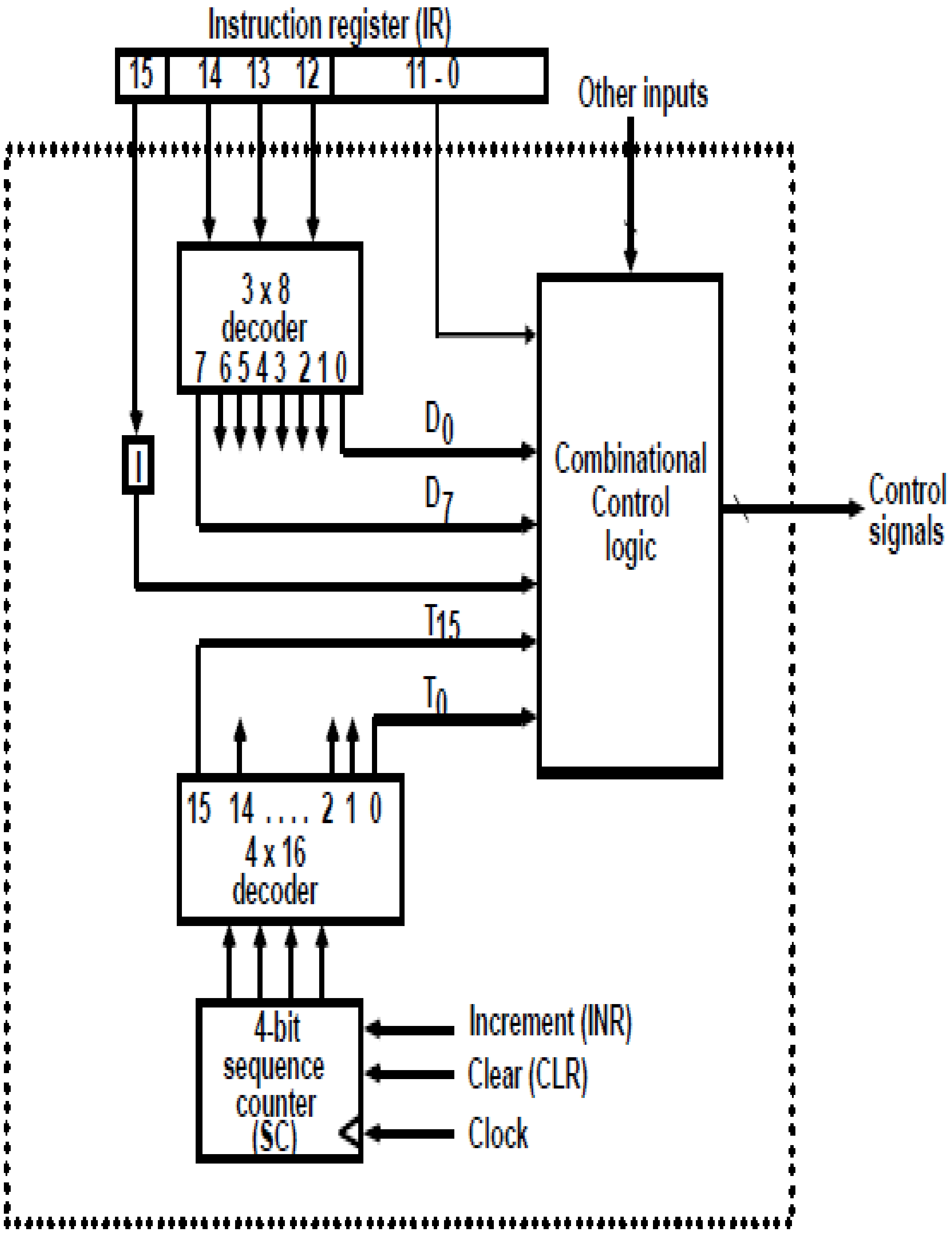


Figure 5-6 Control unit of basic computer.

→The timing for all registers is controlled by a master clock generator. The *clock pulses* are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit. The *clock pulses* do not change the state of a register unless it is enabled by a control signal. The control signals are generated in the control unit and provide control inputs for the multiplexers in the common bus, control inputs in processor registers, and micro-operations for the accumulator.

→There are two major types of control organization: hardwired control and programmed control. In **Hardwired organization,** the control logic is made up of sequential and combinational circuits to generate the control signals.

→In **Micro-programmed Control,** the control memory on the processor contains micro-programs that activate the necessary control signals.

→The block diagram of control unit of the basic computer is shown in **Figure 5-6.**It consists of 2 decoders, a 4-bit sequence counter, Instruction Register and a number control logic gates.

→An instruction read from memory is placed in the instruction register (IR). The instruction register is divided into three parts: the *I* bit, operation code, and address part. First 12-bits (0-11) to specify an address, next 3-bits specify the operation code (op-code) field of the instruction and last left most bit specify the addressing mode *I*. $I = 0$ for direct address $I = 1$ for indirect address.

→First 12-bits (0-11) are applied to the control logic gates. The operation code bits (12 – 14) are decoded with a 3 x 8 decoder. The eight outputs (D_0 through D_7) from a decoder go to the control logic gates to perform specific operation. Last bit 15 is transferred to a *I* flip-flop designated by symbol *I*.

→The 4-bit sequence counter SC can count in binary from 0 through 15. The counter output is decoded into 16 timing pulses T_0 through T_{15}. The sequence counter can be incremented by INR input or clear by CLR input synchronously.

→**For example:**

Consider the case where SC is incremented to provide timing signalsT0, T1, T2, T3, and T4 in sequence. At time T4, SC is cleared to 0 if decoder output D3 is active. This is expressed symbolically by the statement:

D3 T4: SC ← 0

→The timing diagram of **Figure 5-7**shows the time relationship of the control signals. The sequence counter SC responds to the positive transition of the clock. Initially, the CLR input of SC is active. The first positive transition of the clock clears SC to 0, which in turn activates the timing signal T_0 out of the decoder. T_0 is active during one clock cycle. The positive clock transition labeled connected T_0 in the diagram will trigger only those registers whose control inputs are transition, to timing signal T_o. SC is incremented with every positive clock transition unless its CLR input is active. This produces the sequence of timing signals T_0, T_1, T_2, T_3, T_4, and so on. If it is not cleared, the timing signals will continue with T_5, T_6, up to T_{15} and back to T_0.

- Example: $T_0, T_1, T_2, T_3, T_4, T_0, T_1, . . .$

Assume: At time T_4, SC is cleared to 0 if decoder output D3 is active.

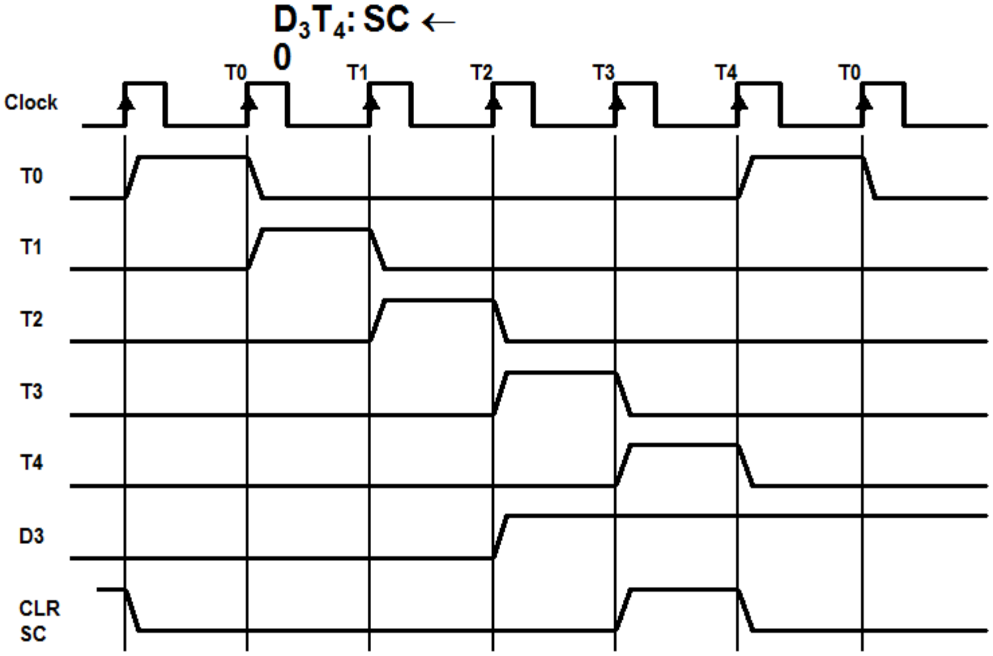


Figure 5-7 Example of control timing signals.

→The last three waveforms in **Figure 5-7** show how SC is cleared when $D_3T_4 = I$. Output D_3 from the operation decoder becomes active at the end of timing signal T_2. When timing signal T4 becomes active, the output of the AND gate that implements the control function D_3T_4 becomes active. This signal is applied to the CLR input of SC. On the next positive clock transition (the one marked T, in the diagram) the counter is cleared to 0. This causes the timing signal T_0 to become active instead of T_5 that would have been active if SC were incremented instead of cleared.

11. Instruction Cycle:

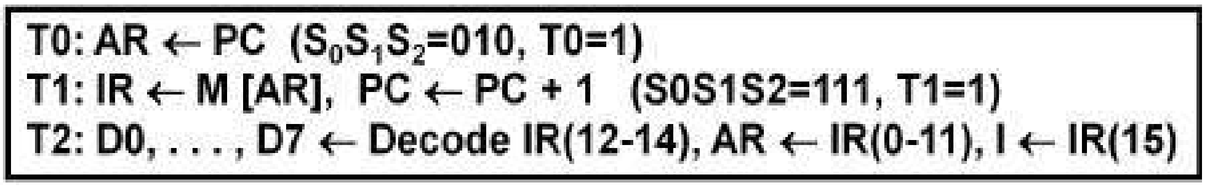
→A program residing in the memory unit of the computer consists of a sequence of instructions. The program is executed in the computer by going through a cycle for each instruction. Each instruction cycle in turn is subdivided into a sequence of sub-cycles or phases. In the basic computer each instruction 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. **This process continues indefinitely unless a HALT instruction is encountered**.

Fetch and Decode:

→Initially, the program counter PC is loaded with the address of the first instruction in the program. The sequence counter SC is cleared to 0, providing a decoded timing signal T_0. After each clock pulse, SC is incremented by one, so that the timing signals go through a sequence T_0, T_1, T_2, and so on. The micro-operations for the fetch and decode phases can be specified by the following register transfer statements.



T0: AR ← PC ($S_0S_1S_2$=010, T0=1)
T1: IR ← M [AR], PC ← PC + 1 (S0S1S2=111, T1=1)
T2: D0, . . . , D7 ← Decode IR(12-14), AR ← IR(0-11), I ← IR(15)

→Since only AR is connected to the address inputs of memory, the address of instruction is transferred from PC to AR.

→**Figure 5-8** shows how the first two register transfer statements are implemented in the bus system. 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 $S_2S_1S_0$ equal to 010.
2. Transfer the content of the bus to AR by enabling the LD input of AR.

→The next clock transition initiates the transfer from PC to AR since $T_1 = 1$. In order to implement the second statement:

T_1: IR ← M [AR], PC ← PC + 1

it is necessary to use timing signal T_1 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 $S_2S_1S_0 = 111$.
3. Transfer the content o f the bus t o IR b y 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 $T_1 = 1$.

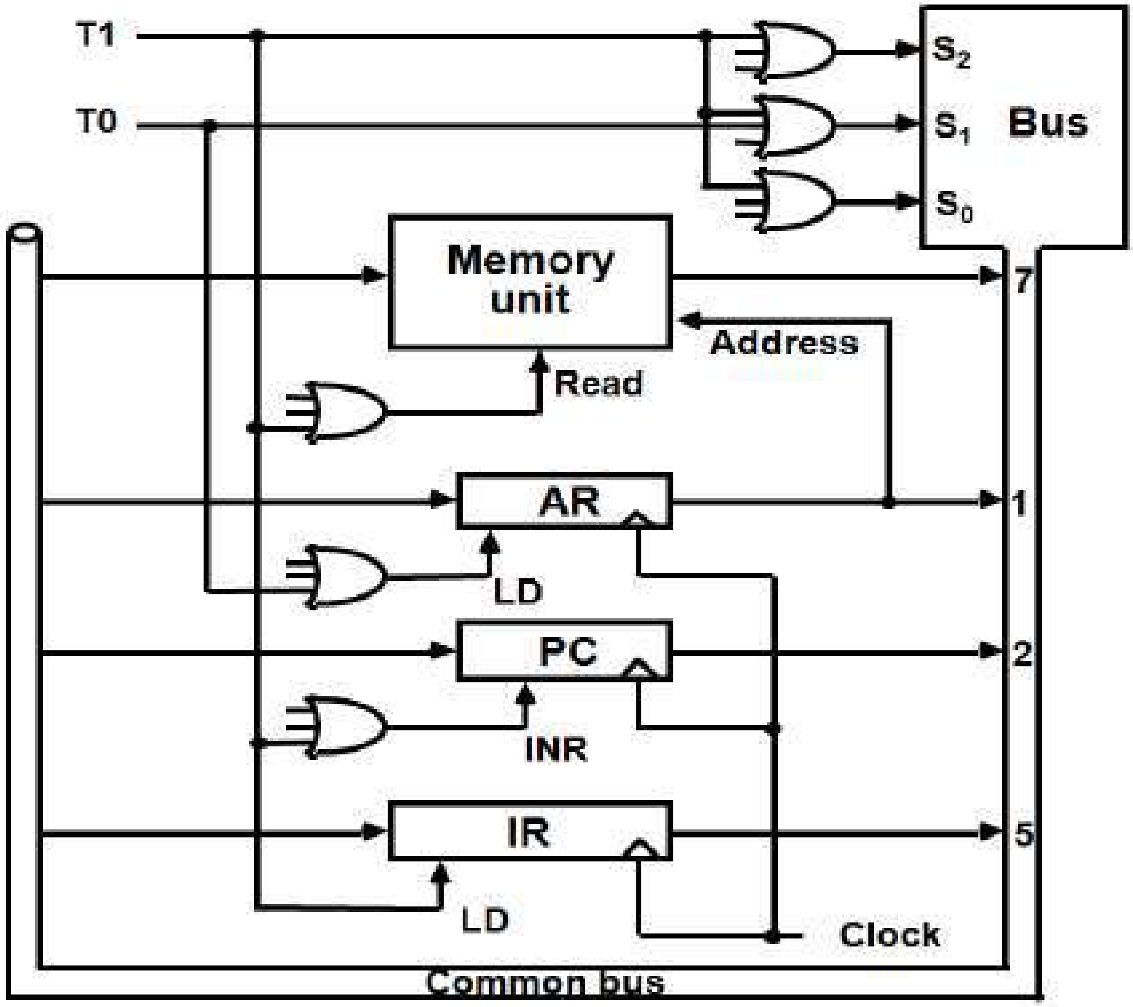


Figure 5-8 Register transfers for the fetch phase.

Determine the Type of Instruction:

→The timing signal that is active after the decoding is T_3. During time T_3 the control unit determines the type of instruction that was just read from memory. The flowchart of **Figure 5-9** presents an initial configuration for the instruction cycle and shows how the control determines the instruction type after the decoding.

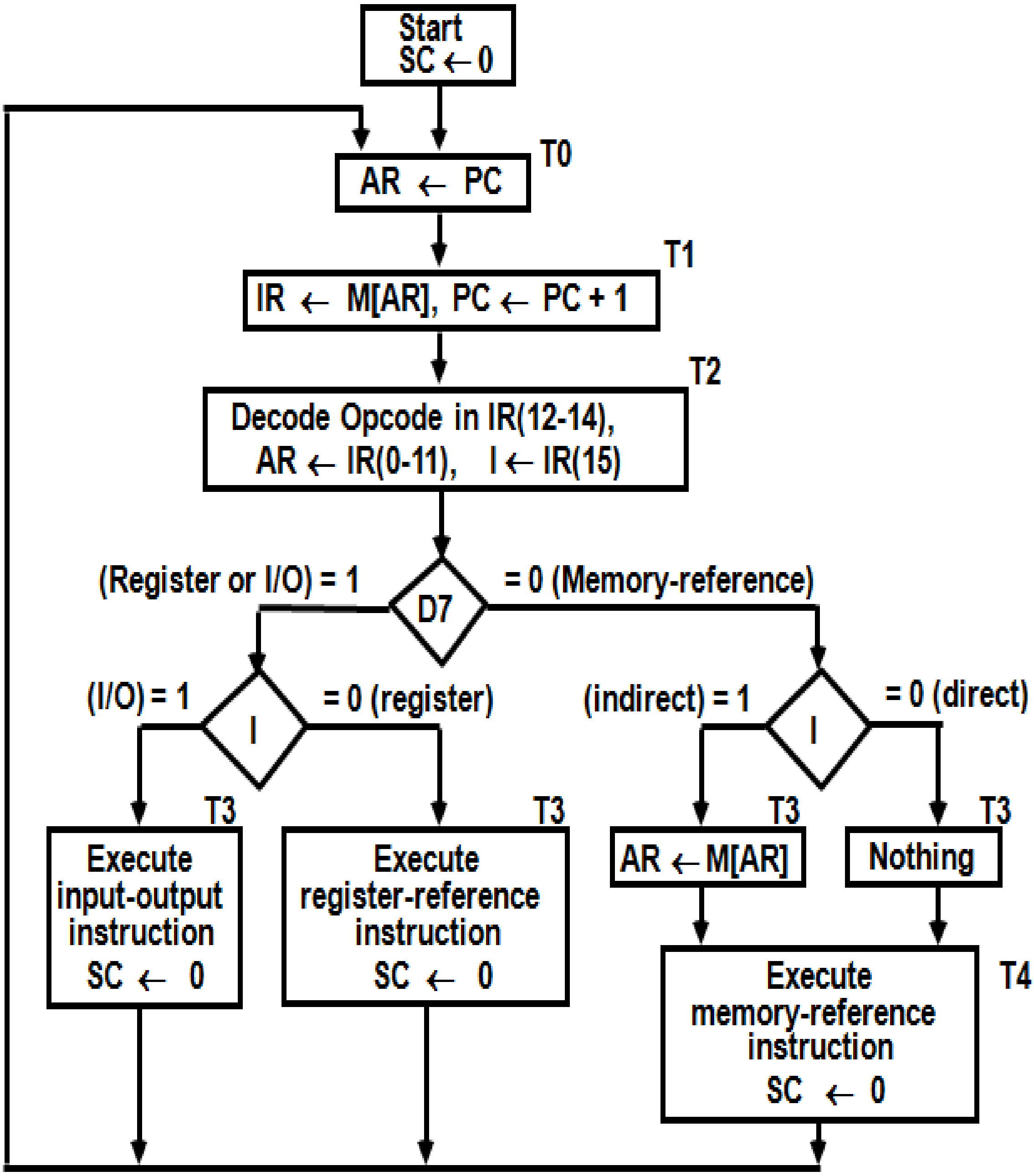
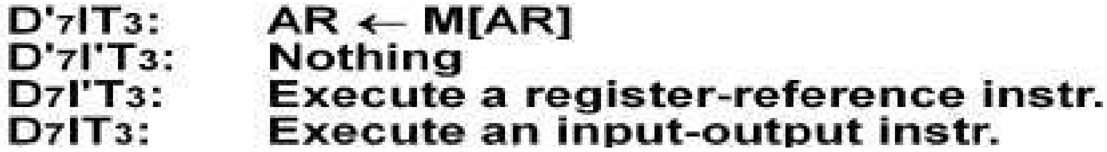

Start
SC ← 0
T0
AR ← PC
T1
IR ← M[AR], PC ← PC + 1
T2
Decode Opcode in IR(12-14),
AR ← IR(0-11), I ← IR(15)
(Register or I/O) = 1
D7
= 0 (Memory-reference)
(I/O) = 1
I
= 0 (register)
(indirect) = 1
I
= 0 (direct)
T3
Execute input-output instruction SC ← 0
T3
Execute register-reference instruction SC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

Figure 5-9. Flowchart for instruction cycle (initial configuration).

→Decoder output D_7 is equal to 1 if the operation code is equal to binary 111. If D_7 = 1, the instruction must be a register-reference or input-output type. If D_7 = 0, the operation code must be one of the other seven values 000 through 110, specifying a memory-reference instruction. Control then inspects the value of the first bit of the instruction, which is now available in flip-flop I. If D_7 = 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

AR ← M [AR]

→The three instruction types are subdivided into four separate paths. The selected operation is activated with the clock transition associated with timing signal T_3. This can be symbolized as follows:



D'_7IT_3:	AR ← M[AR]
$D'_7I'T_3$:	Nothing
$D_7I'T_3$:	Execute a register-reference instr.
D_7IT_3:	Execute an input-output instr.

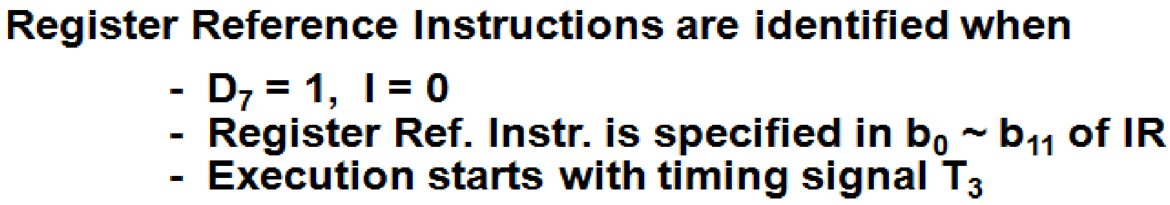
TYPES OF INSTRUCTIONS:

→The basic computer has three 16-bit instruction code formats:

- **Memory Reference Instructions.**
- **Register Reference Instructions.**
- **Input / Output Instructions.**

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 12 instructions. These 12 bits are available in IR(0-11). They were also transferred to AR during time T_2.



Register Reference Instructions are identified when

- $D_7 = 1$, $I = 0$
- Register Ref. Instr. is specified in $b_0 \sim b_{11}$ of IR
- Execution starts with timing signal T_3

→The control functions and micro-operations for the register-reference instructions are listed in **Table 5-3.**

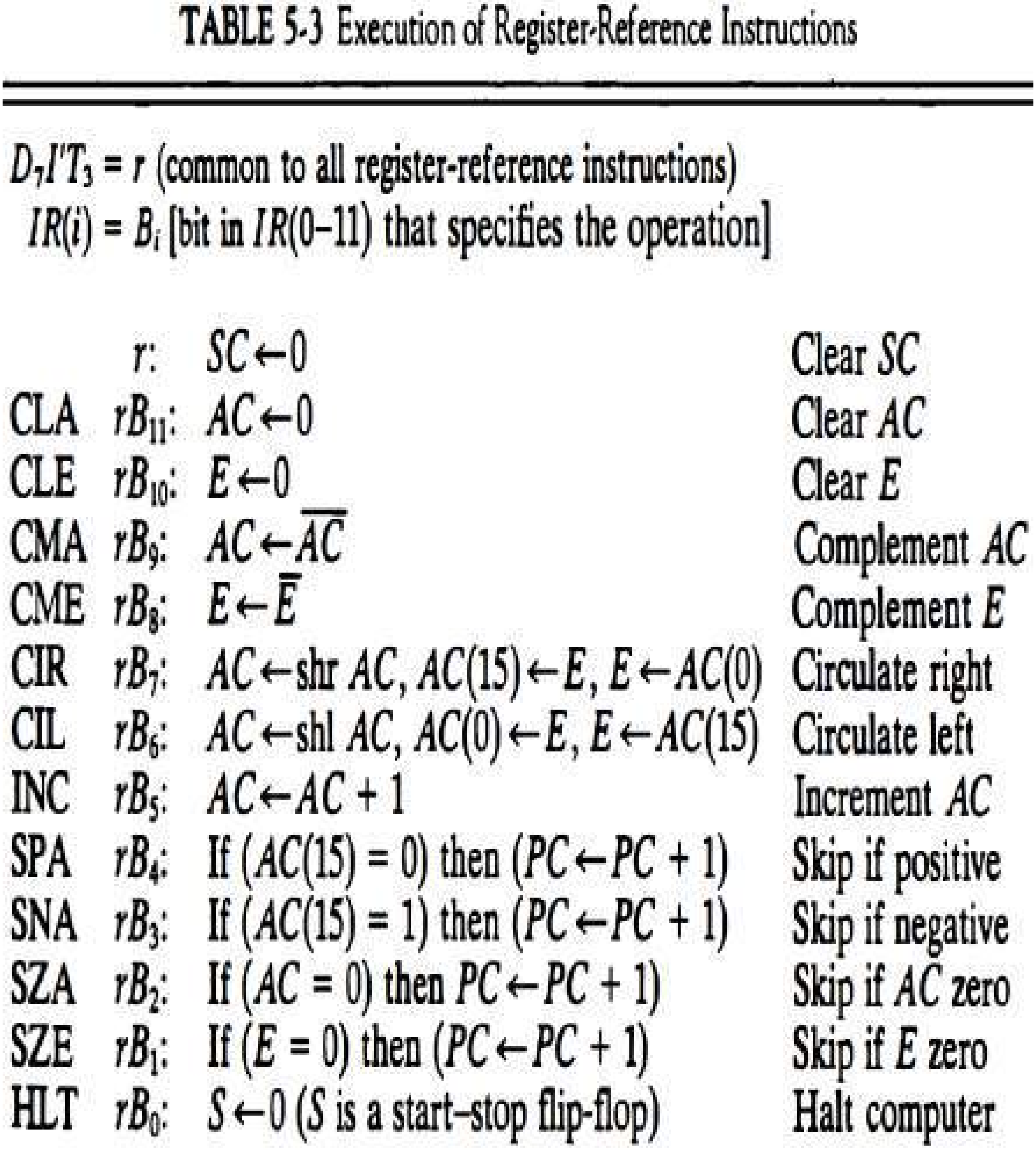


TABLE 5-3 Execution of Register-Reference Instructions

$D_7I'T_3 = r$ (common to all register-reference instructions)

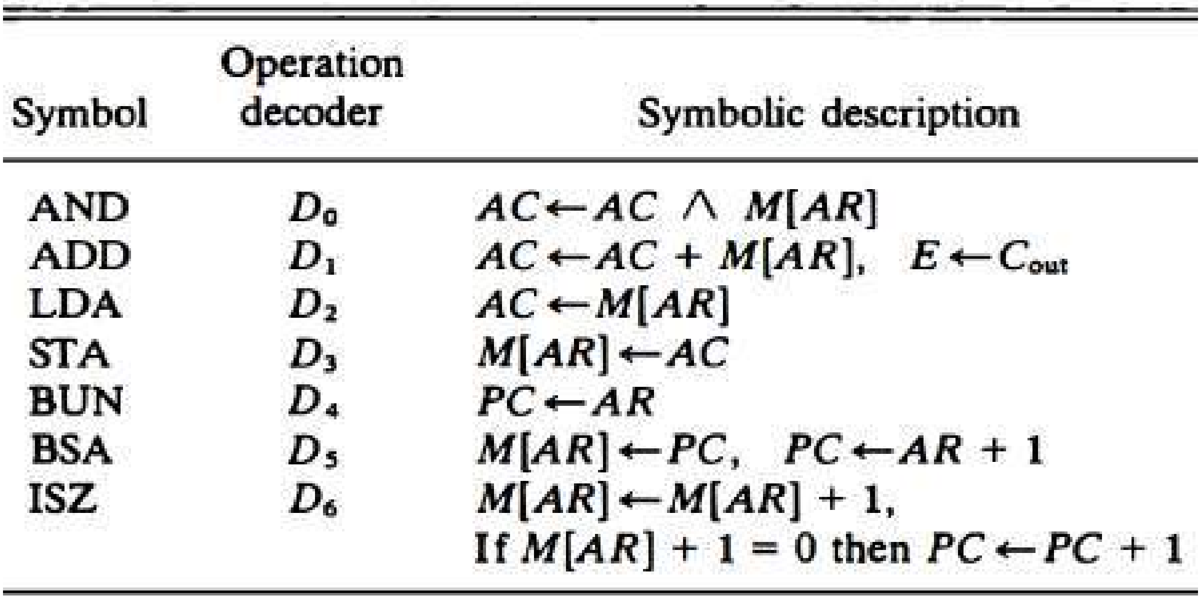
$IR(i) = B_i$ [bit in $IR(0\text{–}11)$ that specifies the operation]

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

12. Memory Reference Instructions:

→**Table 5-4** lists the seven memory-reference instructions. The decoded output D; for i = 0, 1, 2, 3, 4, 5, and 6 from the operation decoder that belongs to each instruction is included in the table. The **effective address** of the instruction is in the address register AR and was placed there during timing signal T_2 when I = 0, or during timing signal T_3 when I = 1. The execution of the memory-reference instructions starts with timing signal T_4.

TABLE 5-4 Memory-Reference Instructions

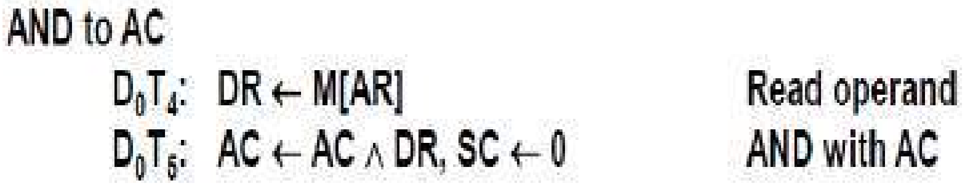


Symbol	Operation decoder	Symbolic description
AND	D_0	$AC \leftarrow AC \wedge M[AR]$
ADD	D_1	$AC \leftarrow AC + M[AR], \quad E \leftarrow C_{out}$
LDA	D_2	$AC \leftarrow M[AR]$
STA	D_3	$M[AR] \leftarrow AC$
BUN	D_4	$PC \leftarrow AR$
BSA	D_5	$M[AR] \leftarrow PC, \quad PC \leftarrow AR + 1$
ISZ	D_6	$M[AR] \leftarrow M[AR] + 1$, If $M[AR] + 1 = 0$ then $PC \leftarrow PC + 1$

→All memory-reference instructions have to wait until T_4 so that the timing is the same whether the operand is direct or indirect.

AND to AC:

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



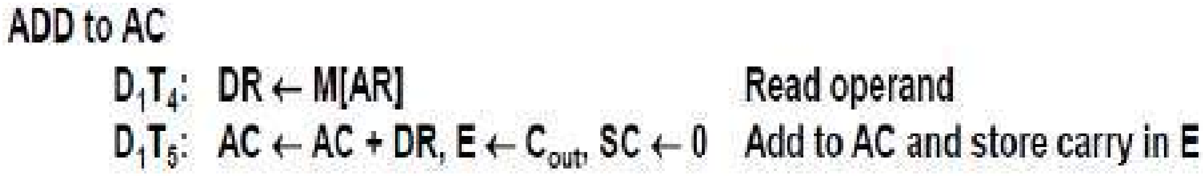
AND to AC

D_0T_4: $DR \leftarrow M[AR]$ Read operand

D_0T_5: $AC \leftarrow AC \wedge DR, SC \leftarrow 0$ AND with AC

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 micro-operations needed to execute this instruction are:



ADD to AC

D_1T_4: $DR \leftarrow M[AR]$ Read operand

D_1T_5: $AC \leftarrow AC + DR, E \leftarrow C_{out}, SC \leftarrow 0$ Add to AC and store carry in E

LDA: Load to AC:

→This instruction transfers the memory word specified by the effective address to AC. The micro-operations needed to execute this instruction are:

LDA: Load to AC

D_2T_4: $DR \leftarrow M[AR]$ //Read operand

D_2T_5: $AC \leftarrow DR, SC \leftarrow 0$ //Load AC with DR

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:

STA: Store AC

D_3T_4: $M[AR] \leftarrow AC, SC \leftarrow 0$ // store data into memory location

BUN: Branch Unconditionally:

→This instruction transfers the program to the instruction specified by the effective address. Remember that 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 sequence and we say that the program branches (or jumps) unconditionally. The instruction is executed with one micro-operation:

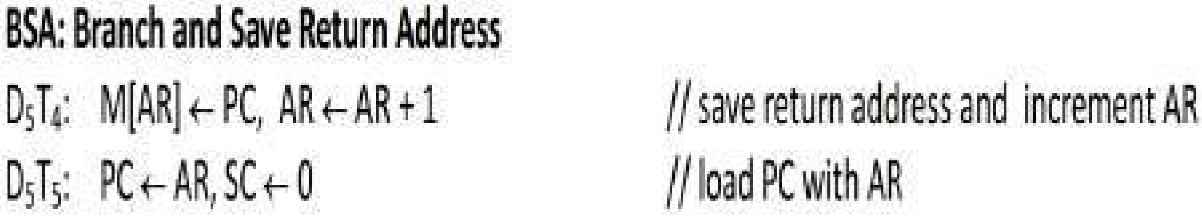
BUN: Branch Unconditionally

D_4T_4: $PC \leftarrow AR, SC \leftarrow 0$ //Branch to specified address

BSA: Branch and Save Return Address:

→BSA instruction performs the function usually referred to as a subroutine call.

→BSA is used to branch to a subprogram. This requires saving the return address, which is saved at the operand’s effective address with the subprogram beginning one word later in memory:

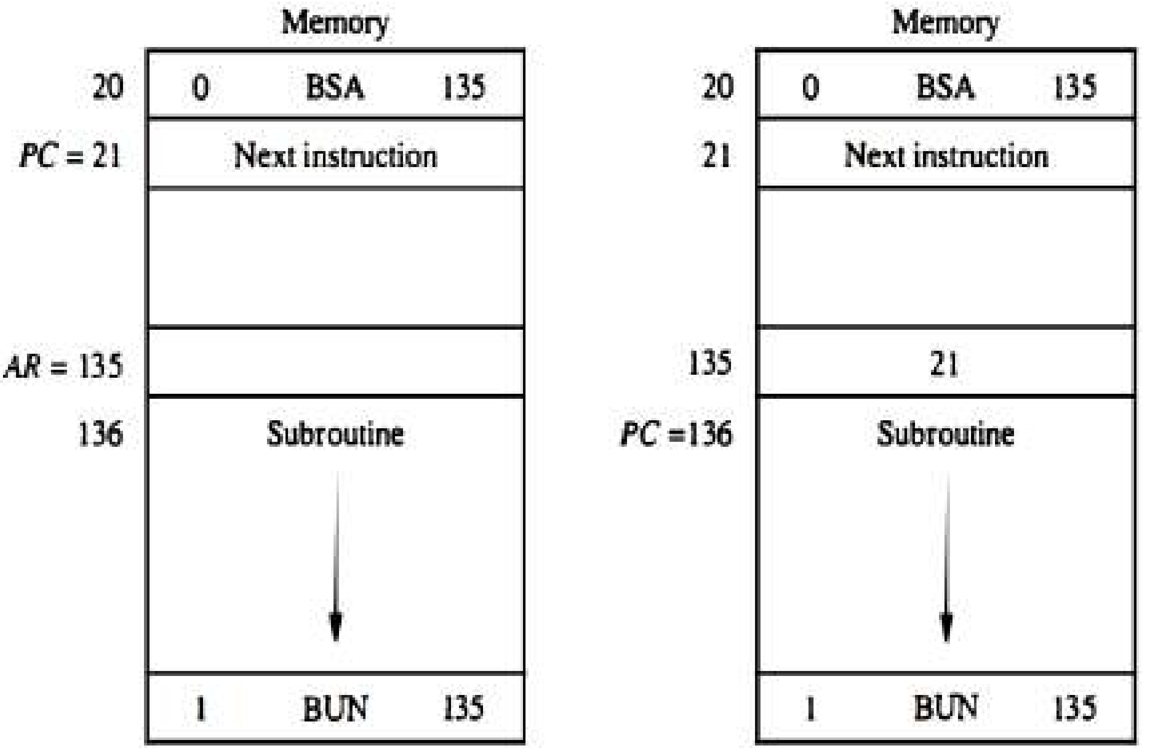


BSA: Branch and Save Return Address

D_5T_4: $M[AR] \leftarrow PC,\ AR \leftarrow AR + 1$ // save return address and increment AR

D_5T_5: $PC \leftarrow AR, SC \leftarrow 0$ // load PC with AR

Figure 5-10 Example of BSA instruction execution.



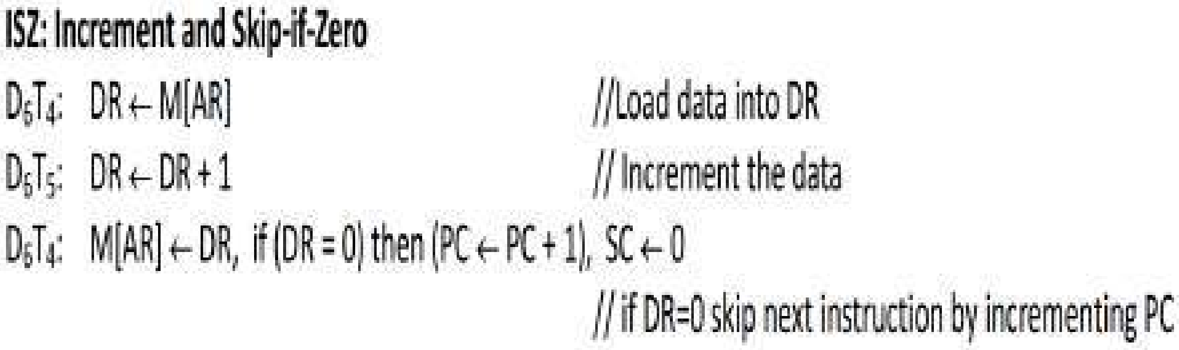
(a) Memory, PC, and AR at time T_4

(b) Memory and PC after execution

ISZ: Increment and Skip if Zero:

→ISZ skips the next instruction if the operand stored at the effective address is 0. This requires that the PC incremented, which cannot be done directly:

This is done with the following sequence of micro-operations:



ISZ: Increment and Skip-if-Zero

D_6T_4: $DR \leftarrow M[AR]$ //Load data into DR

D_6T_5: $DR \leftarrow DR + 1$ // Increment the data

D_6T_4: $M[AR] \leftarrow DR$, if $(DR = 0)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$

// if DR=0 skip next instruction by incrementing PC

Control Flowchart:

→A flowchart showing all micro-operations for the execution of the seven memory-reference instructions is shown in **Figure 5-11**. The control functions are indicated on top of each box. The micro-operations that are performed during time T_4, T_5, or T, depend on the operation code value. This is indicated in the flowchart by six different paths, one of which the control takes after the instruction is decoded. The sequence counter SC is cleared to 0 with the last timing signal in each case. This causes a transfer of control to timing signal T_0 to start the next instruction cycle.

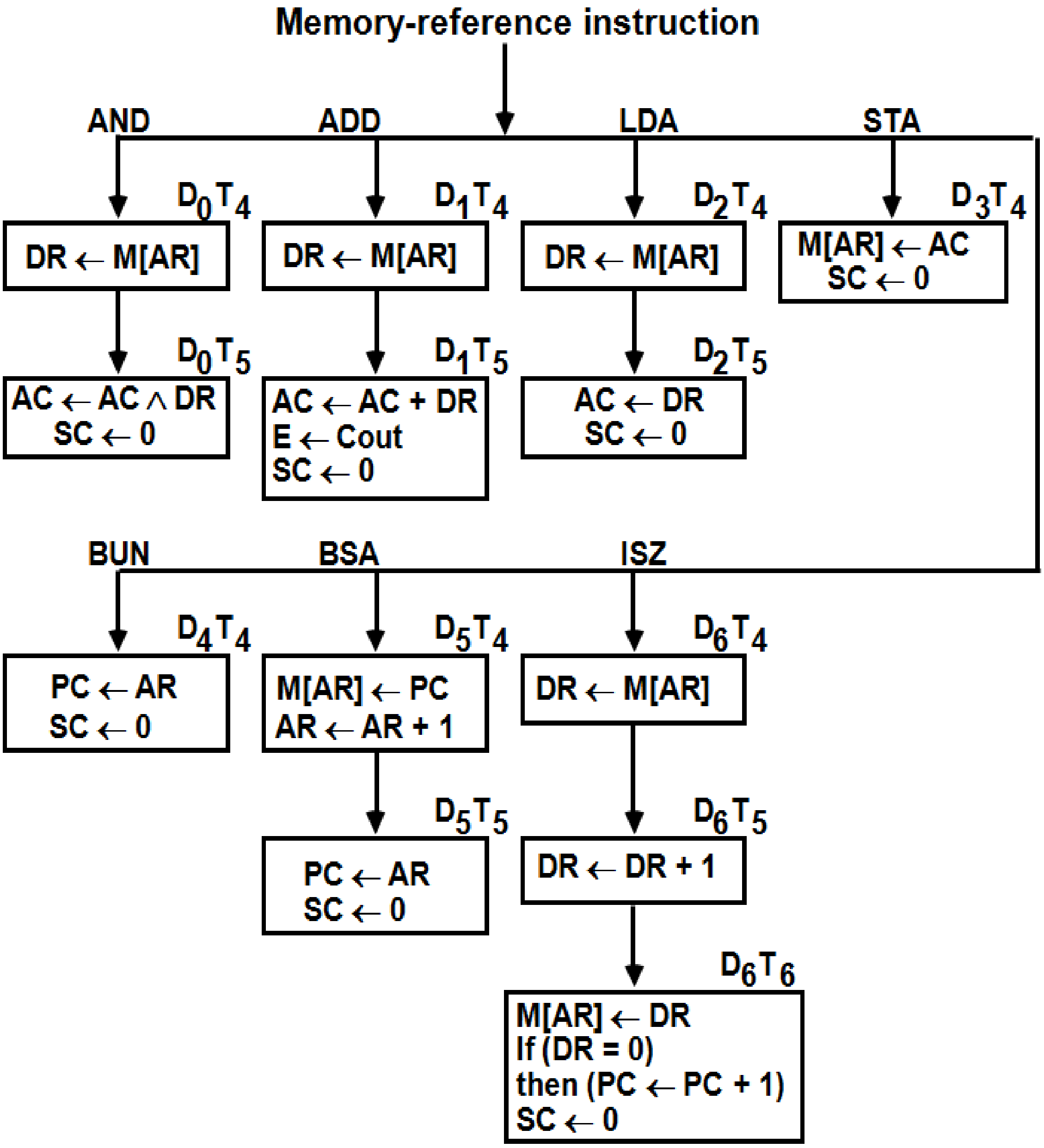


Figure 5-11 Flowchart for memory-reference instructions.

13. Input- Output and Interrupt:

→ In computer, instructions and data stored in memory come from some input device and Computational results must be transmitted to the user through some output device.

Input Output Configuration:

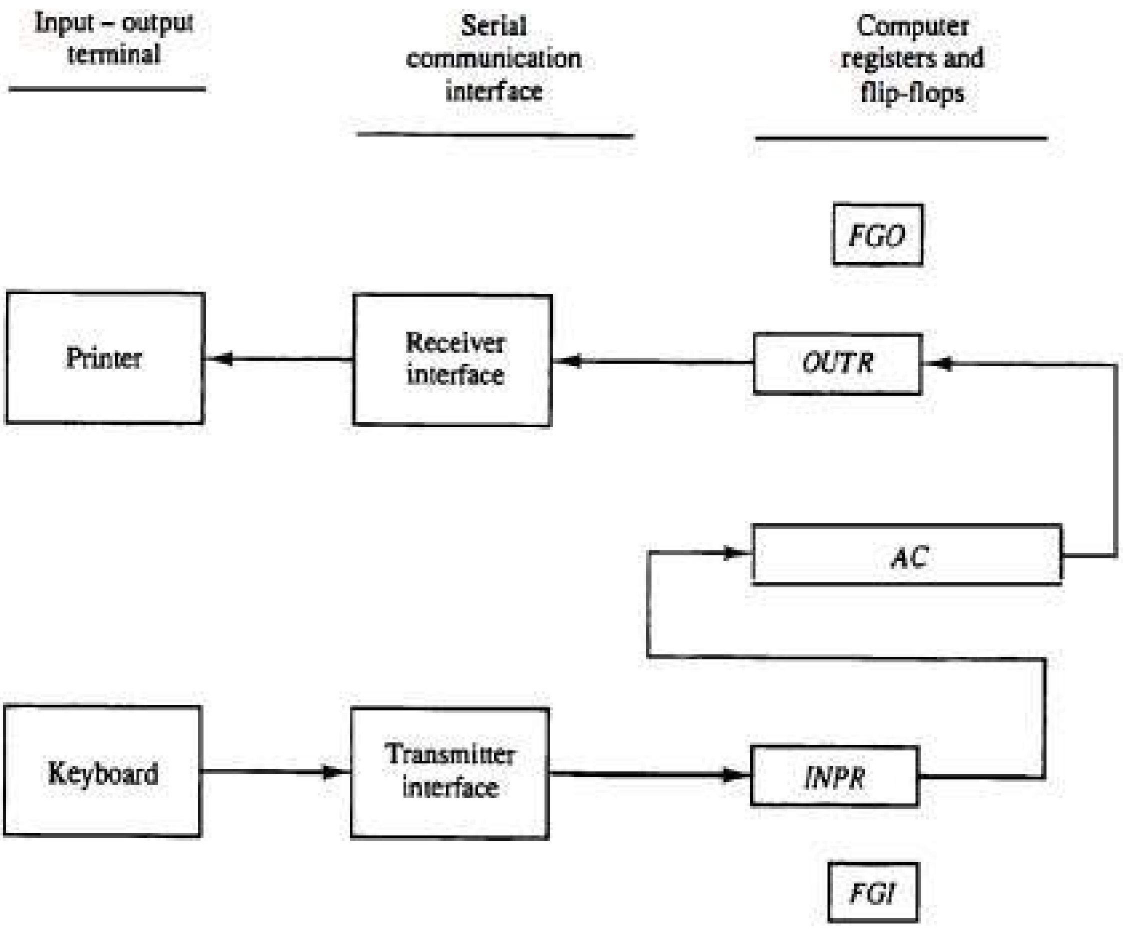
→ The terminal sends and receives 8 bit data converted to serial information and receives serial information and convert it back to parallel 8 bits.

→ The serial info from the keyboard is shifted into the input register INPR.

→ The serial info 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 **Figure 5-12**. The transmitter interface receives serial information from the keyboard and transmits it to INPR. The receiver interface receives information from OUTR and sends it to the printer serially.

Figure 5-12 Input-output configuration.



- **INPR:** Input register - 8 bits
- **OUTR:** Output register- 8 bits
- **FGI:** Input flag - 1 bit (Is a control flip-flop, set to 1 when new information is available)
- **FGO:** Output flag - 1 bit
- **IEN**: Interrupt enable - 1 bit

Input Register:

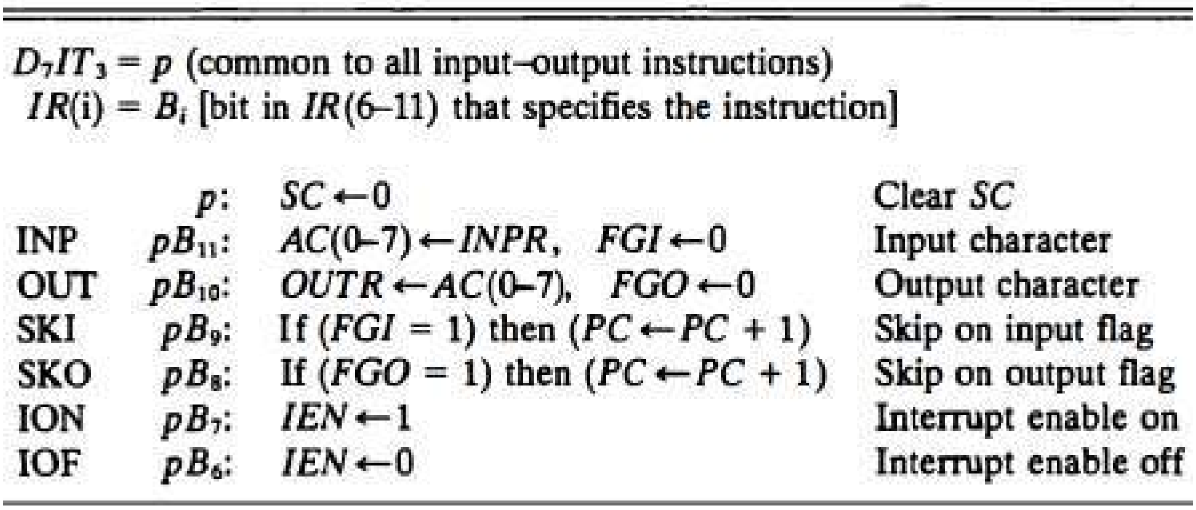
→Scenario1: when a key is struck in the keyboard, an 8-bit alphanumeric code is shifted into INPR and the input flag FGI is set to 1. As long as the flag is set, the information in INPR cannot be changed by striking another key. The control checks the flag bit, if 1, contents of INPR is transferred in parallel to AC and FGI is cleared to 0. Once the flag is cleared, new information can be shifted into INPR by striking another key

Output Register:

→Scenario2: OUTR works similarly but the direction of information flow is reversed. Initially FGO is set to 1. The computer checks the flag bit; if it is 1, the information is transferred in parallel to OUTR and FGO is cleared to 0. The output device accepts the coded information, prints the corresponding character and when operation is completed, it sets FGO to 1.

Input-Output Instructions:

TABLE 5-5 Input-Output Instructions



$D_7IT_3 = p$ (common to all input–output instructions)
$IR(i) = B_i$ [bit in $IR(6–11)$ that specifies the instruction]

	p:	$SC \leftarrow 0$	Clear SC
INP	pB_{11}:	$AC(0–7) \leftarrow INPR, \quad FGI \leftarrow 0$	Input character
OUT	pB_{10}:	$OUTR \leftarrow AC(0–7), \quad FGO \leftarrow 0$	Output character
SKI	pB_9:	If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$	Skip on input flag
SKO	pB_8:	If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$	Skip on output flag
ION	pB_7:	$IEN \leftarrow 1$	Interrupt enable on
IOF	pB_6:	$IEN \leftarrow 0$	Interrupt enable off

→ Input and 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. The control functions and micro-operations for the input-output instructions are listed in **Table 5-5**.

Program Interrupt:

→Open communication only when some data has to be passed is called as an **interrupt.** Interrupts permit other CPU instructions to execute while waiting for I/O to complete. The I/O interface, instead of the CPU, monitors the I/O device. When the interface founds that the I/O device is ready for data transfer, it generates an interrupt request to the CPU.

→Upon detecting an interrupt, the CPU stops momentarily the task it is doing, branches to the service routine to process the data transfer, and then returns to the task it was performing.

→Scenario3: Consider a computer which completes instruction cycle in 1μs. Assume I/O device that can transfer information at the maximum rate of 10 characters /sec. Equivalently, one character every 100000μs. Two instructions are executed when computer checks the flag bit and decides not to transfer information. Which means computer will check the flag 50000 times between each transfer. Computer is wasting time while checking the flag instead of doing some useful processing task.

IEN (Interrupt-enable flip-flop):

→IEN can be set and cleared by instructions. When cleared, the computer cannot be interrupted.

→The way that the interrupt is handled by the computer can be explained by means of the flowchart of **Figure 5-13**.

→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. At the end of the execute phase, control checks the value of R, and if it is equal to 1, it goes to an interrupt cycle instead of an instruction cycle.

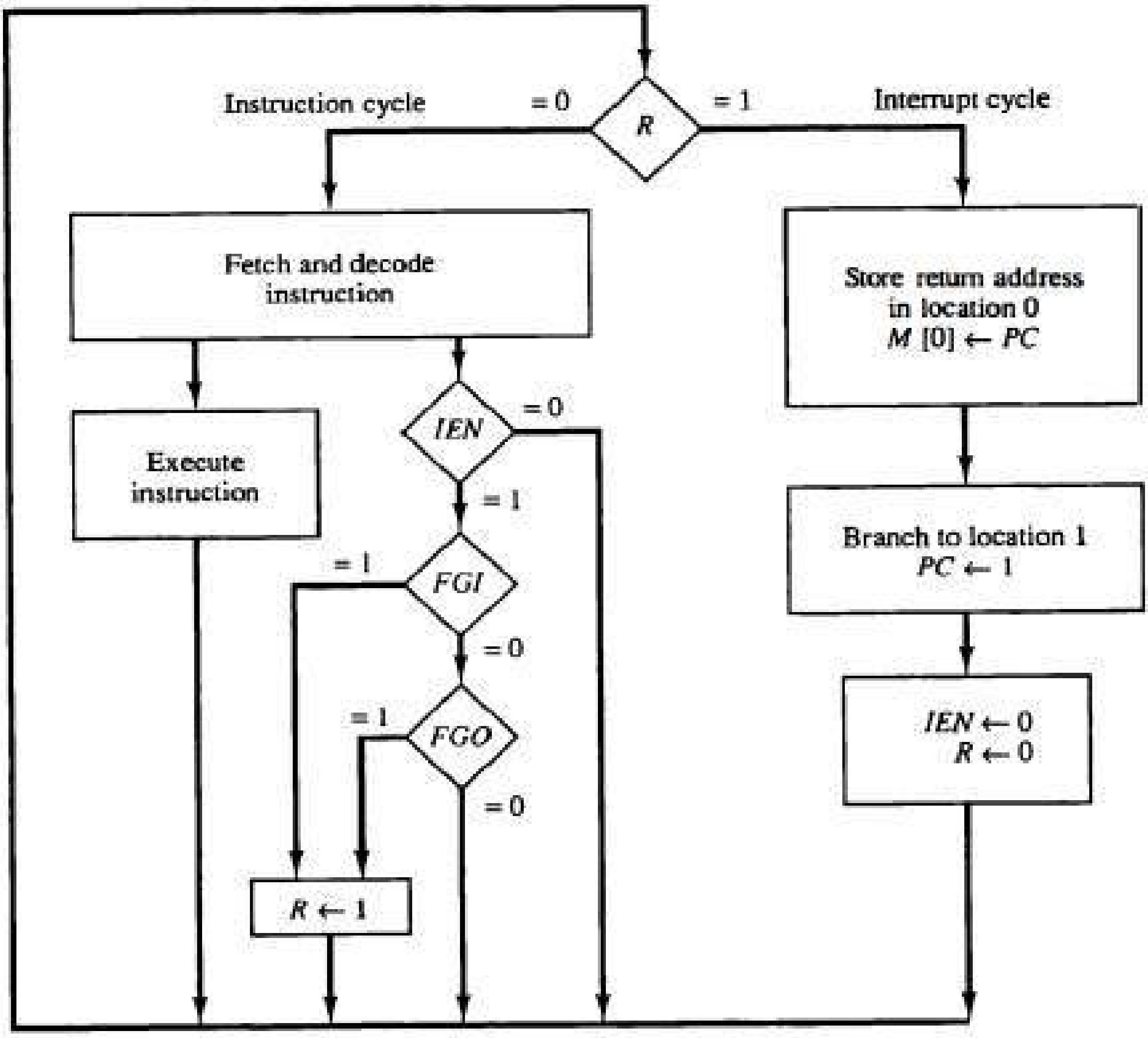


Figure 5-13 Flowchart for interrupt cycle.

Interrupt Cycle:

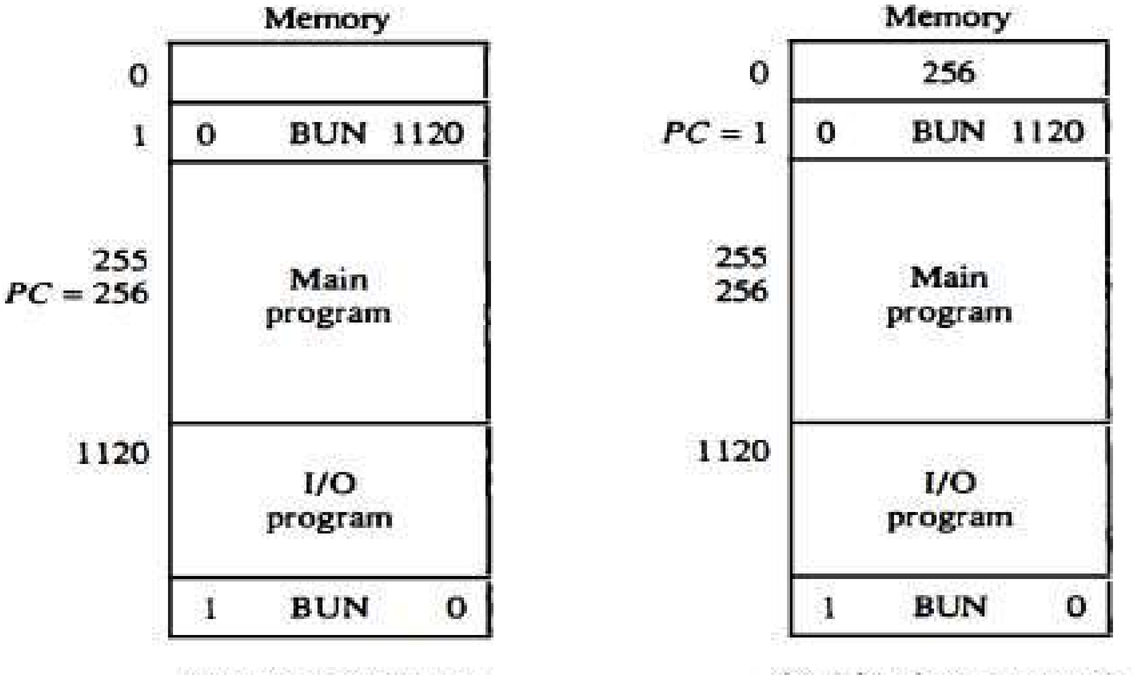
→The interrupt cycle is a hardware implementation of a branch and save return address operation.

→An example that shows what happens during the interrupt cycle is shown in **Figure. 5-14**.

→Suppose that an interrupt occurs and R is set to 1 while the control is executing the instruction at address 255. At this time, the return address 256 is in PC. The programmer has previously placed an input-output service program in memory starting from address 1120 and a BUN 1120 instruction at address 1. This is shown in **Figure 5-14(a).**

→When control reaches timing signal T0 and finds that R = 1, it proceeds with the interrupt cycle. The content of PC (256) is stored in memory location 0, PC is set to 1, and R is cleared to 0. At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1 since this is the content of PC. The branch instruction at address 1 causes the program to transfer to the input-output service program at address 1120. This program checks the flags, determines which flag is set, and then transfers the required input or output information. Once this is done, the instruction ION is executed to set IEN to 1 (to enable further interrupts), and the program returns to the location where it was interrupted. This is shown in **Figure 5-14(b).**

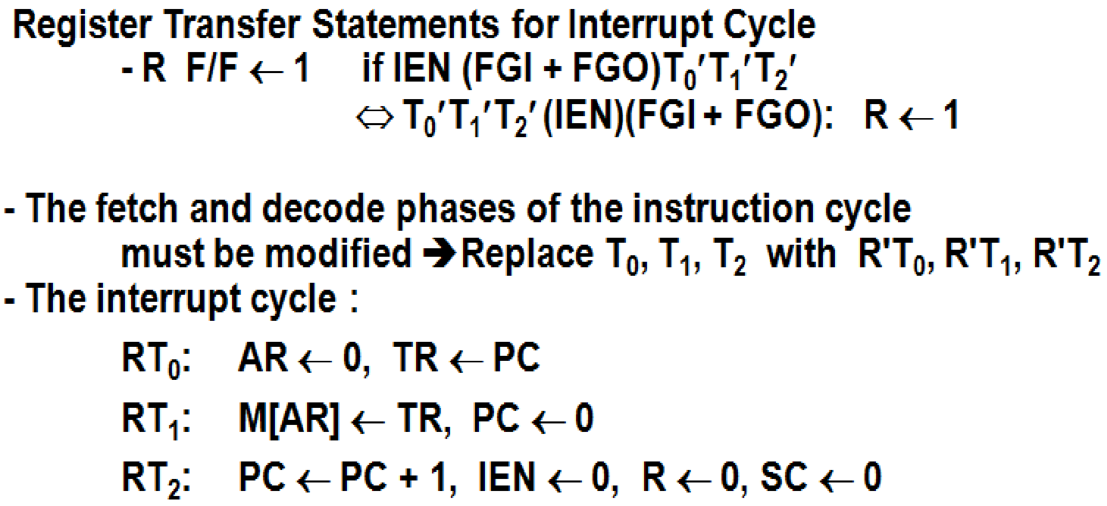
Figure 5-14 Demonstration of the interrupt cycle.



(a) Before interrupt

(b) After interrupt cycle

Resister transfer operations in interrupt cycle:



Register Transfer Statements for Interrupt Cycle

- R F/F $\leftarrow$ 1 if IEN (FGI + FGO)$T_0'T_1'T_2'$

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

- The fetch and decode phases of the instruction cycle must be modified ➔Replace T_0, T_1, T_2 with $R'T_0$, $R'T_1$, $R'T_2$

- The interrupt cycle :

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

14. Complete Computer Description:

→The final flow chart of instruction cycle including interrupt cycle for the basic computer is shown in **Figure5-15.**

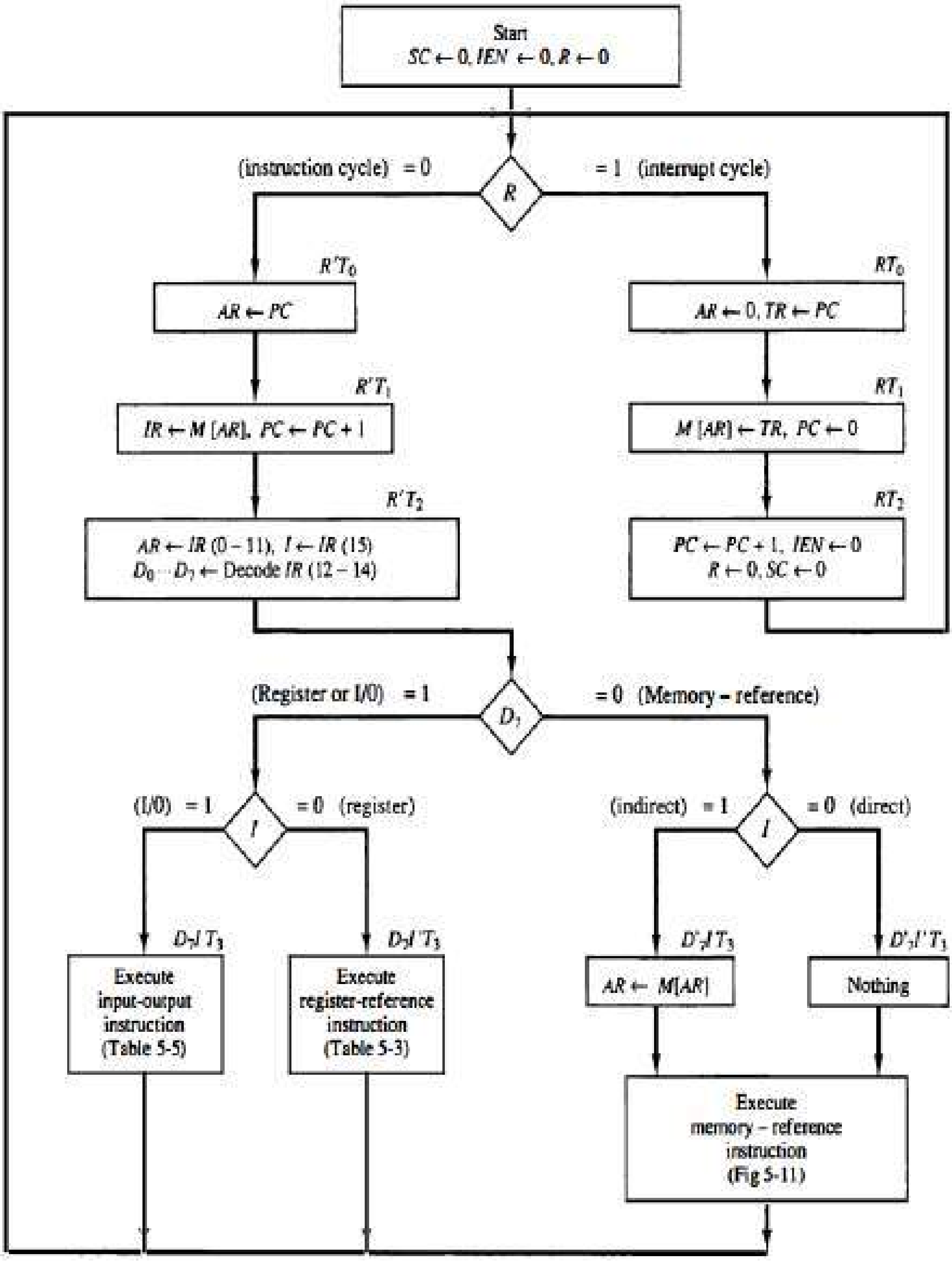
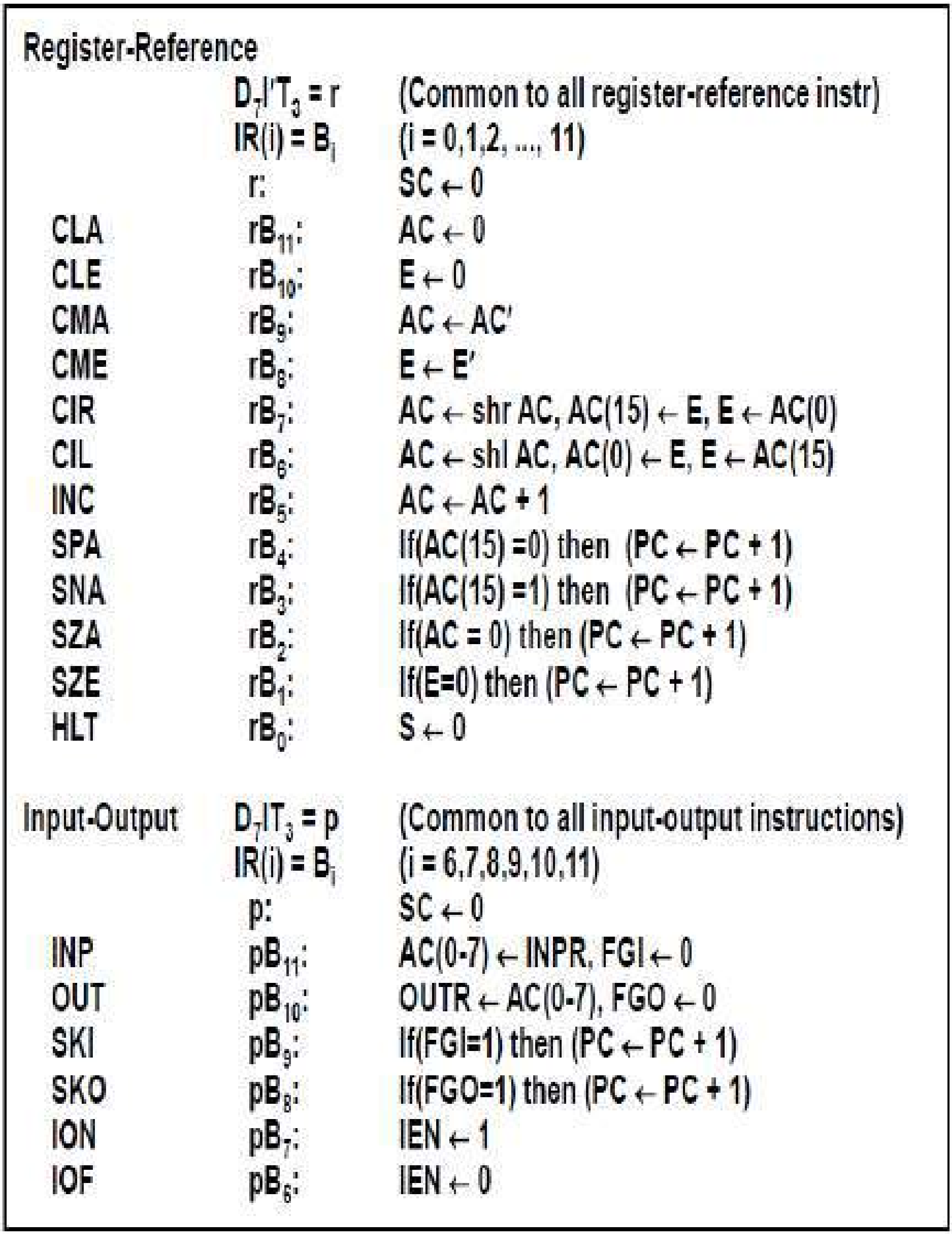
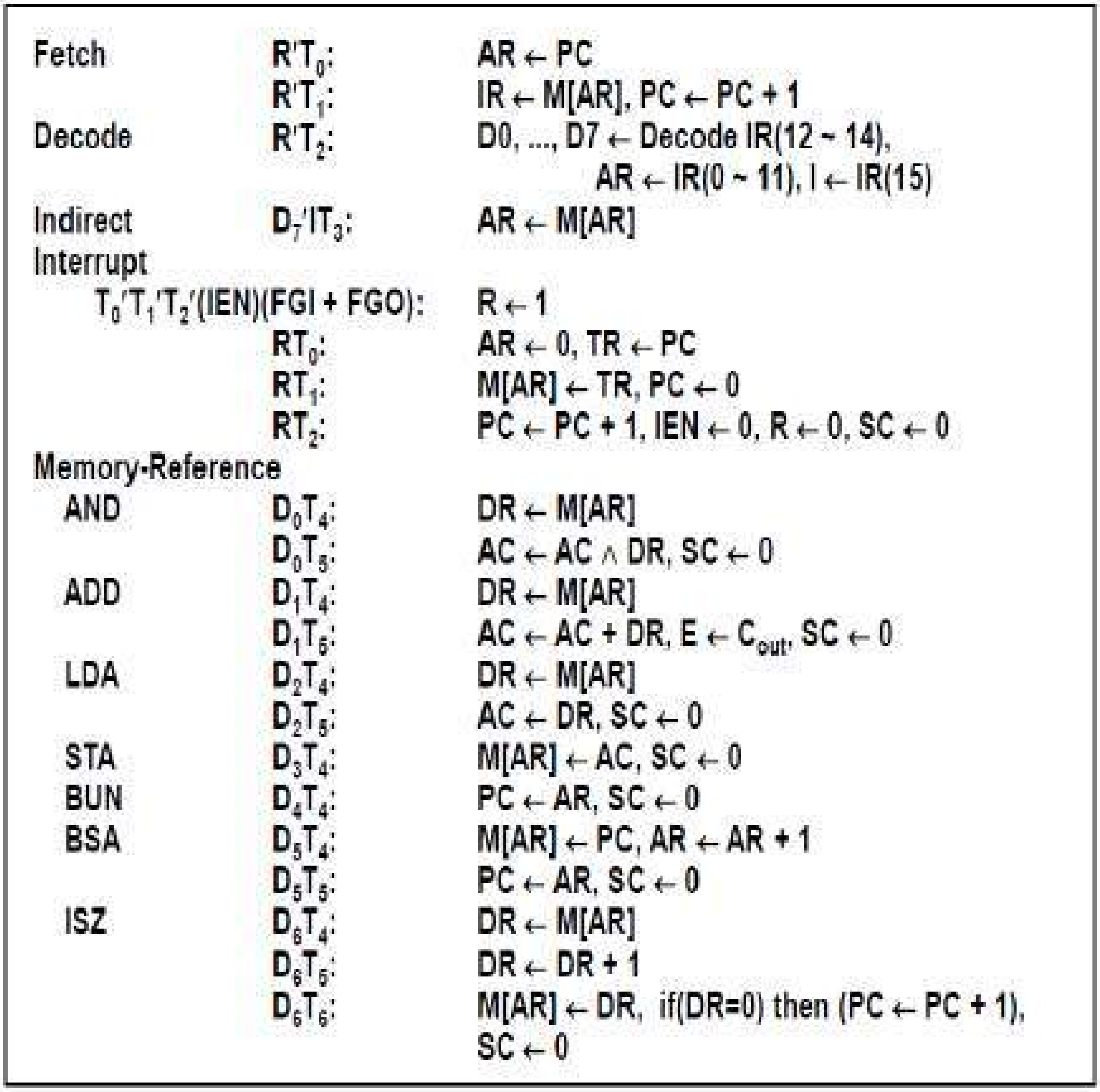

Start
$SC \leftarrow 0, IEN \leftarrow 0, R \leftarrow 0$
(instruction cycle) = 0
R
= 1 (interrupt cycle)
$R'T_0$
$AR \leftarrow PC$
$R'T_1$
$IR \leftarrow M[AR], PC \leftarrow PC + 1$
$R'T_2$
$AR \leftarrow IR(0-11), I \leftarrow IR(15)$
$D_0 \cdots D_7 \leftarrow$ Decode $IR(12-14)$
RT_0
$AR \leftarrow 0, TR \leftarrow PC$
RT_1
$M[AR] \leftarrow TR, PC \leftarrow 0$
RT_2
$PC \leftarrow PC + 1, IEN \leftarrow 0$
$R \leftarrow 0, SC \leftarrow 0$
(Register or I/0) = 1
D_7
= 0 (Memory – reference)
(I/0) = 1
I
= 0 (register)
D_7IT_3
$D_7I'T_3$
Execute input-output instruction (Table 5-5)
Execute register-reference instruction (Table 5-3)
(indirect) = 1
I
= 0 (direct)
D'_7IT_3
$D'_7I'T_3$
$AR \leftarrow M[AR]$
Nothing
Execute memory – reference instruction (Fig 5-11)

Figure 5-15 Flowchart for computer operation.

→The control functions and micro-operations for the entire computer are summarized in **Table 5-6**



Register-Reference		
	$D_7I'T_3 = r$	(Common to all register-reference instr)
	$IR(i) = B_i$	$(i = 0,1,2, ..., 11)$
	r:	$SC \leftarrow 0$
CLA	rB_{11}:	$AC \leftarrow 0$
CLE	rB_{10}:	$E \leftarrow 0$
CMA	rB_9:	$AC \leftarrow AC'$
CME	rB_8:	$E \leftarrow E'$
CIR	rB_7:	$AC \leftarrow \text{shr } AC, AC(15) \leftarrow E, E \leftarrow AC(0)$
CIL	rB_6:	$AC \leftarrow \text{shl } AC, AC(0) \leftarrow E, E \leftarrow AC(15)$
INC	rB_5:	$AC \leftarrow AC + 1$
SPA	rB_4:	If(AC(15) =0) then $(PC \leftarrow PC + 1)$
SNA	rB_3:	If(AC(15) =1) then $(PC \leftarrow PC + 1)$
SZA	rB_2:	If(AC = 0) then $(PC \leftarrow PC + 1)$
SZE	rB_1:	If(E=0) then $(PC \leftarrow PC + 1)$
HLT	rB_0:	$S \leftarrow 0$
Input-Output	$D_7IT_3 = p$	(Common to all input-output instructions)
	$IR(i) = B_i$	$(i = 6,7,8,9,10,11)$
	p:	$SC \leftarrow 0$
INP	pB_{11}:	$AC(0\text{-}7) \leftarrow INPR, FGI \leftarrow 0$
OUT	pB_{10}:	$OUTR \leftarrow AC(0\text{-}7), FGO \leftarrow 0$
SKI	pB_9:	If(FGI=1) then $(PC \leftarrow PC + 1)$
SKO	pB_8:	If(FGO=1) then $(PC \leftarrow PC + 1)$
ION	pB_7:	$IEN \leftarrow 1$
IOF	pB_6:	$IEN \leftarrow 0$



Fetch	$R'T_0$:	AR ← PC
	$R'T_1$:	IR ← M[AR], PC ← PC + 1
Decode	$R'T_2$:	D0, ..., D7 ← Decode IR(12 ~ 14), AR ← IR(0 ~ 11), I ← IR(15)
Indirect	$D_7'IT_3$:	AR ← M[AR]
Interrupt		
$T_0'T_1'T_2'(IEN)(FGI + FGO)$:		R ← 1
	RT_0:	AR ← 0, TR ← PC
	RT_1:	M[AR] ← TR, PC ← 0
	RT_2:	PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0
Memory-Reference		
AND	D_0T_4:	DR ← M[AR]
	D_0T_5:	AC ← AC $\wedge$ DR, SC ← 0
ADD	D_1T_4:	DR ← M[AR]
	D_1T_5:	AC ← AC + DR, E ← C_{out}, SC ← 0
LDA	D_2T_4:	DR ← M[AR]
	D_2T_5:	AC ← DR, SC ← 0
STA	D_3T_4:	M[AR] ← AC, SC ← 0
BUN	D_4T_4:	PC ← AR, SC ← 0
BSA	D_5T_4:	M[AR] ← PC, AR ← AR + 1
	D_5T_5:	PC ← AR, SC ← 0
ISZ	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

Design of Basic Computer:

The basic computer consists of the following hardware components:

1. A memory unit with 4096 words of 16 bits each.
2. Nine registers: AR, PC, DR, AC, IR, TR, OUTR, INPR, and SC.
3. Seven flip-flops: I, S, E, R, IEN, FGI, and FGO.
4. Two decoders: a 3 x 8 operation decoder and a 4 x 16 timing decoder.
5. A 16-bit common bus.
6. Control logic gates.
7. Adder and logic circuit connected to the input of AC.

COMPUTER ORGANIZATION – UNIT-2

IMPORTANT QUESTIONS

1. Explain the operation of 4-bit adder-subtractor with a neat diagram?
2. Discuss about various shift micro-operations?
3. Describe the working of arithmetic logic shift unit?
4. Describe a common bus system for four registers of 4 bit s each, using three state buffers and explain its functionality?
5. Discuss about Logic micro operations?
6. What are the register transfer logic languages? Explain few RTL statements for branching with their actual functioning?
7. Register A holds the 8-bit binary 11011001.Determine the B operand the logic micro operation to be performed in order to change the value in A to:
    (i) 001101101
    (ii) 11111101.
8. The following transfer statements specify a memory. Explain the memory operation in each case. (i) R2←M[AR] (ii)M[AR] ← R3 (iii) R5←M[R5]
9. Explain Register Transfer?
10. Write about stored program organization?
11. Explain in detail about control unit of basic computer with a neat diagram?
12. Discuss the phases of an instruction cycle?
13. What are the basic computer registers and explain with block diagram how information is transferred among the registers?
14. Draw the flowchart for Interrupt cycle and explain how input-output operations are performed.
15. Explain memory reference instructions?
16. Explain instruction cycle? Explain computer registers?
17. Write the sequence of micro-operations and draw the flow chart for interrupt cycle?
18. Show the hardware implementation for the following statements. The registers are 4-bit in length. T0: A←R0, T1:A←R1, T2: A←R2, T3←R3.
19. Write about the different Logic micro operations?
20. Describe Three State Buffers?
21. Micro-operations and Register Transfer Language.
22. If A=1101 and B=1001, perform the mask operation.
23. Explain Binary adder with a neat diagram?
24. Distinguish between Micro and Macro operations?

25. Explain the categories of micro operations?
26. What is the use of Register Transfer Language (RTL)?
27. Draw the circuit diagram of 4-bit binary adder?
28. Draw the diagram for register transfer for fetch phase?
29. What is instruction code?
30. What is direct addressing?

COMPUTER ORGANIZATION CHAPTER-3

Central Processing Unit: General Register Organization, STACK Organization. Instruction Formats, Addressing Modes, Data Transfer and Manipulation, Program Control, Reduced Instruction Set Computer.

Micro programmed Control: Control Memory, Address Sequencing, Micro Program example, Design of Control Unit.

1. General Register Organization:

Introduction:

→The part of computer that performs the bulk of data processing operations is called the central processing unit and is referred to as ***CPU***.

→The CPU is made of 3 parts, as shown in **figure**

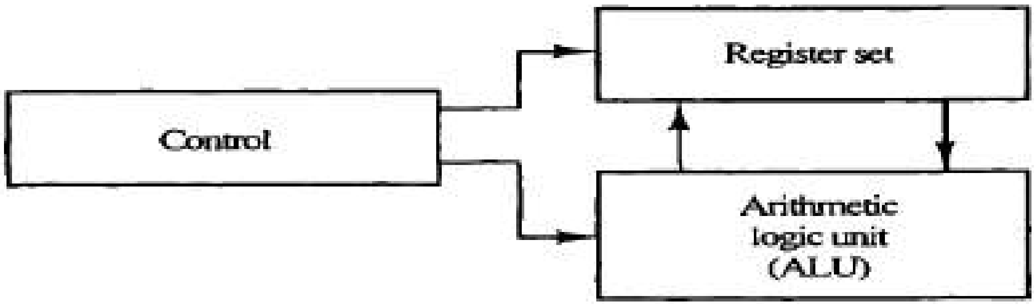


Figure 8-1 Major components of CPU.

1. **Registers:** The register set stores intermediate data used during the execution of the instructions.

2. **ALU:** The arithmetic logic unit (ALU) performs the required micro operations for executing the instructions.

3. **Control Unit**: The control unit supervises the transfer of data among the registers and instructs the ALU as to which operation to perform.

→Design Examples of simple CPU

l. Hardwired Control

2. Micro-programmed Control

→User who programs the computer in machine/assembly language must be aware of

1) Instruction Formats

2) Addressing Modes

3) Register Sets

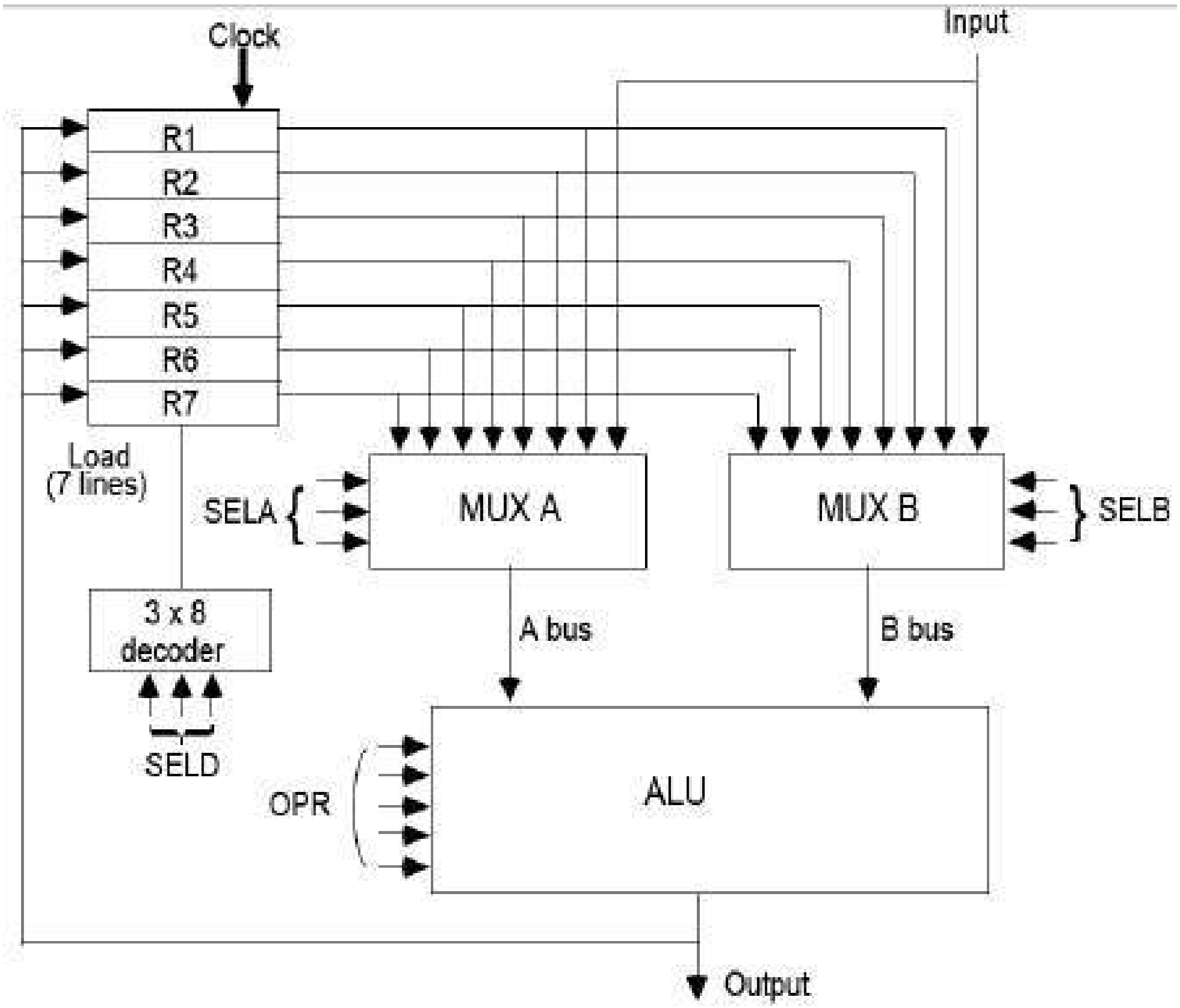
General Register Organization:

Register:

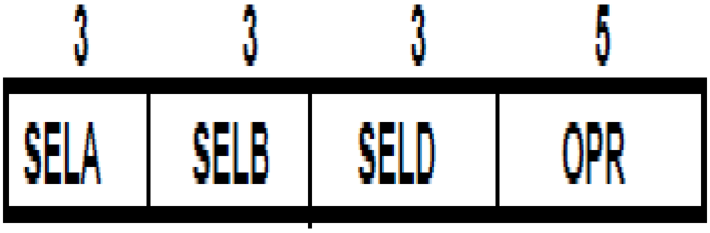
→Memory locations are needed for storing pointers, counters, return address, temporary results, and partial products during multiplication. Memory access is the most time-consuming operation in a computer. More convenient and efficient way is to store intermediate values in processor registers.

→A bus organization for seven CPU registers is shown in **Figure.**

→The output of each register is connected to two multiplexers (MUX) to form two buses A and B. The selection lines in each multiplexer select one register or the input data for the particular bus. The A and B buses form the inputs to a **common arithmetic logic circuit (ALU).**



(a). Block diagram



3	3	3	5
SELA	SELB	SELD	OPR

(b). Control word.

Figure 8-2 Register set with common ALU.

→**ALU:** The operation (OPR) selected in the ALU determines the arithmetic or logic micro-operation. The result of the micro-operation is available for data output and also goes into the inputs of all the registers.

→**3 X 8 Decoder**: select the register (by SELD) that receives the information from ALU.

→The control unit that operates the CPU bus system directs the information flow through the registers and ALU by selecting the various components in the system. For example, to perform the operation:

R1 ← R2 + R3

→**The control must provide binary selection variables to the following selector inputs:**

1) **MUX A selector (SELA):** to place the content of R2 into BUS A.

BUS A← R2

2) **MUX B selector (SELB):** to place the content of R3 into BUS B.

BUS B← R3

3) **ALU operation selector (OPR):** to provide the arithmetic addition A + B.

ALU to ADD

4) **Decoder destination selector (SELD):** to transfer the content of the output bus into R1.

R1← Out Bus

→These four control signals are generated in control unit in start of each clock cycle ensuring operands are selected beside correct ALU operation and result is chosen in one clock cycle only.

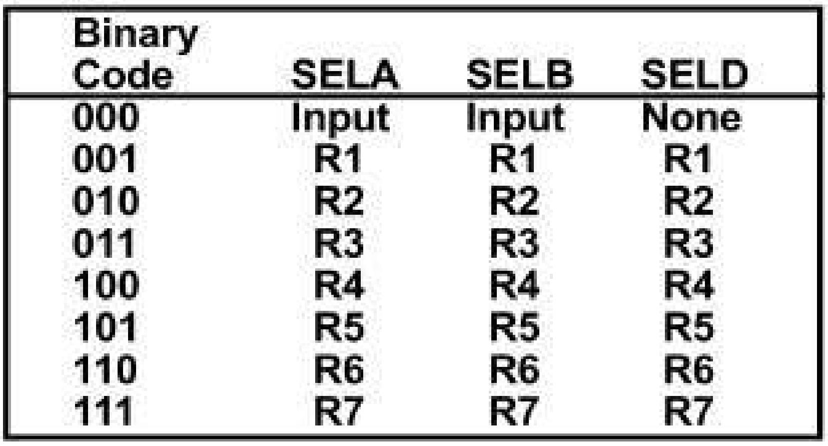
Control Word:

→There are 14 binary selection inputs in the unit, and their combined value specifies a **control word.** The 14-bit control word is defined in **Fig. 8-2(b).**

→**14 bit control word (4 fields):**

- **SELA (3 bits):** select a source register for the A input of the ALU
- **SELB (3 bits):** select a source register for the B input of the ALU
- **SELD (3 bits):** select a destination register using the 3 X 8 decoder
- **OPR (5 bits):** select one of the operations in the ALU

→The encoding of the register selections is specified in **Table 8-1**.



Binary Code	SELA	SELB	SELD
000	Input	Input	None
001	R1	R1	R1
010	R2	R2	R2
011	R3	R3	R3
100	R4	R4	R4
101	R5	R5	R5
110	R6	R6	R6
111	R7	R7	R7

TABLE: 8-1 Encoding of Register Selection Fields.

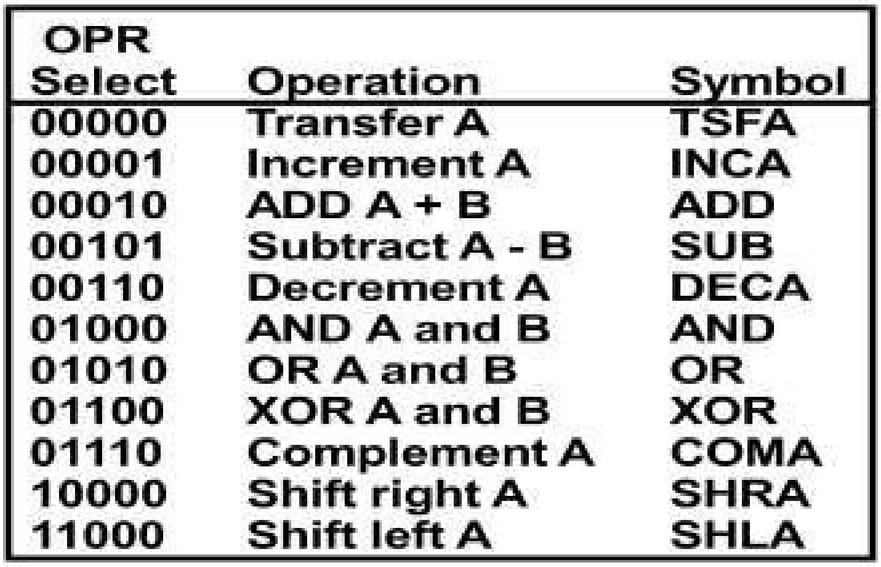
→The register selected by fields SELA, SELB and SELD is the one whose decimal number is equivalent to the binary number in the code.

→When SELA or SELB is 000 (Input) the corresponding MUX selects the external input data.

→When SELD = 000 (None), no destination register is selected but the contents of the output bus are available in the external output.

→**Encoding of ALU Operation (OPR):**

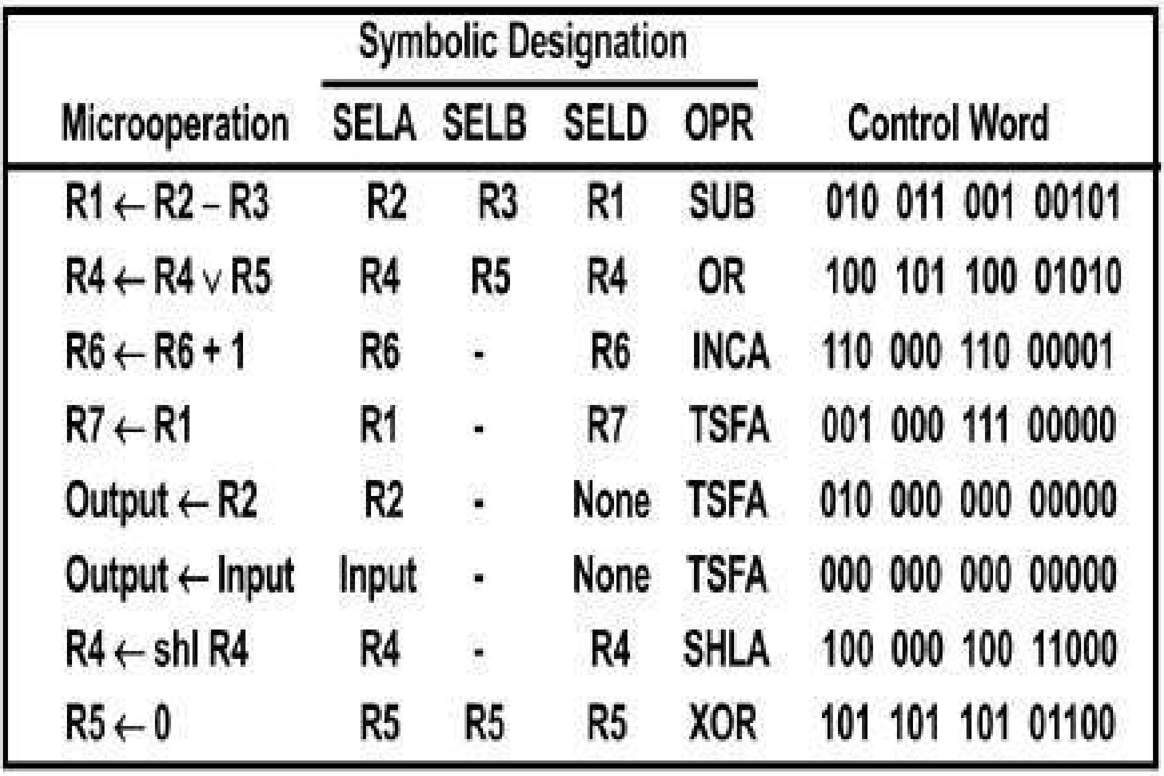
→The OPR field has five bits and each operation is designed with a symbolic name.



OPR Select	Operation	Symbol
00000	Transfer A	TSFA
00001	Increment A	INCA
00010	ADD A + B	ADD
00101	Subtract A - B	SUB
00110	Decrement A	DECA
01000	AND A and B	AND
01010	OR A and B	OR
01100	XOR A and B	XOR
01110	Complement A	COMA
10000	Shift right A	SHRA
11000	Shift left A	SHLA

TABLE 8-2 Encoding of ALU Operations

→**Examples of Micro-operations:**



	Symbolic Designation				
Microoperation	SELA	SELB	SELD	OPR	Control Word
R1 ← R2 – R3	R2	R3	R1	SUB	010 011 001 00101
R4 ← R4 ∨ R5	R4	R5	R4	OR	100 101 100 01010
R6 ← R6 + 1	R6	-	R6	INCA	110 000 110 00001
R7 ← R1	R1	-	R7	TSFA	001 000 111 00000
Output ← R2	R2	-	None	TSFA	010 000 000 00000
Output ← Input	Input	-	None	TSFA	000 000 000 00000
R4 ← shl R4	R4	-	R4	SHLA	100 000 100 11000
R5 ← 0	R5	R5	R5	XOR	101 101 101 01100

TABLE 8-3 Examples of Micro-operations for the CPU.

2. Stack Organization:

→A **Stack (LIFO: Last In Fist Out)**is a storage device that stores information in such a manner that the item stored last is the first item retrieved.

→Stack Pointer (SP): The register that holds the address for the stack. SP always points at the top item in the stack.

→Two Operations of a stack are Insertion and Deletion of Items.

- **PUSH (Push Down),** operation of insertion of items into stack.
- **POP (Pop Up),** operation of deletion item from stack.

→These operations are simulated by incrementing or decrementing the **stack pointer register (SP).**

1. Register Stack:

→ A stack can be placed in a portion of a large memory or it can be organized as a collection of a finite number of memory words or registers.

→**Stack pointer (SP)** points to the register that is currently at the top of stack.

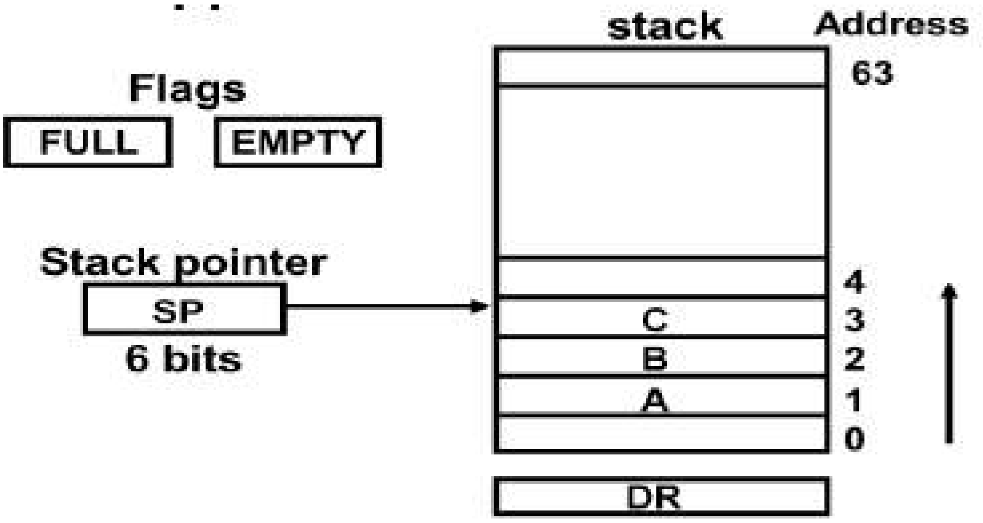


Figure 8-3 Block diagram of a 64-word stack.

→**Figure 8-3** shows the organization of a 64-word register stack. The stack pointer register SP contains a binary number whose value is equal to the address of the word that is currently on top of the stack. Three items are placed in the stack: A, B, and C, in that order. Item C is on top of the stack so that the content of SP is now 3. To remove the top item, the stack is popped by reading the memory word at address 3 and decrementing the content of SP. Item B is now on top of the stack since SP holds address 2. To insert a new item, the stack is pushed by incrementing SP and writing a word in the next-higher location in the stack. Note that item C has been read out but not physically removed.

→In a 64-word stack, the stack pointer contains 6 bits because 2 6 = 64. Since SP has only six bits, it cannot exceed a number greater than 63 (111111 in binary). When63 is incremented by 1, the result is 0 since 111111 + 1 = 1000000 in binary, but SP can accommodate only the six least significant bits.

PUSH:

→Initially SP is cleared to 0, EMPTY is set to 1(true), and FULL is cleared to 0(false), so that SP points to the word at address 0 and the stack is marked empty and not full.

→If the stack is not full (if FULL=0), a new item is inserted with a push operation.

→The push operation is implemented with the following sequence of micro-operations:

/* Initially, SP = 0, EMPTY = 1(true), FULL = 0(false) */

```
SP ←SP + 1                        Increment stack pointer.
M [SP] ←DR                        Write item on top of the stack
If (SP = 0) THEN (FULL ← 1)       Check if stack is full
EMTY ←0                           Mark the stack not empty
```

→The first item is stored at address 1, and the last item is stored at address 0.

Note that:

- Always we use DR to pass word into stack.
- M [SP] memory word specified by address currently in SP.
- First item stored in stack is at address 1.
- Last item stored in stack is at address 0. That is FULL = 1.
- Any push to stack means EMTY = 0.

POP:

→A new item is deleted from the stack if the stack is not empty (I f EMPTY=0).

→The pop operation consists of the following sequence of micro-operations:

/* Initially, SP = 0, EMPTY = 1(true), FULL = 0(false) */

DR←M [SP]Read item from the top of stack

SP ←SP – 1Decrement stack pointer

If (SP = 0) THEN (EMTY ←1) Check if stack is empty

FULL ←0 Mark the stack not full

Note That:

- Top of stack is read into DR.
- If SP reached 0 then stack is EMTY = 1. That when SP was 1 then pop occurred. No more pops can happen from here.
- Any pop from stacks means FULL = 0.

2. Memory Stack:

→A stack can exist as a stand-alone unit as in **Figure.8-3** (or) stack can be implemented in a Random Access Memory attached to a CPU.

→**Figure 8-4** shows a portion of computer memory partitioned into three segments: program, data and stack.

→The **Program Counter PC** points at the address of the next instruction in the program. The **Address Register AR** points at an array of data. The **Stack pointer SP** points at the top of the stack. Three registers are connected a common address bus, and either one can provide an address for memory. PC is used during the fetch phase to read an instruction. AR is used during the execute phase to read an operand. SP is used to push or pop items into or from the stack.

→As shown in **Figure. 8-4**, the initial value of SP is 4001 and the stack grows (pushed) with decreasing addresses. Thus the first item in the stack is at address 4000, the second item is stored at address 3999, and the last address that can be used for the stack is 3000.

→We assume that the items in the stack communicate with a data register DR. A New item is inserted with **push operation** as follows:

SP ←SP -1

M [SP] ←DR

→The stack pointer is decremented so that it points at the address of the next word. A memory write operation inserts the word from the DR into the top of the stack. A new item is deleted with a **pop operation** as follows:

DR ←M [SP]
SP ←SP + 1

→The top item is read from the stack into DR. The stack pointer is then incremented to point at the next item in the stack.

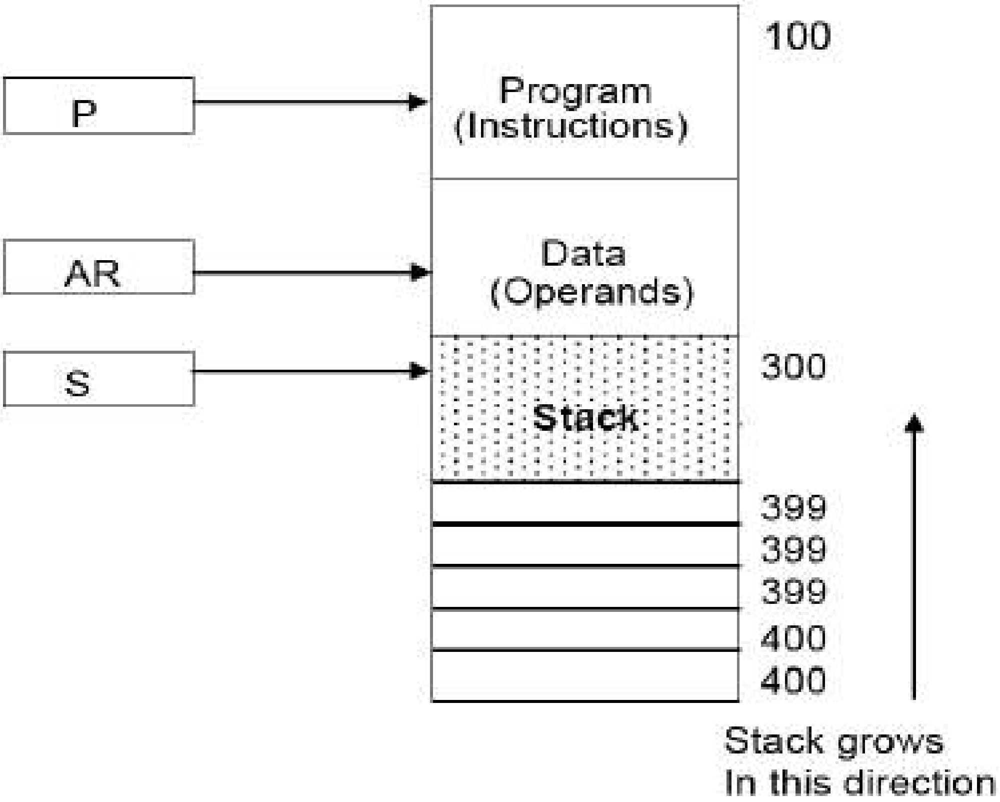


Figure 8-4 Computer memory with program, data, and stack segments.

Stack limits:

→Check for stack overflow (*full*) / underflow (*empty*)

→Checked by using two register: Upper Limit and Lower Limit Register

→After PUSH Operation: SP compared with the upper limit register.

→After POP Operation: SP compared with the lower limit register.

3. Reverse Polish Notation (RPN):

→The common mathematical method of writing arithmetic expressions imposes difficulties when evaluated by a computer.

→A stack organization is very effective for evaluating arithmetic expressions.

→ We write in infix notation such as:

A*B + C*D

→The 3 notations to evaluate expressions:

1. A + B Infix notation

2. +AB Prefix notation (Polish notation)

3. AB+ Postfix notation (reverse Polish notation)

→**Reverse Polish Notation** is in a form suitable for stack manipulation. The expression:

A * B + C * D

Is written in Reverse Polish notation as AB * CD *+

A * B + C * D → AB * CD *+:

→**Evolution of Arithmetic Expressions:**

Example, (3 * 4) + (5 * 6) →34 * 56 * +

Figure 8-5 Stack operations to evaluate 3 • 4 + 5 • 6.

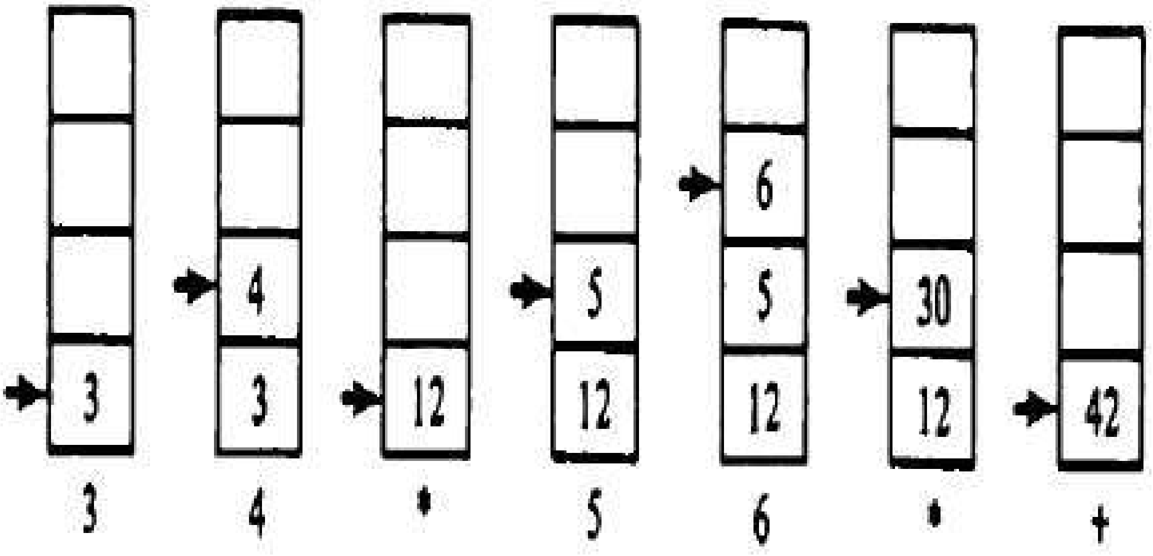


Figure 8-5 Stack operations to evaluate, (3 * 4) + (5 * 6)

3. Instruction Formats:

→It is the function of the control unit within the CPU to interpret each instruction code. The bits of the instructions are divided into groups called fields.

Fields in Instruction Formats:

1) **Operation Code Field**: An operation code field that specifies the operation to be performed

2) **Address Field**: An address field that designates a memory address or a processor registers.

3) **Mode Field**: A mode field that specifies the way the operand or the effective address is determined (*Addressing Mode*)

→A **register address** is a binary number of *k* bits that defines one of 2^k registers in the CPU. Thus a CPU with 16 processor registers R0 through R15 will have a register address fields of four bits. The binary number 0101, for example, will designate registerR5.

→Computers may have instructions of several different lengths containing varying number of addresses. The number of address fields in the instruction format of a computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organizations:

- **Single accumulator organization**
- **General register organization**
- **Stack organization**

→**Single register (Accumulator) organization:**

» Basic Computer is a good example.

» Accumulator is the only general purpose register.

» Uses implied accumulator register for all operations.

E.g**. ADD X // AC ← AC + M[X]**

LDA Y // AC ← M[Y]

→**General register organization:**

» Used by most modern computer processors

» Any of the registers can be used as the source or destination for computer operations.

E.g**. ADD R1, R2, R3 // R1← R2 + R3**

ADD R1, R2 // R1 ← R1 + R2

MOV R1, R2 // R1 ← R2

ADD R1, X // R1 ← R1 + M[X]

→**Stack organization:**

» All operations are done with the stack

» For example, an OR instruction will pop the two top elements from the stack, do a Logical OR on them, and push the result on the stack

E.g**. PUSH X // TOS ← M[X]**

ADD // TOS=TOP(S) + TOP(S)

Types of instructions:

→To illustrate the influence of the number of addresses on computer programs, we will evaluate the arithmetic statement **X = (A + B) • (C + D)** using **zero, one, two, or three address instructions**.

COMPUTER ORGANIZATION CHAPTER-3

→We will use the symbols ADD, SUH, MUL, and DIV for the four arithmetic operations; MOV for the transfer-type operation; and LOAD and STORE for transfers to and from memory and AC register. We will assume that the operands are in memory addresses A, B, C, and D, and the result must be stored in memory at address X.

Three-Address Instructions:

→Computer with three addresses instruction format can use each address field to specify either processor register or memory operand.

Program to evaluate X = (A + B) * (C + D):

ADD R1, A, B R1 ← M [A] + M [B]

ADD R2, C, D R2 ← M[C] + M [D]

MUL X, R1, R2 M[X] ← R1 * R2

- Results in short programs
- Instruction becomes long (many bits)

→The advantage of the three address formats is that it results in short program when evaluating arithmetic expression. The disadvantage is that the binary-coded instructions require too many bits to specify three addresses.

Two-address instructions:

→These are most common used commercial computers. Each address field can specify either a processor register or a memory word.

Program to evaluate X = (A + B) * (C + D):

MOV R1, A R1 ←M [A]

ADD R1, B R1 ← R1 + M [B]

MOV R2, C R2 ← M[C]

ADD R2, D R2 ← R2 + D

MUL R1, R2 R1 ← R1 * R2

MOV X, R1 M[X] ← R1

- Tries to minimize the size of instruction.
- Size of program is relative larger.

One-address instructions:

→It used an implied accumulator (AC) register for all data manipulation. For multiplication / division, there is a need for a second register.

Program to evaluate X = (A + B) * (C + D):

LOAD A AC ←M [A]

ADD B AC ← AC + M [B]

STORE TM [T] ← AC

LOAD C AC ← M[C]

ADD D AC ← AC + M [D]

MUL T AC ← AC * M [T]

STORE X M[X] ← AC

- Memory access is only limited to load and store.
- Large program size.

Zero-address instructions:

→Zero-address instructions can be found in a stack-organized computer.

→**Program to evaluate X = (A + B) * (C + D):**

PUSH A TOS ← A

PUSH B TOS ←B

ADDTOS ← (A +B)

PUSH C TOS ← C

PUSH D TOS ← D

ADD TOS ← (C + D)

MUL TOS ← (C + D) * (A + B)

POPX M[X] ← TOS

COMPUTER ORGANIZATION CHAPTER-3

→A stack organized computer does not use an address field for the instruction ADD and MUL. The PUSH & POP instruction, however, need an address field to specify the operand that communicates with the stack (TOS → Top of the stack).

RISC instructions:

→Program to evaluate X = (A + B) * (C + D):

LOAD R1, A R1 ←M [A]
LOAD R2, B R2 ←M [B]
LOAD R3, C R3 ← M[C]
LOAD R4, D R4 ←M [D]
ADD R1, R1, R2 R1 ← R1 + R2
ADD R3, R3, R4 R3 ← R3 + R4
MULR1, R1, R3 R1 ← R1 * R3
STOREX, R1 M[X] ← R1

4. ADDRESSING MODES:

→The **addressing mode** specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.

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

1. To give programming versatility 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.

→The control unit of a computer is designed to go through an instruction cycle that is divided into three major phases:

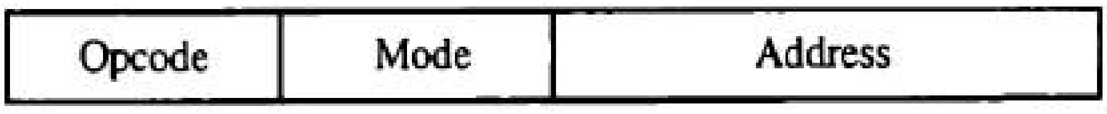
1. Fetch the instruction from memory.
2. Decode the instruction.
3. Execute the instruction.

→Program Counter (PC):

→There is one register in the computer called the **program counter or PC** that keeps track of the instructions in the program stored in memory. PC holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory. The decoding done in step 2 determines the operation to be performed, the addressing mode of the instruction, and the location of the operands. The computer then executes the instruction and returns to step 1 to fetch the next instruction in sequence.

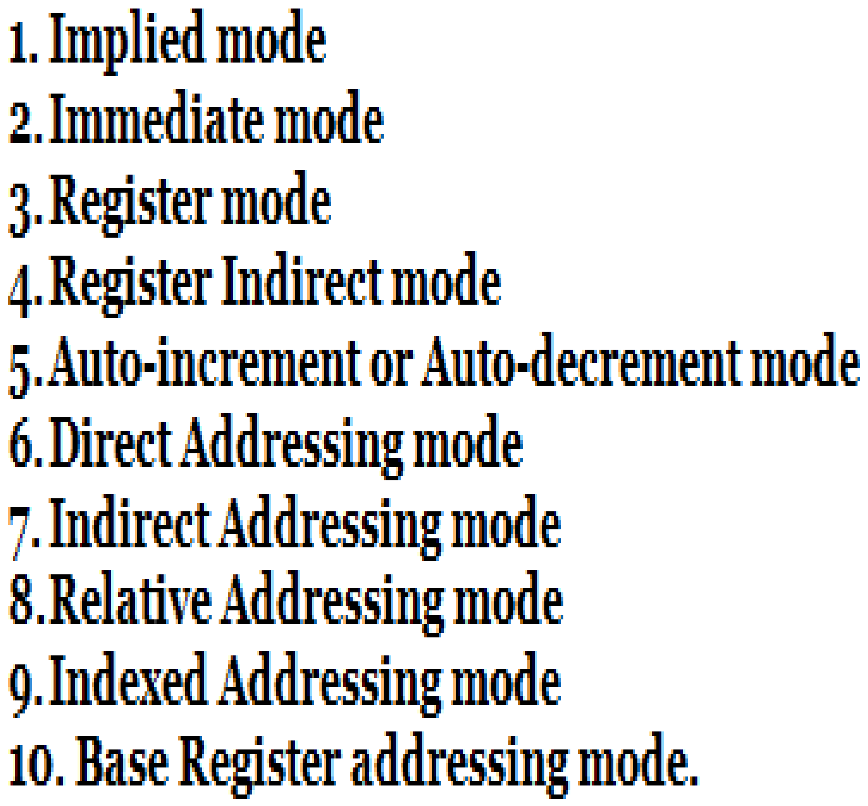
→An example of an **instruction format** with a distinct addressing mode field is shown in **Figure.**

Figure 8-6 Instruction format with mode field.



Opcode	Mode	Address

Different types of Addressing Modes:



1. Implied mode
2. Immediate mode
3. Register mode
4. Register Indirect mode
5. Auto-increment or Auto-decrement mode
6. Direct Addressing mode
7. Indirect Addressing mode
8. Relative Addressing mode
9. Indexed Addressing mode
10. Base Register addressing mode.

COMPUTER ORGANIZATION CHAPTER-3

→Two addressing modes require no address fields: the **implied mode** and **immediate mode.**

1. Implied mode:

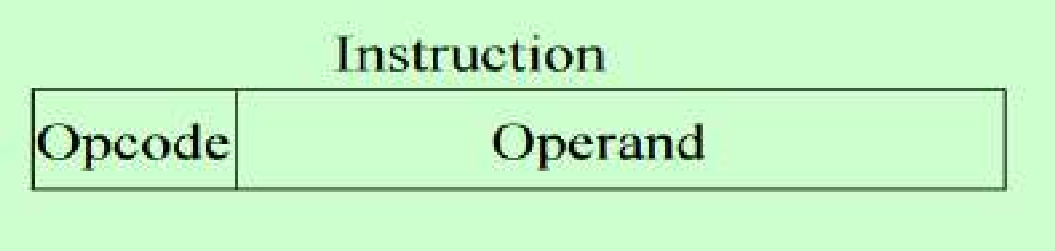
- The operands are specified implicitly in the definition of the instruction
- No need to specify address in the instruction
- All register reference instruction that uses an accumulator are implied mode instructions.
- Zero address instructions in stack-organized computers are implied mode instruction since operands are implied always at top of stack.
- Examples from Basic Computer: CLA, CME, INP

  ADD X;

  PUSH Y;

2. Immediate mode:

- Instead of specifying the address of the operand, operand itself is specified in the instruction.
- Operand is part of instruction.
- Operand = operand field
- E.g. ADD 5 —Add 5 to contents of accumulator —5 is the operand
- No memory reference to access data
- Fast to acquire an operand
- Range of operands limited to # of bits in operand field (< word size)

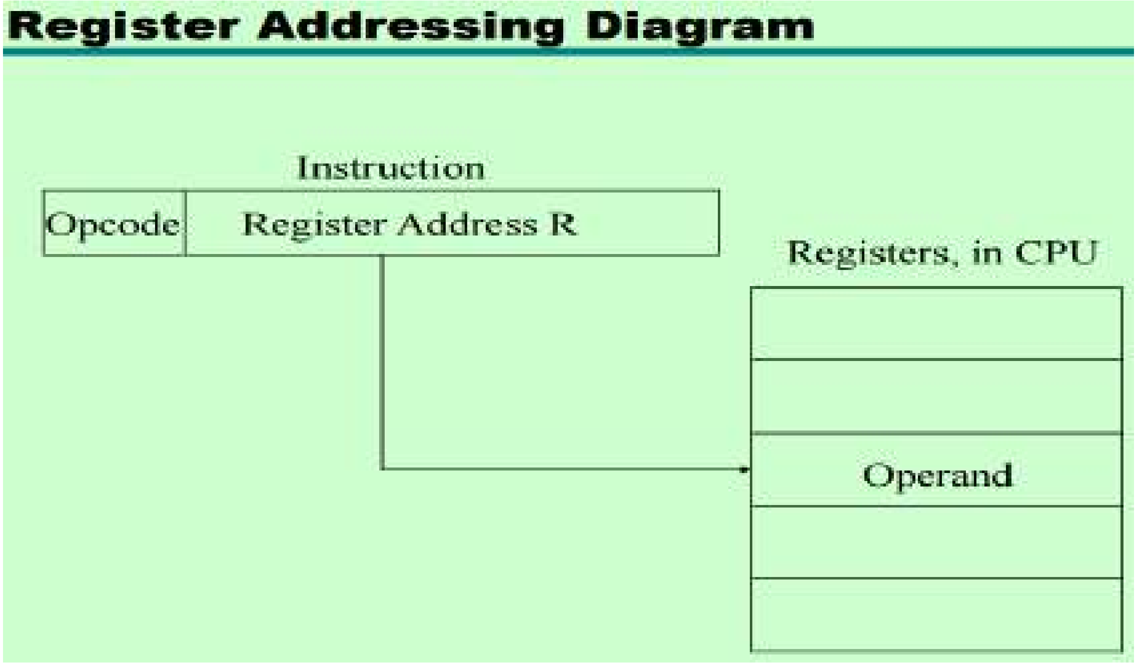


3. Register mode:

- Address specified in the instruction is the register address that resides within CPU.
  - Operand is held in register named in address filed
  - **Effective Address (EA) = Register address R**
  - **Advantages:** Very small address field —Shorter instructions —Faster instruction fetch

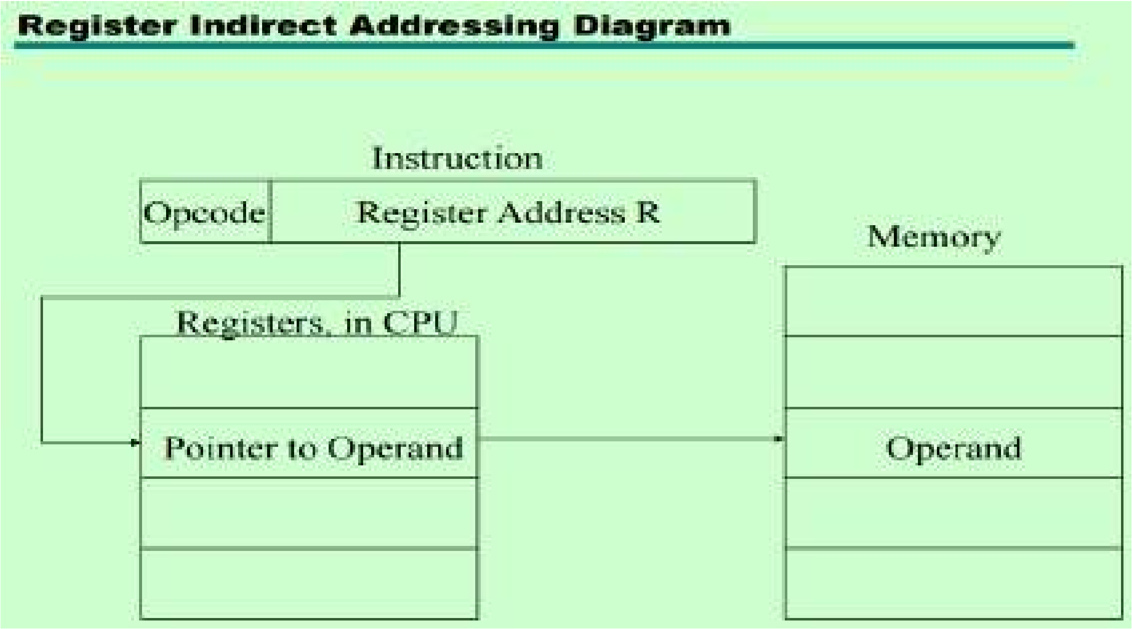
    • Faster memory access to operand(s)

- **Disadvantage:** Very limited address space



4. Register indirect mode:

- In this mode the instruction specifies a register which contains the memory address of the operand.
- **EA (effective address) = content of R – (R)**
- Operand is in memory cell pointed to by contents of register R Comparison with (memory) indirect: • Same large address space (2n) • One less memory access!



5. Auto-increment or Auto-decrement mode:

- It is similar to register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory.
- When 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.
- Automatically implement **Increment/Decrement** content of **specified register**.

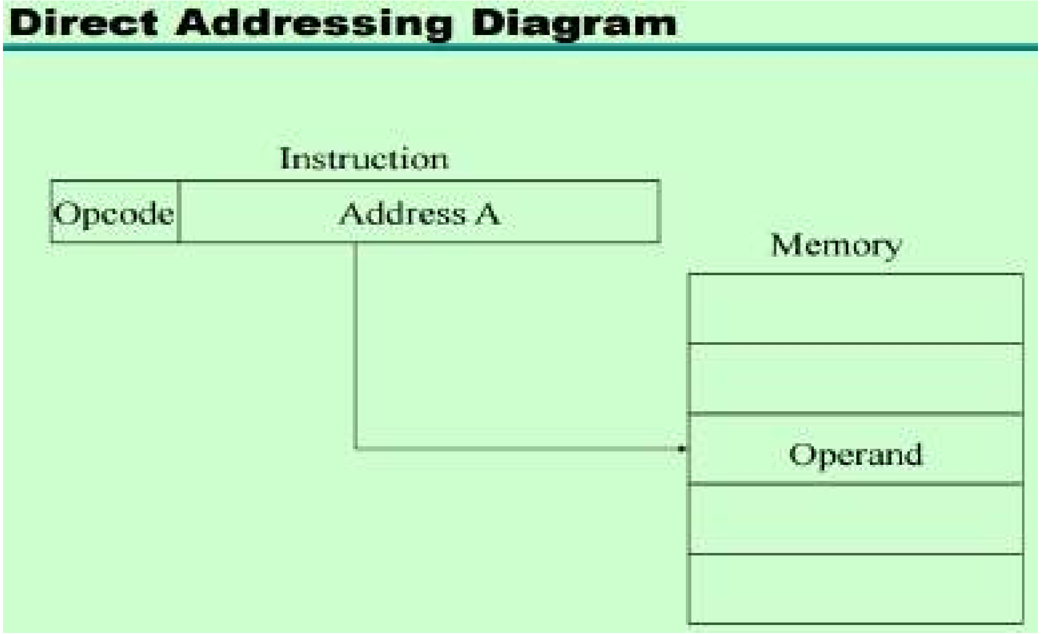
Effective address: The effective address is defined to be the memory address obtained from the computation dictated by the given addressing mode. The effective address is the address of the operand in a computational-type instruction.

6. Direct addressing mode:

- In this mode Address field contains address of operand.
- **Effective address (EA) = address field (A)**
- E.g. ADD A

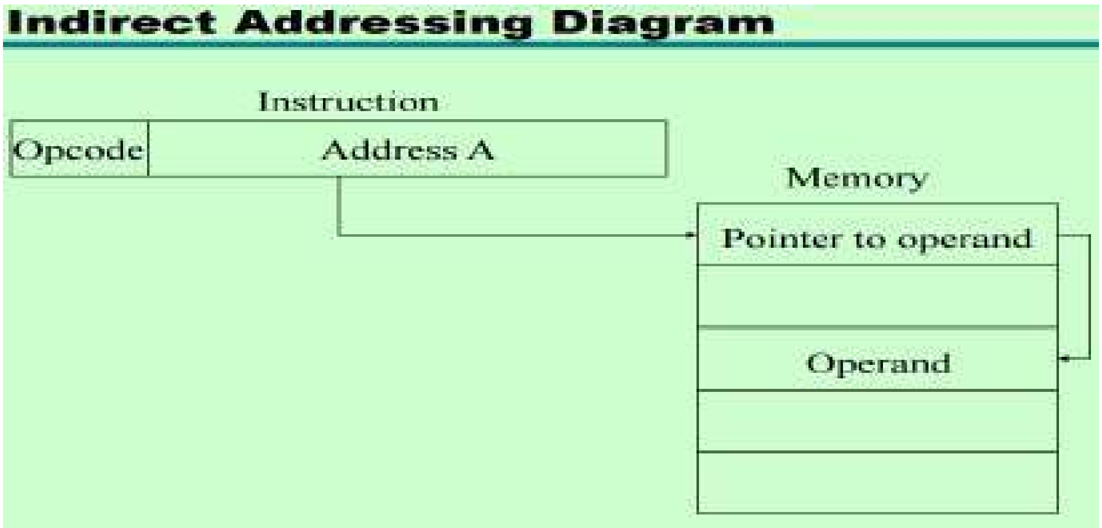
  —Add contents of cell A to accumulator

  —Look in memory at address A for operand
- Single memory reference to access data.
- No additional calculations to work out effective address.
- Range of addresses limited by # of bits in A (< word length).



7. Indirect addressing mode:

- Memory cell pointed to by address field contains the address of (pointer to) the operand
- **Effective Address (EA) = Contents of Address Register- (A)**
    - Look in A, find address (A) and look there for operand.
- E.g. ADD (A) —Add contents of cell pointed to by contents of A to accumulator.
- **Advantage**: Large address space —2^n where n = word length.
- **Disadvantage:** Multiple memory accesses to find operand slower.



→Displacement Addressing:

• EA = A + (R)

• Address field holds two values

—A = base value

—R = register that holds displacement

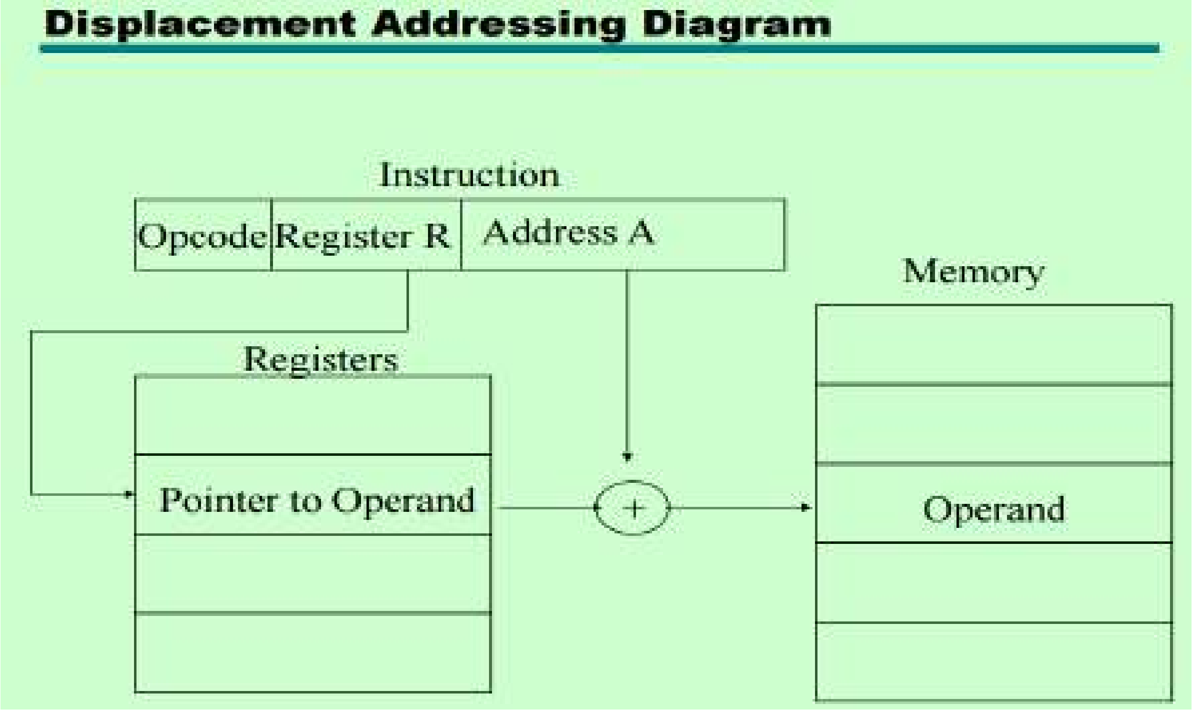
—or vice versa

- Has many versions, of which we mention these 3:

—Relative

—Base-register

—Indexing



8. Relative (to PC) Addressing mode:

- It's a version of displacement addressing.
- In this mode the content of the program counter PC added to the address part of the instruction in order to obtain the effective address.
- R = Program counter, PC

- **Effective Address (EA) = Address Register A + (PC)**
  —The operand is A cells away from the current cell (the one pointed to by PC)
- Example: if PC=825 and address part in instruction =24. Then address branched to =826 + 24 = 850

9. Base register Addressing mode:

- It's a version of displacement addressing
- In this mode the content of base register is added to the address part of the instruction to obtain EA.
- Similar to index addressing except register is called base register instead of index register
- It's a generalized relative addressing, where other registers can play the role of PC
- A holds displacement and R holds pointer to base address

- **Effective Address (EA) = A + (R)**
  ---R may be explicit or implicit
- E.g. . six segment registers in 80x86: CS, DS, ES, FS, GS, SS

10. Indexed Addressing mode:

- It's a version of displacement addressing
- In this mode the content of an index register is added to the address part of the instruction to obtain the effective address EA. The index register is special CPU register that contains an index value.

- **Effective Address (EA) = Base Address Register A + Contents of Address Register-(A)**
- Address field holds two values

—A = base value

—R = register that holds displacement

—or vice versa

Addressing modes (Example):

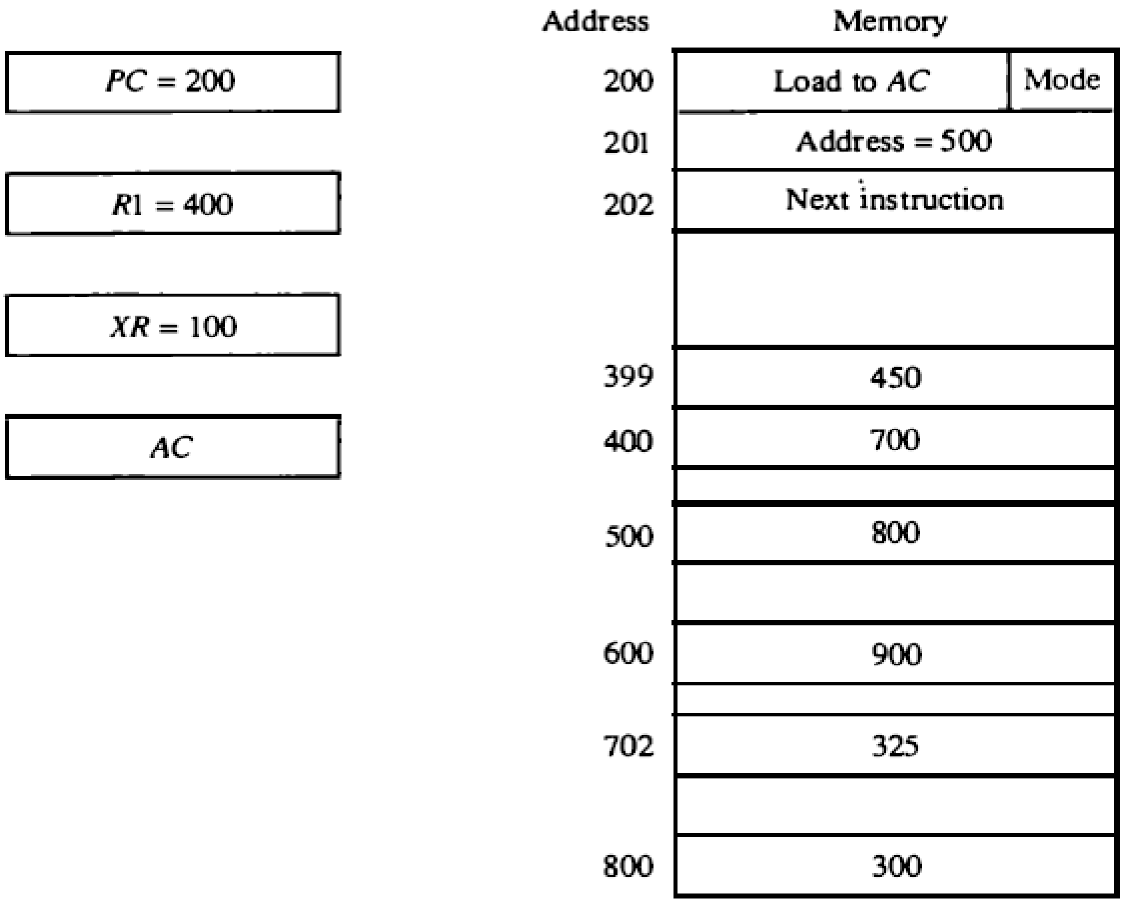
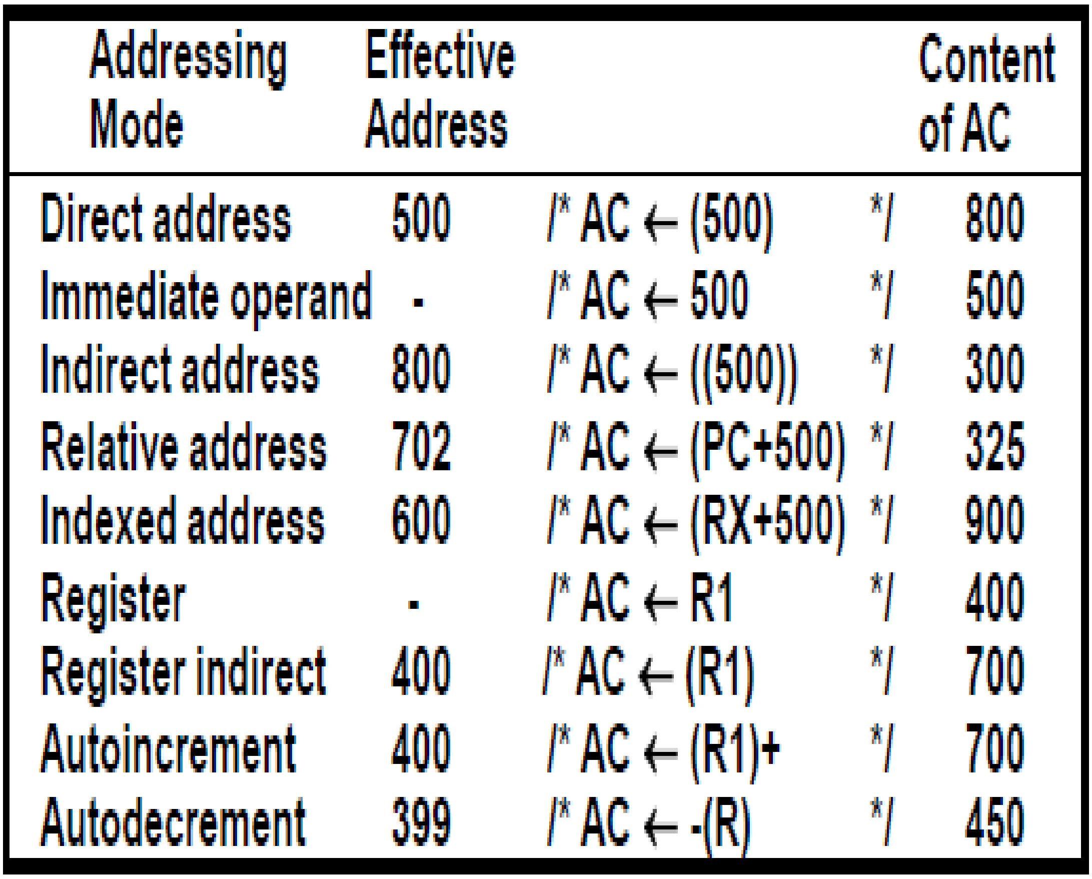


Figure 8-7 Numerical example for addressing modes.

Direct address	500	// AC ← M [500]
Value = 800		
Immediate operand		// AC ← 500
Value = 500		
Indirect address	500	// AC ← M [M [500]]
Value = 300		
Relative address	500	// AC ← M [PC+500]
Value = 325		
Indexed address	500	// AC ← (IX+500)
Value = 900		
Register	500	// AC ← R1
Value = 400		
Register indirect	500	// AC ← M [R1]
Value = 700		
Autoincrement	500	// AC ← (R1)
Value = 700		
Autodecrement	399	//* AC ← -(R) */
Value = 450		



Addressing Mode	Effective Address		Content of AC
Direct address	500	/* AC ← (500) */	800
Immediate operand	-	/* AC ← 500 */	500
Indirect address	800	/* AC ← ((500)) */	300
Relative address	702	/* AC ← (PC+500) */	325
Indexed address	600	/* AC ← (RX+500) */	900
Register	-	/* AC ← R1 */	400
Register indirect	400	/* AC ← (R1) */	700
Autoincrement	400	/* AC ← (R1)+ */	700
Autodecrement	399	/* AC ← -(R) */	450

5. Data Transfer and Manipulation:

→Most computer instructions can be classified into three categories:

- **Data transfer instructions**
- **Data manipulation instructions**
- **Program control instructions**

→**Data transfer instructions** cause transfer of data from one location to another without changing the binary information content.

→**Data manipulation instructions** are those that perform arithmetic, logic, and shift operations.

→**Program control instructions** provide decision-making capabilities and change the path taken by the program when executed in the computer.

Data transfer instructions:

→Data transfer instructions move data from one place in the computer to another without changing the data content. The most common transfers are between memory and processor registers, between processor registers and input or output, and between the processor registers them.

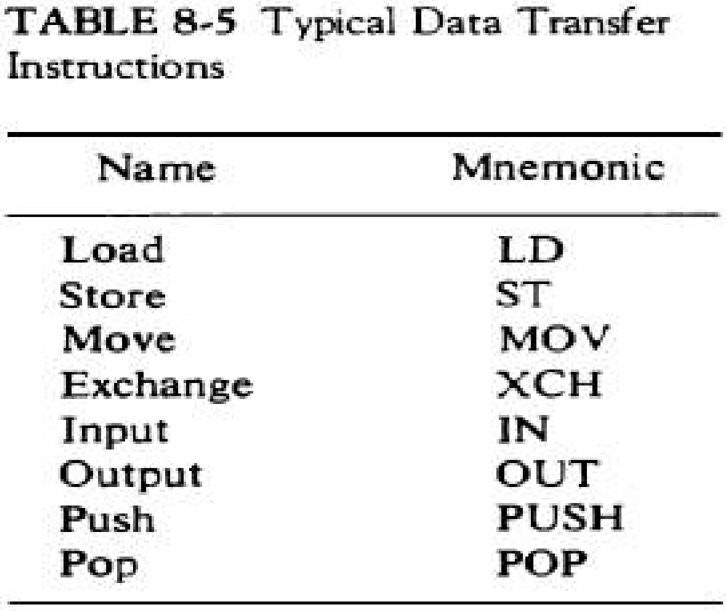


TABLE 8-5 Typical Data Transfer Instructions

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

- **Load** instruction is used to transfer data from memory to processor register(s) (Accumulator).
- **Store** instruction transfers data from register(s)(Accumulator) to memory.
- **Move** instruction is used to move data from registers and from register to memory and vice versa.
- **Exchang**e instruction swaps data between 2 registers or between 2 memory locations.
- **Input-Output** instructions transfer data between processors and IO device.
- **Push-Pop** instructions transfer data between stack and registers.

→Some assembly language conventions modify the mnemonic symbol to differentiate between addressing modes. For example, the mnemonic for load immediate becomes LDI.

→As an example, consider the load to accumulator instruction when used with eight different addressing modes.

→**Table 8-6** shows the recommended assembly language convention and the actual transfer accomplished in each case.

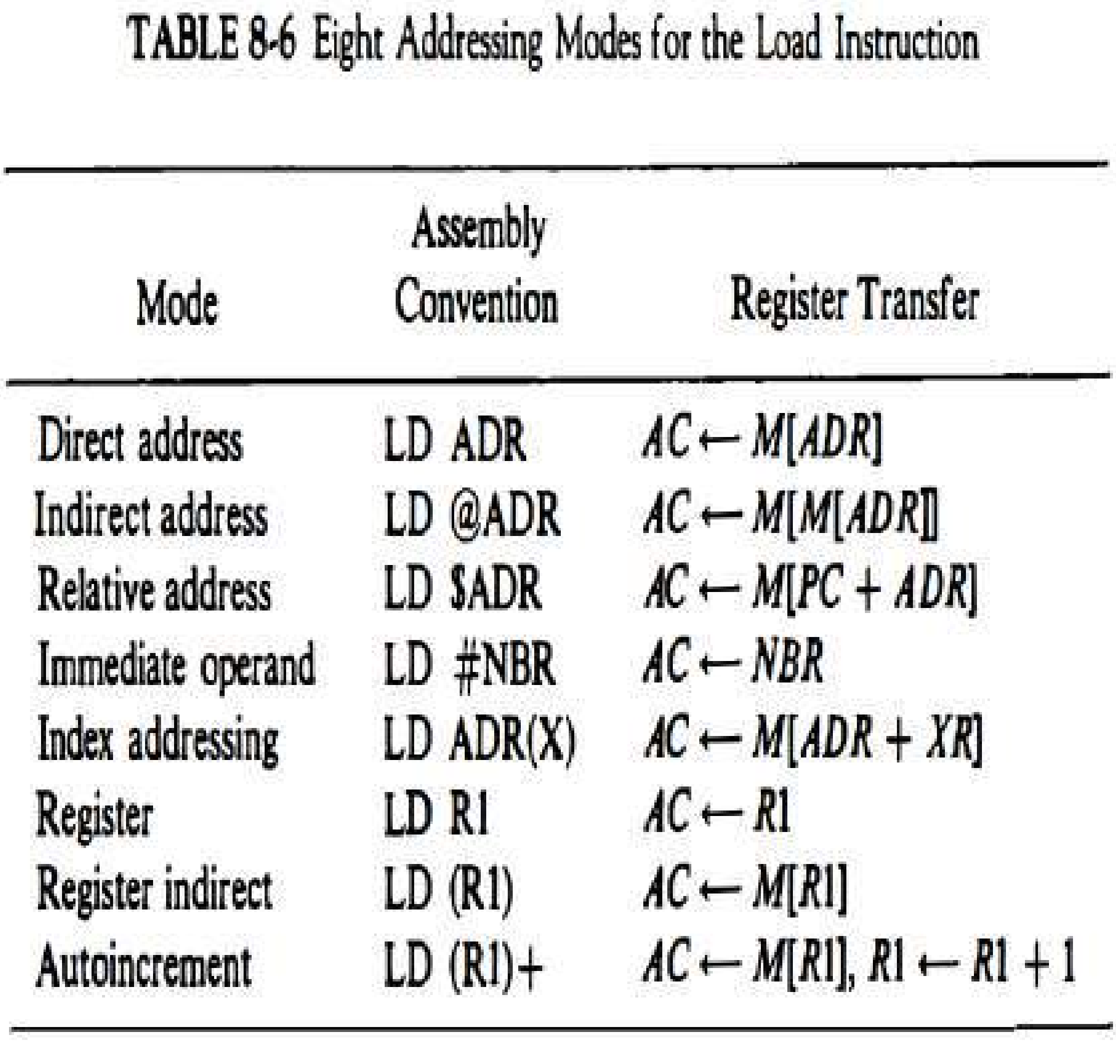


TABLE 8-6 Eight Addressing Modes for the Load Instruction

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

Data Manipulation Instructions:

→Data manipulation instructions perform operations on data and provide the computational capabilities for the computer.

The data manipulation instructions in a typical computer are usually divided into three basic types:

1. Arithmetic instructions

2. Logical and bit manipulation instruction

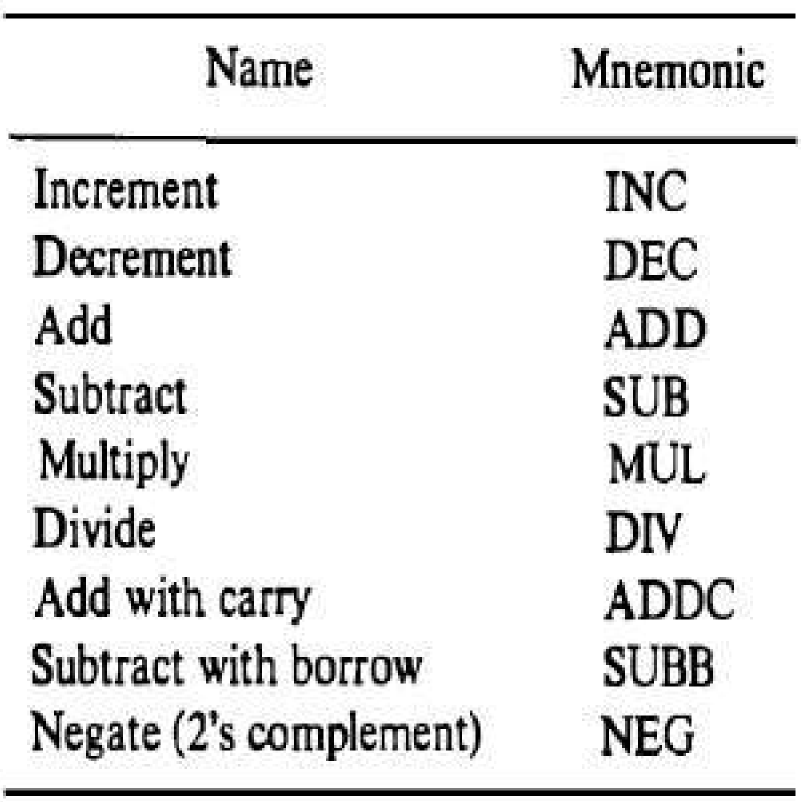
3. Shift instructions

Arithmetic instructions:

→The four 4 basic operations are addition, subtraction, multiplication, and division. Most computers provide instructions for all for operations. Multiplication and Division usually generated using software subroutines.

→A list of typical arithmetic instructions is shown in **Table 8-7**.

TABLE 8-7 Typical Arithmetic Instructions



Name	Mnemonic
Increment	INC
Decrement	DEC
Add	ADD
Subtract	SUB
Multiply	MUL
Divide	DIV
Add with carry	ADDC
Subtract with borrow	SUBB
Negate (2's complement)	NEG

→The mnemonics for three add instructions that specify different data types are shown below.

ADDI Add two binary integer numbers

ADDF Add two floating point numbers

ADDD Add two decimal numbers in BCD

Logical and Bit Manipulation Instructions:

→Logical instructions perform binary operations on strings of bits stored in registers. They are useful for manipulating individual bits or a group of bits that represent binary-coded information. The logical instructions consider each bit of the operand separately and treat it as

Boolean variable. By proper application of the logical instructions it is possible to change bit values, to clear a group of bits, or to insert new bit values into operands stored in registers or memory words.

→Some typical logical and bit manipulation instructions are listed in **Table**

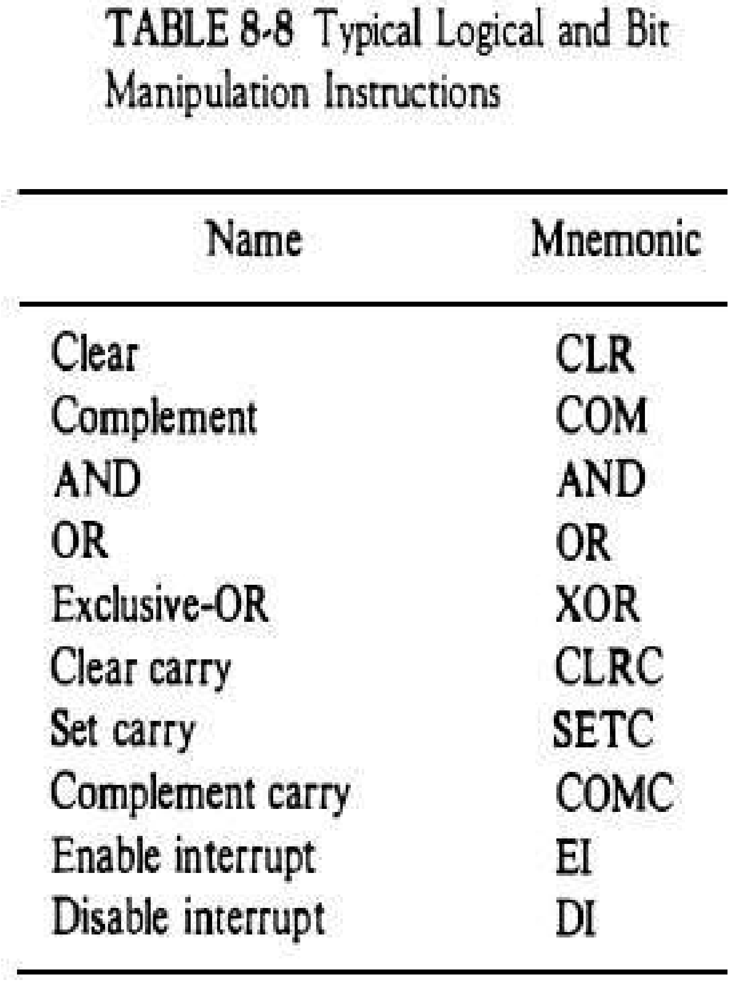


TABLE 8-8 Typical Logical and Bit Manipulation Instructions

Name	Mnemonic
Clear	CLR
Complement	COM
AND	AND
OR	OR
Exclusive-OR	XOR
Clear carry	CLRC
Set carry	SETC
Complement carry	COMC
Enable interrupt	EI
Disable interrupt	DI

Clear selected bits:

→The AND instruction is used to clear a bit or a selected group of bits of an operand. For any Boolean variable x, the relationships x b0 = O and x b1 = x dictate that a binary variable ANDed with a 0 produces a 0; but the variable does not change in value when ANDed with a 1.

→Therefore, the AND instruction can be used to clear bits of an operand selectively by ANDing the operand with another operand that has 0' s in the bit positions that must be cleared. The AND instruction is also called a mask because it masks or inserts 0's in a selected portion of an operand.

Set selected bits:

→The OR instruction is used to set a bit or a selected group of bits of an operand. For any Boolean variable x, the relationships x + 1 = 1 and x + 0 = x dictate that a binary variable ORed with a 1 produces a 1; but the variable does not change when ORed with a 0.

→Therefore, the OR instruction can be used to selectively set bits of an operand by ORing it with another operand with 1' s in the bit positions that must be set to 1.

Complement selected bits:

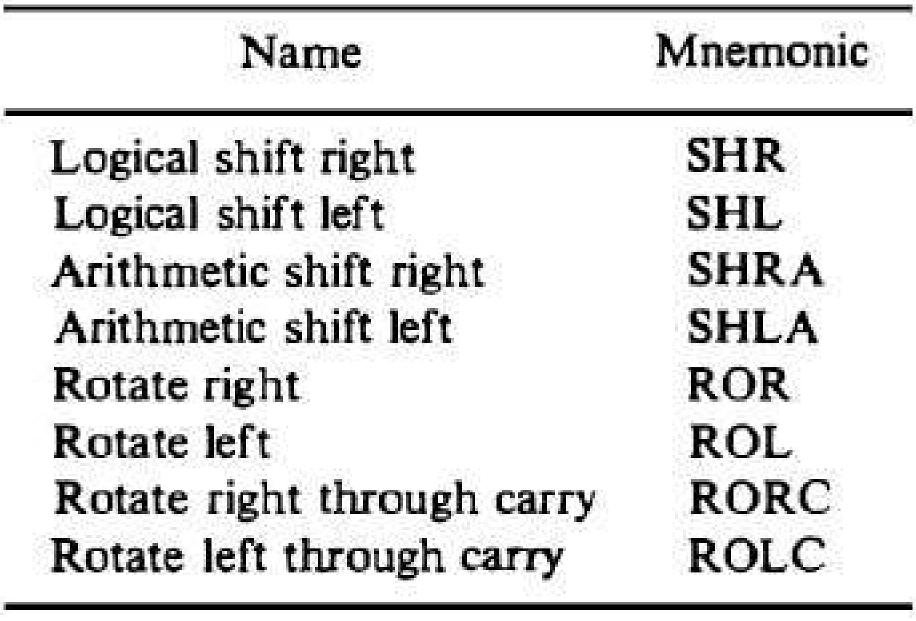
→Similarly, the XOR instruction is used to selectively complement bits of bits an operand. This is because of the Boolean relationships $x \oplus 1 = x'$ and $x \oplus 0 = x.$. Thus a binary variable is complemented when XORed with a 1 but does not change in value when XORed with a 0.

Shift Instructions:

→Instructions to shift the content of an operand are quite useful and are often provided in several variations. Shifts are operations in which the bits of a word are moved to the left or right. The bit shifted in at the end of the word determines the type of shift used. Shift instructions may specify logical shifts, arithmetic shifts, or rotate-type operations. In either case the shift may be to the right or to the left.

→**Table 8-9** lists four types of shift instructions.

TABLE 8-9 Typical Shift Instructions



Name	Mnemonic
Logical shift right	SHR
Logical shift left	SHL
Arithmetic shift right	SHRA
Arithmetic shift left	SHLA
Rotate right	ROR
Rotate left	ROL
Rotate right through carry	RORC
Rotate left through carry	ROLC

→The logical shift inserts 0 to the end bit position. The end position is the leftmost bit for shift right and the rightmost bit position for the shift left. Arithmetic shifts usually conform to the rules for signed-2's complement numbers.

→A possible instruction code format of a shift instruction may include five fields as follows:

OP REG TYPE RL COUNT

→Here OP is the operation code field; REG is a register address that specifies the location of the operand; TYPE is a 2-bit field specifying the four different types of shifts; RL is a 1-bit field specifying a shift right or left; and COUNT is a k-bit field specifying up to 2^k - 1 shifts.

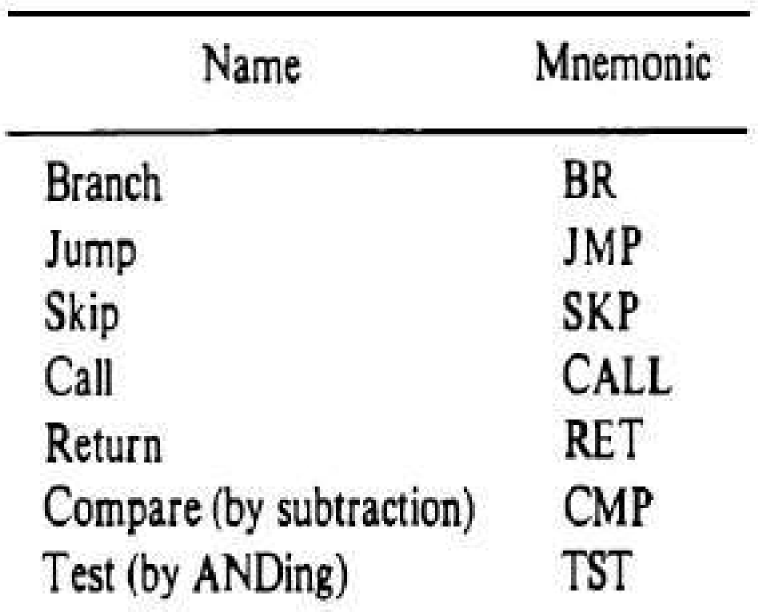
6. Program Control:

→**Program control instructions** provide decision-making capabilities and change the program path. Typically, the program counter is incremented during the fetch phase to the location of the next instruction. A program control type of instruction may change the address value in the program counter and cause the flow of control to be altered.

→This provides control over the flow of program execution and a capability for branching to different program segments.

→Some typical program control instructions are listed in **Table 8-10.**

TABLE 8-10 Typical Program Control Instructions



Name	Mnemonic
Branch	BR
Jump	JMP
Skip	SKP
Call	CALL
Return	RET
Compare (by subtraction)	CMP
Test (by ANDing)	TST

COMPUTER ORGANIZATION CHAPTER-3

→**Branch and Jump instructions** may be conditional or unconditional. An unconditional branch instruction causes a branch to the specified address without any conditions. The conditional branch instruction specifies a condition such as branch if positive or branch if zero. If the condition is met, the program counter is loaded with the branch address and the next instruction is taken from this address. If the condition is not met, the program counter is not changed and the next instruction is taken from the next location in sequence.

→**The Skip instruction** does not need an address field and is therefore a zero-address instruction. A conditional skip will skip the next instruction if the condition is met.

```
SKIP   ON   COND
       BRA   AD1
       BRA   AD2
```

→The **Call and Return instructions** are used in conjunction with subroutines.

→The compare instruction performs a subtraction between two operands but the result of the operation is not retained.

→**Test instruction** performs AND between 2 operands and conditional flags will be changed accordingly.

Status Bit Conditions:

It is sometimes convenient to supplement the ALU circuit in the CPU with a status register where status bit conditions can be stored for further analysis. Status bits are also called *condition-code bits* or *flag bits.*

→**Figure 8-8** shows the block diagram of an 8-bit ALU with a 4-bit status register.

→The four status bits are symbolized by C. S, Z, and V. The bits are set or cleared as a result of an operation performed in the ALU.

1. **Bit C (carry)** is set to 1 if the end carry C_8 is 1.It is cleared to 0 if the carry is zero.

2. **Bit S (sign**) is set to 1 if the highest order bit F_7 is 1. It is set to 0 if the bit is zero.

3. **Bit Z (zero)** is set to 1 if the output of the ALU contains all 0's. It is cleared to 0 otherwise. In other words, Z=1 if the output is zero and Z=0 if the output is not zero.

4. **Bit V (overflow)** is set to 1 if the exclusive-OR of the last two carries is equal to 1, and cleared to 0 otherwise.

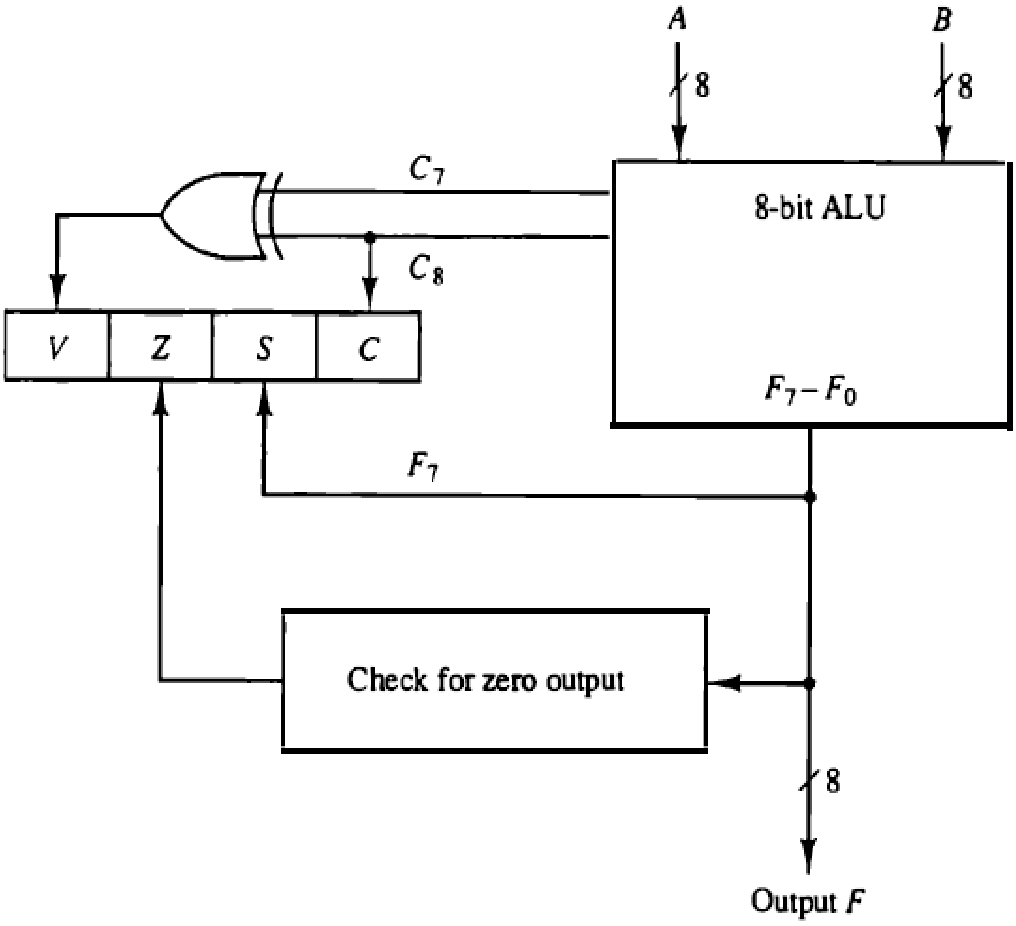


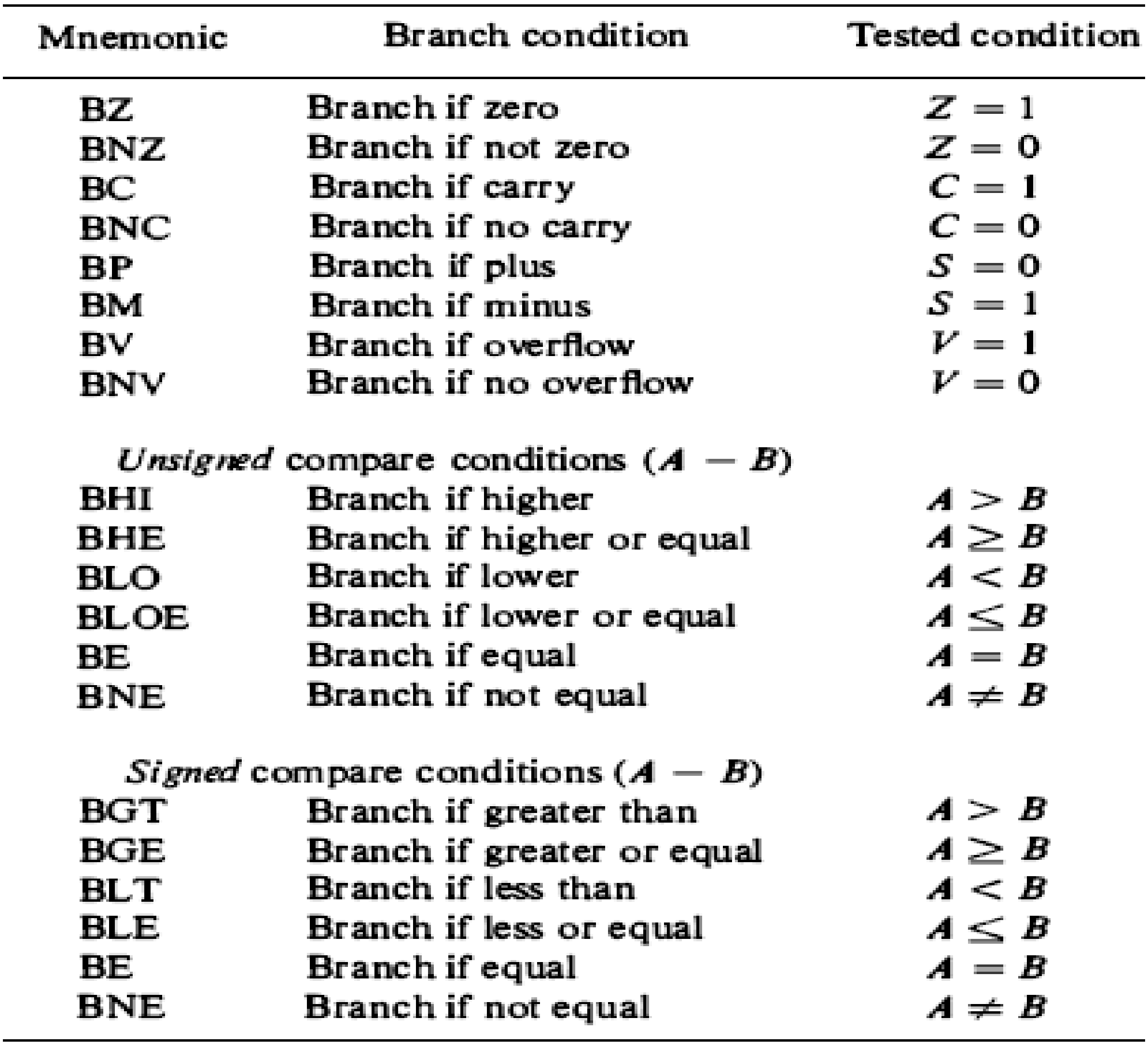
Figure 8-8 Status register bits.

Conditional Branch Instructions:

→**Table 8-11** gives a list of the most common branch instructions. Each mnemonic is constructed with the letter B (for branch) and an abbreviation of the condition name. When the opposite condition state is used, the letter N (for no) is inserted to define the 0 state. Thus BC is Branch on Carry, and BNC is Branch on No Carry. If the stated condition is true, program control is transferred to the address specified by the instruction. If not, control continues with the instruction that follows. The conditional instructions can be associated also with the **jump, skip, call, or return** type of program control instructions.

→The **zero status** bit is used for testing if the result of an ALU operation is equal to zero or not. The carry bit is used to check if there is a carry out of the most significant bit position of the ALU. The **sign bit** reflects the state of the most significant bit of the output from the ALU. S = 0 denotes a positive sign and S = 1, a negative sign.

TABLE 8-11 Conditional Branch Instructions



Mnemonic	Branch condition	Tested condition
BZ	Branch if zero	$Z = 1$
BNZ	Branch if not zero	$Z = 0$
BC	Branch if carry	$C = 1$
BNC	Branch if no carry	$C = 0$
BP	Branch if plus	$S = 0$
BM	Branch if minus	$S = 1$
BV	Branch if overflow	$V = 1$
BNV	Branch if no overflow	$V = 0$
Unsigned compare conditions ($A - B$)		
BHI	Branch if higher	$A > B$
BHE	Branch if higher or equal	$A \geq B$
BLO	Branch if lower	$A < B$
BLOE	Branch if lower or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$
Signed compare conditions ($A - B$)		
BGT	Branch if greater than	$A > B$
BGE	Branch if greater or equal	$A \geq B$
BLT	Branch if less than	$A < B$
BLE	Branch if less or equal	$A \leq B$
BE	Branch if equal	$A = B$
BNE	Branch if not equal	$A \neq B$

COMPUTER ORGANIZATION CHAPTER-3

Subroutine Call and Return:

→A subroutine is a self-contained sequence of instructions that performs a given computational task.

→The instruction that transfers program control to a subroutine is known as **call subroutine, jump to subroutine, branch to subroutine, or branch and save address.**

→The return from subroutine instruction causes the stack to pop and the contents of the top of the stack are transferred to the program counter. In this way, the return is always to the program that last called a subroutine. A subroutine call is implemented with the following micro-operations:

Execution of CALL:

SP ← SP – 1 Decrement stack pointer

M [SP] ← PC Push content of PC onto the stack

PC ← Effective Address Transfer control to the subroutine

→If another subroutine is called by the current subroutine, the new return address is pushed into the stack, and so on. The instruction that returns from the last subroutine is implemented by the micro-operations:

Execution of RET:

PC ← M [SP]	Pop stack and transfer to PC
SP ← SP + 1	Increment stack pointer

Program Interrupt:

→The concept of program interrupt is used to handle a variety of problems that arise out of normal program sequence.

→**Program interrupt** refers to the transfer of program control from a currently running program to another service program as a result of an external or internal generated request. Control returns to the original program after the service program is executed.

→An interrupt procedure is similar to a subroutine call except:

- The interrupt is usually initiated by an internal or external signal rather than from the execution of an instruction.

- The address of the interrupt service routine program is determined by the hardware rather than the address field of an instruction
- An interrupt procedure usually stores all information necessary to define the state of the CPU storing only the program counter.

→The state of the CPU at the end of the execute cycle is determined from:

1. The content of the PC
2. The content of all processor registers
3. The content of certain status conditions

→The collection of all status bit conditions in the CPU is sometimes called a program status word or PSW. The PSW is stored in a separate hardware register and contains the status information that characterizes the state of the CPU.

→When the CPU is executing a program that is part of the operating system, it is said to supervisor mode be in the supervisor or system mode. Certain instructions are privileged and can be executed in this mode only.

→Three types of **interrupts:**

1. **External interrupts**
2. **Internal interrupts**
3. **Software interrupts**

→**External interrupts** come from I/O devices, from a timing device, from a circuit monitoring the power supply, or from any other external source.

→Examples that cause external interrupts are l/0 device requesting transfer of data, l/0 device finished transfer of data, elapsed time of an event, or power failure.

→**Internal interrupts** arise from illegal or erroneous use of an instruction or data. Internal interrupts are also called traps.

→Examples of interrupts caused by internal error conditions are register overflow, attempt to divide by zero, an invalid operation code, stack overflow, and protection violation.

→**External and internal interrupts** are initiated from signals that occur in the hardware of the CPU. A **software interrupt** is initiated by executing an instruction.

→**Software interrupt** is a special call instruction that behaves like an interrupt rather than a subroutine call. It can be used by the programmer to initiate an interrupt procedure at any desired point in the program.

→The most common use of software interrupt is associated with a supervisor call instruction. This instruction provides means for switching from a CPU user mode to the supervisor mode.

7. RISC (Reduced Instruction Set Computers):

→An important aspect of computer architecture is the design of the instruction set for the processor. The instruction set determines the way that machine language programs are constructed. Many computers have instructions sets of about 100 - 250 instructions.

→These computers employ a variety of data types and a large number of addressing modes – **complex instruction set computer (CISC).**

→A **RISC** uses fewer instructions with simple constructs so they can be executed much faster within the CPU without having to use memory as often

→The essential goal of **CISC** architecture is to attempt to provide a single machine instruction for each statement that is written in a high-level language

The major characteristics of CISC architecture are:

1. A large number of instructions-typically from 100 to 250 instructions.
2. Some instructions that perform specialized tasks and are used infrequently.
3. A large variety of addressing modes-typically from 5 to 20 different modes.
4. Variable-length instruction formats.
5. Instructions that manipulate operands in memory.

RISC Characteristics:

→The concept of RISC architecture involves an attempt to reduce execution time by simplifying the instruction set of the computer.

→The major characteristics of a **RISC** processor are:

1. Relatively few instructions.
2. Relatively few addressing modes.
3. Memory access limited to load and store instructions.

4. All operations done within the registers of the CPU.
5. Fixed-length, easily decoded instruction format.
6. Single-cycle instruction execution.
7. Hardwired rather than micro-programmed control.

→A characteristic of RISC processors is their ability to execute one instruction per dock cycle. This is done by overlapping the fetch, decode, and execute phases of two or three instructions by using a procedure referred to as pipe lining. A load or store instruction may require two clock cycles because access to memory takes more time than register operations. Efficient pipelining, as well as a few other characteristics, are sometimes attributed to RISC, although they may exist in non-RISC architectures as well.

→Other characteristics attributed to RISC architecture are:

1. A relatively large number of registers in the processor unit.
2. Use of overlapped register windows to speed-up procedure call and return.
3. Efficient instruction pipeline.
4. Compiler support for efficient translation of high-level language programs
into machine language programs.

Overlapped register windows:

→*Overlapped register windows* are used to pass parameters and avoids the need for saving and restoring register values during procedure calls

A characteristic of some RISC processors is their use of overlapped register windows to provide the passing of parameters and avoid the need for saving and restoring register values. Each procedure call results in the allocation of a new window consisting of a set of registers from the register file for use by the new procedure. Each procedure call activates a new register window by incrementing a pointer, while the return statement decrements the pointer and causes the activation of the previous window. Windows for adjacent procedures have overlapping registers that are shared to provide the passing of parameters and results.

→The concept of overlapped register windows is illustrated in Fig. 8-9. The system has a total of 74 registers. Registers RO through R9 are global registers that hold parameters shared by all procedures. The other 64 registers are divided into four windows to accommodate procedures A, B, C, and D. Each register window consists of 10 local registers and two sets of six registers common to adjacent windows. Local registers are used for local variables. Common registers are used for exchange of parameters and results between adjacent procedures. The common overlapped registers permit parameters to be passed without the actual movement of data. Only one register window is activated at any given time with a pointer indicating the active window. Each procedure call activates a new register window by incrementing the pointer. The high registers of the calling procedure overlap the low registers of the called procedure, and therefore the parameters automatically transfer from calling to called procedure.

→In general, the organization of register windows will have the following relationships:
Number of global registers = G
Number of local registers in each window = L
Number of registers common to two windows = C
Number of windows = W

→The number of registers available for each window is calculated as follows:

Window size = L + 2C + G

The total number of registers needed in the processor is

Register file = (L + C)W + G

→In the example of **Figure. 8-9,** we have G = 10, L = 10, C = 6, and W = 4. The window size is 10 + 12 + 10 = 32 registers, and the register file consists of (10 + 6) x 4 + 10 = 74 registers.

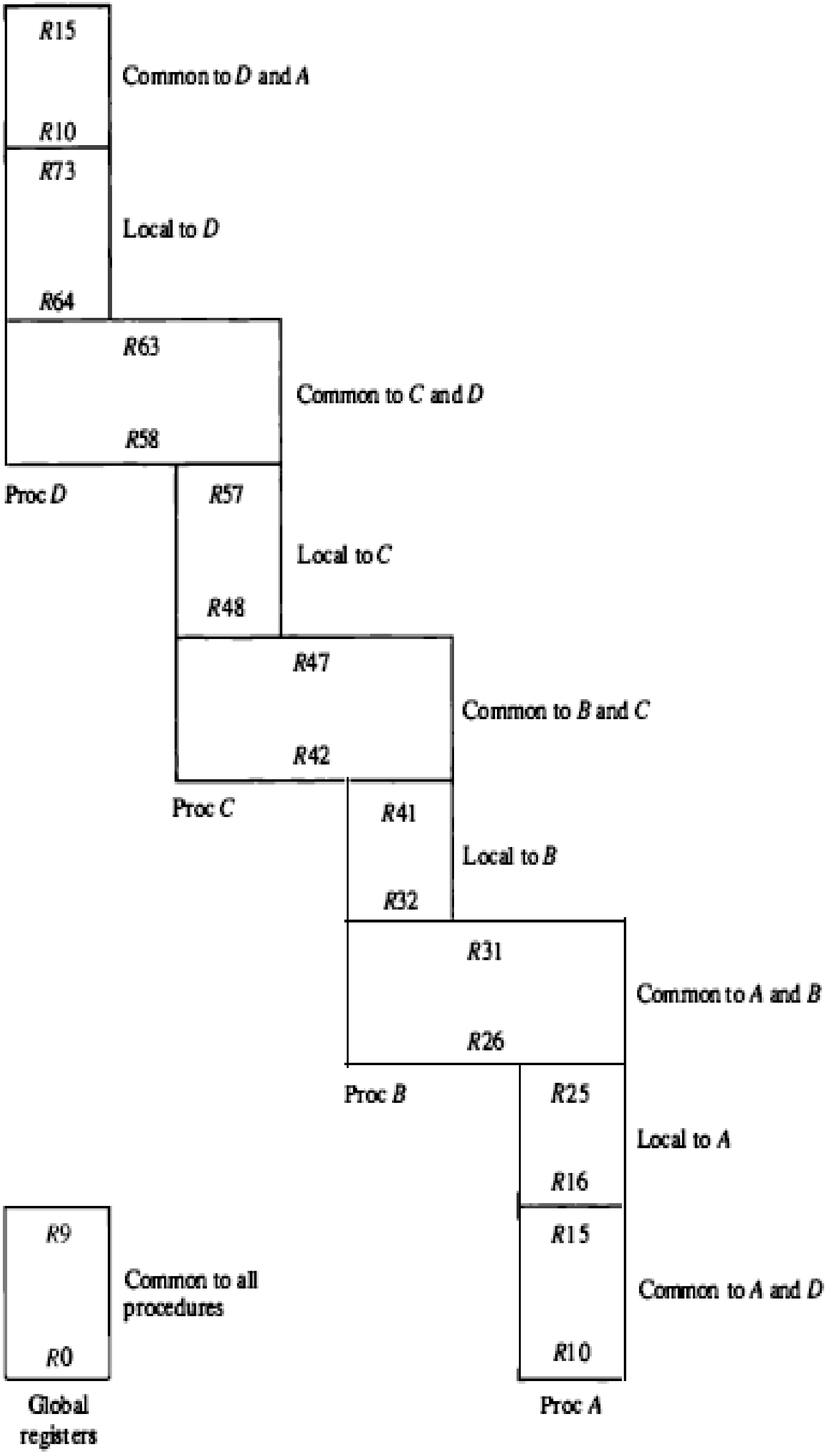


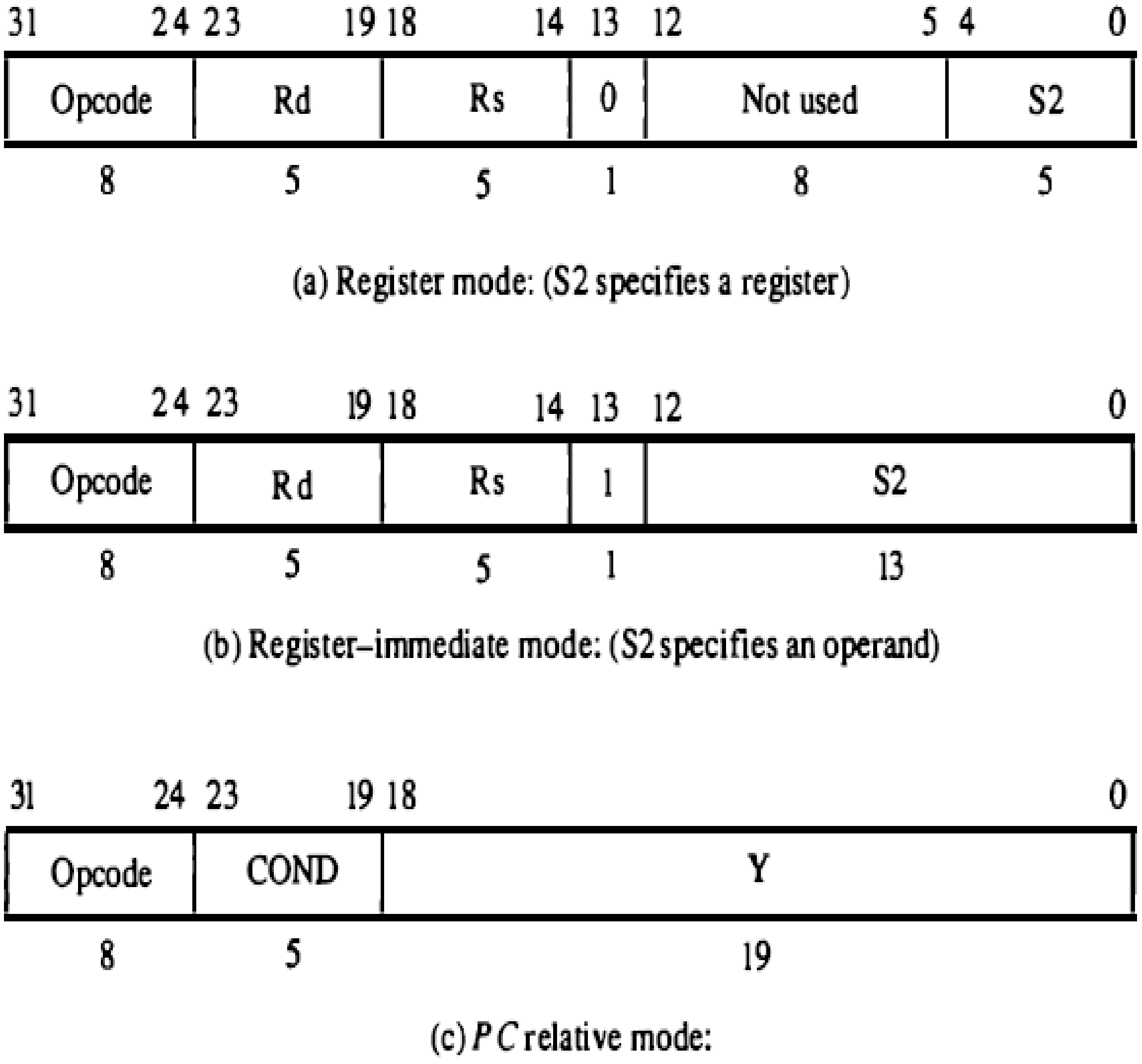
Figure 8-9 Overlapped register windows.

BERKELY RISC I :

→The Berkeley RISC I is a 32-bit integrated circuit CPU. It supports 32-bit addresses and either 8-, 16-, or 32-bit data. It has a 32-bit instruction format and a total of 31 instructions. There are three basic addressing modes: register addressing, immediate operand, and relative to PC addressing for branch instructions. It has a register file of 138 registers arranged into 10 global registers and 8 windows of 32 registers in each. The 32 registers in each window have an organization similar to the one shown in Fig. 8-9. Since only one set of 32 registers in a window is accessed at any given time, the instruction format can specify a processor register with a register field of five bits.

→**Figure 8-10** shows the 32-bit instruction formats used for register-to-register instructions and memory access instructions.

Figure 8-10 Berkeley RISC I instruction formats.



(a) Register mode: (S2 specifies a register)

(b) Register–immediate mode: (S2 specifies an operand)

(c) PC relative mode:

→The 31 instructions of RlSC I are listed in **Table 8-12.** They have been grouped into three categories.

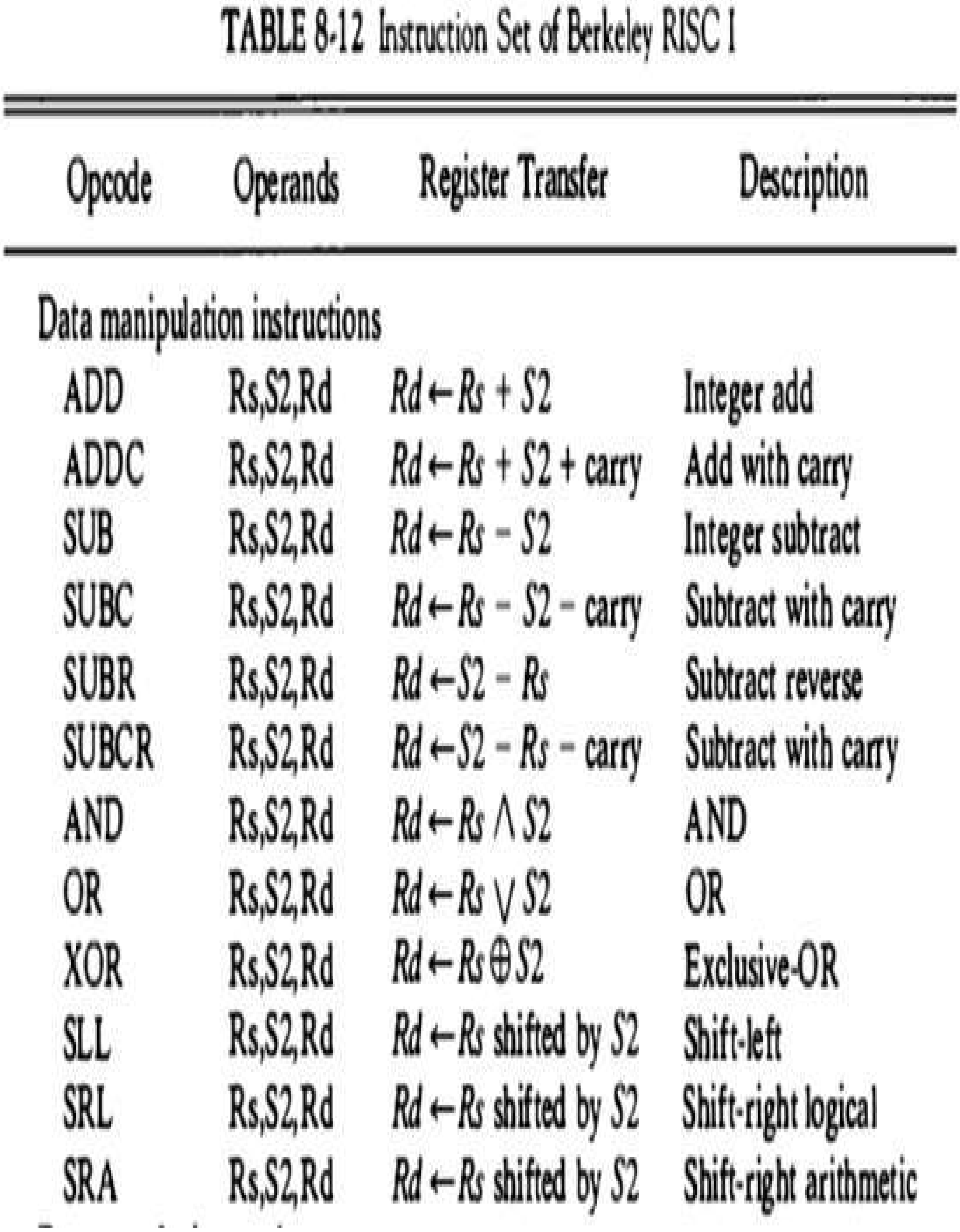
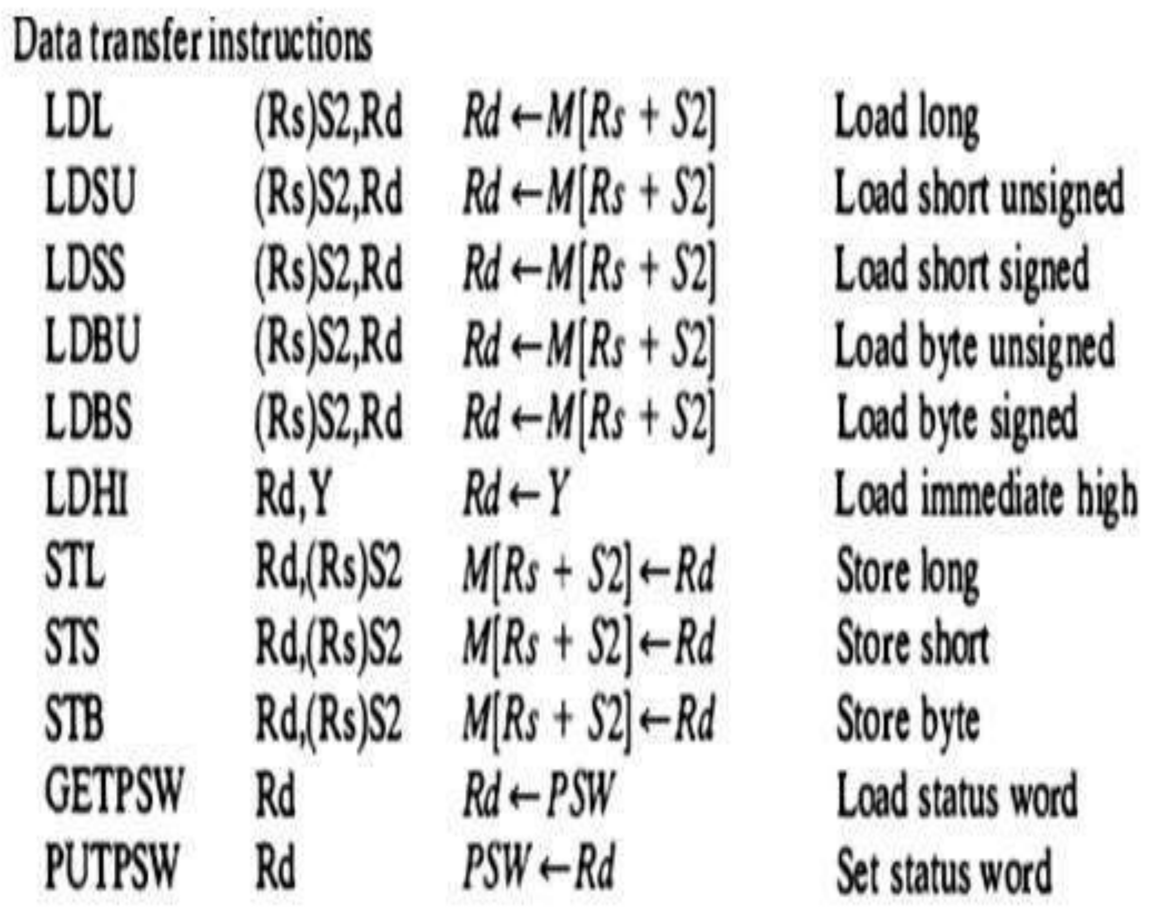


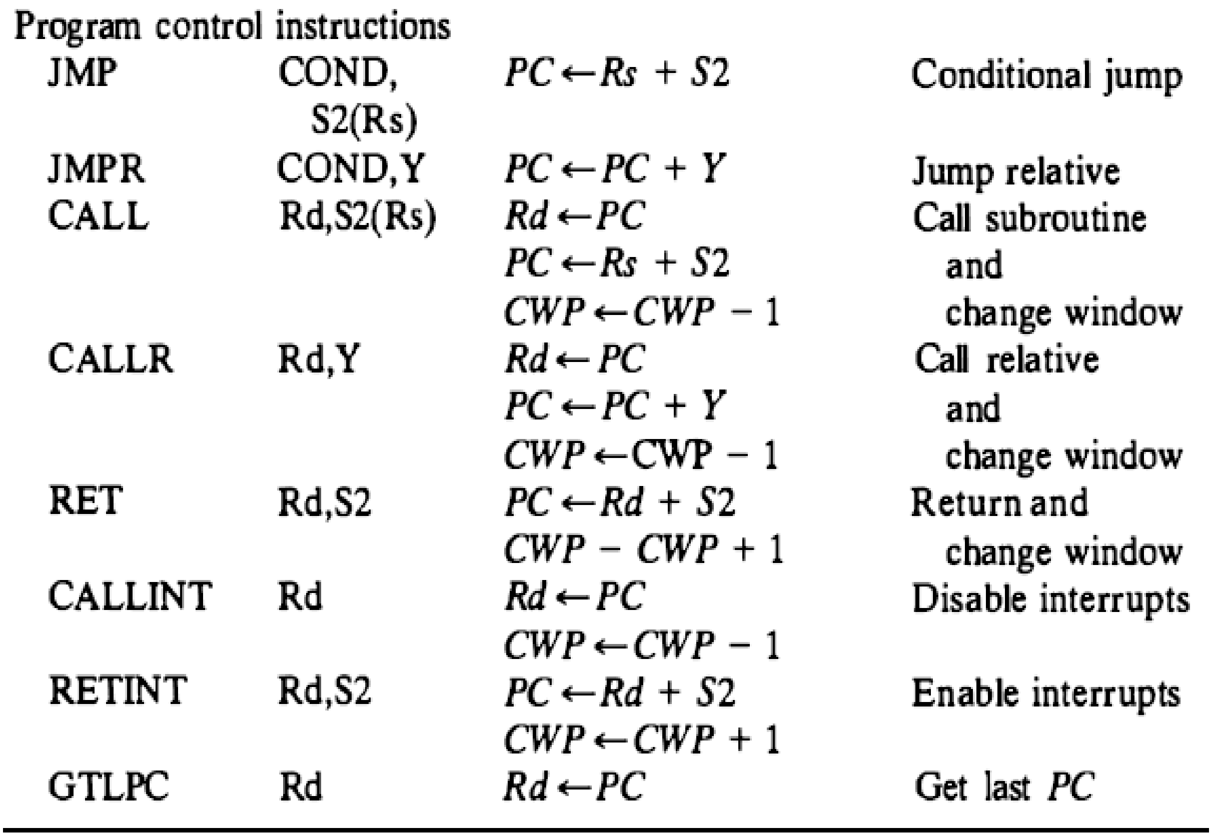
TABLE 8-12 Instruction Set of Berkeley RISC I

Opcode	Operands	Register Transfer	Description
Data manipulation instructions			
ADD	Rs,S2,Rd	$Rd \leftarrow Rs + S2$	Integer add
ADDC	Rs,S2,Rd	$Rd \leftarrow Rs + S2 + \text{carry}$	Add with carry
SUB	Rs,S2,Rd	$Rd \leftarrow Rs - S2$	Integer subtract
SUBC	Rs,S2,Rd	$Rd \leftarrow Rs - S2 - \text{carry}$	Subtract with carry
SUBR	Rs,S2,Rd	$Rd \leftarrow S2 - Rs$	Subtract reverse
SUBCR	Rs,S2,Rd	$Rd \leftarrow S2 - Rs - \text{carry}$	Subtract with carry
AND	Rs,S2,Rd	$Rd \leftarrow Rs \wedge S2$	AND
OR	Rs,S2,Rd	$Rd \leftarrow Rs \vee S2$	OR
XOR	Rs,S2,Rd	$Rd \leftarrow Rs \oplus S2$	Exclusive-OR
SLL	Rs,S2,Rd	$Rd \leftarrow Rs$ shifted by $S2$	Shift-left
SRL	Rs,S2,Rd	$Rd \leftarrow Rs$ shifted by $S2$	Shift-right logical
SRA	Rs,S2,Rd	$Rd \leftarrow Rs$ shifted by $S2$	Shift-right arithmetic



Data transfer instructions

LDL	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load long
LDSU	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load short unsigned
LDSS	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load short signed
LDBU	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load byte unsigned
LDBS	(Rs)S2,Rd	$Rd \leftarrow M[Rs + S2]$	Load byte signed
LDHI	Rd,Y	$Rd \leftarrow Y$	Load immediate high
STL	Rd,(Rs)S2	$M[Rs + S2] \leftarrow Rd$	Store long
STS	Rd,(Rs)S2	$M[Rs + S2] \leftarrow Rd$	Store short
STB	Rd,(Rs)S2	$M[Rs + S2] \leftarrow Rd$	Store byte
GETPSW	Rd	$Rd \leftarrow PSW$	Load status word
PUTPSW	Rd	$PSW \leftarrow Rd$	Set status word



Program control instructions

JMP	COND, S2(Rs)	$PC \leftarrow Rs + S2$	Conditional jump
JMPR	COND,Y	$PC \leftarrow PC + Y$	Jump relative
CALL	Rd,S2(Rs)	$Rd \leftarrow PC$ $PC \leftarrow Rs + S2$ $CWP \leftarrow CWP - 1$	Call subroutine and change window
CALLR	Rd,Y	$Rd \leftarrow PC$ $PC \leftarrow PC + Y$ $CWP \leftarrow CWP - 1$	Call relative and change window
RET	Rd,S2	$PC \leftarrow Rd + S2$ $CWP - CWP + 1$	Return and change window
CALLINT	Rd	$Rd \leftarrow PC$ $CWP \leftarrow CWP - 1$	Disable interrupts
RETINT	Rd,S2	$PC \leftarrow Rd + S2$ $CWP \leftarrow CWP + 1$	Enable interrupts
GTLPC	Rd	$Rd \leftarrow PC$	Get last *PC*

8. Control Memory:

→In digital computer, function of control unit is to initiate sequences of micro-operations. Types of micro-operations for particular system are finite. The complexity of digital system is dependent on the number of sequences of micro-operations that are performed. There are two major types of control organization: **hardwired control and micro programmed control.**

Hardwired Programmed Control Unit:

→We can take one of two approaches to ensure control lines are set properly. The first approach is to physically connect all of the control lines to the actual machine instructions. The instructions are divided up into fields, and different bits in the instruction are combined through various digital logic components to drive the control lines. This is called **hardwired control**, and is illustrated in **Figure (1)**

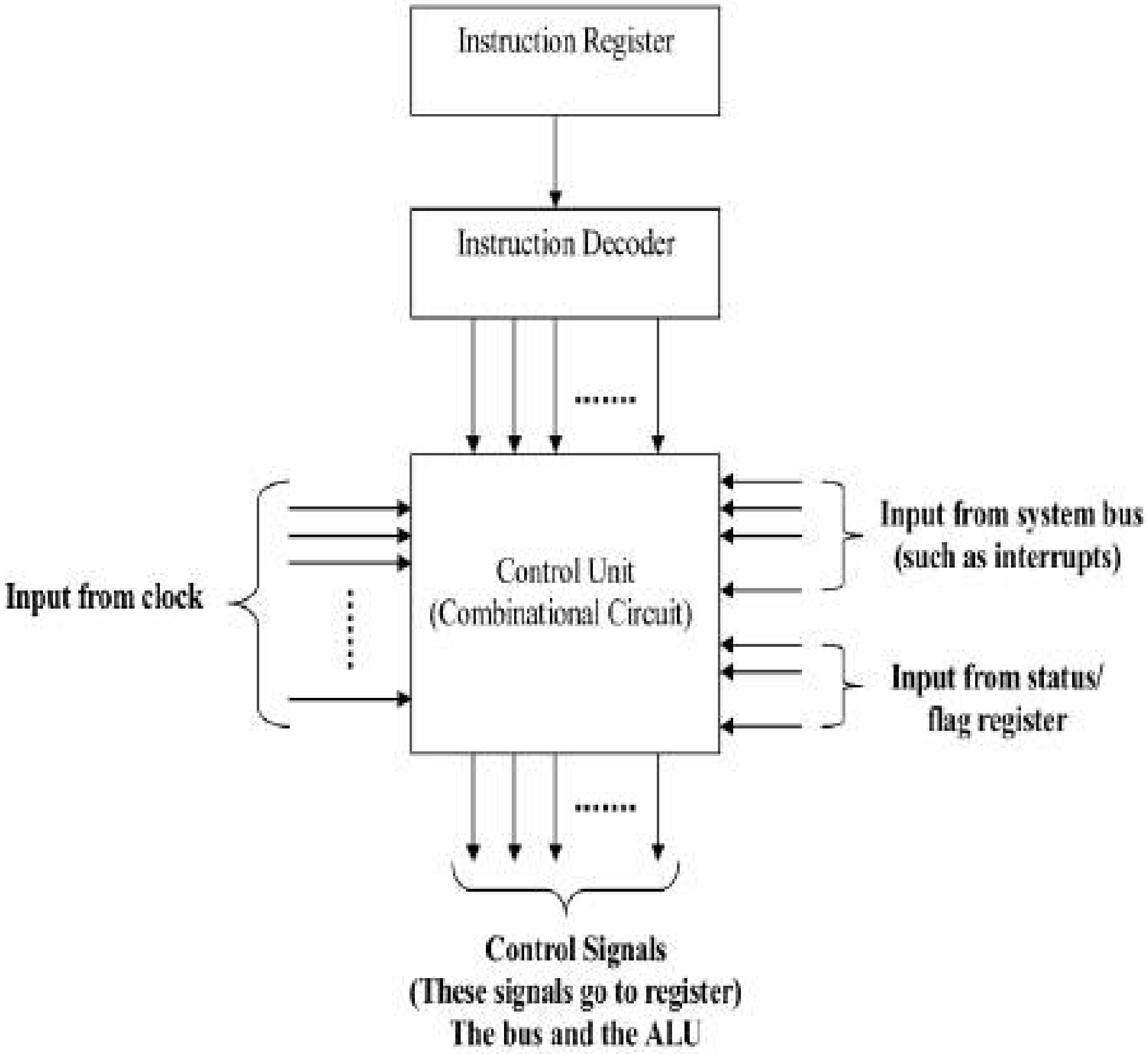


Figure1. Hardwired Control Organization.

COMPUTER ORGANIZATION CHAPTER-3

→In the **hardwired organization**, the control unit is implemented using hardware (for example: NAND gates, flip-flops, and counters).We need a special digital circuit that uses , as inputs, the bits from the operation-code field in our instructions, bits from the flag (or status) register, signals from the bus, and signals from the clock. It should produce, as outputs, the control signals to drive the various components in the computer.

→The **advantage** of hardwired control is that is very fast.

→The **disadvantage** is that the instruction set and the control logic are directly tied together by special circuits that are complex and difficult to design or modify. If someone designs a hardwired computer and later decides to extend the instruction set, the physical components in the computer must be changed. This is prohibitively expensive, because not only must new chips be fabricated but also the old ones must be located and replaced.

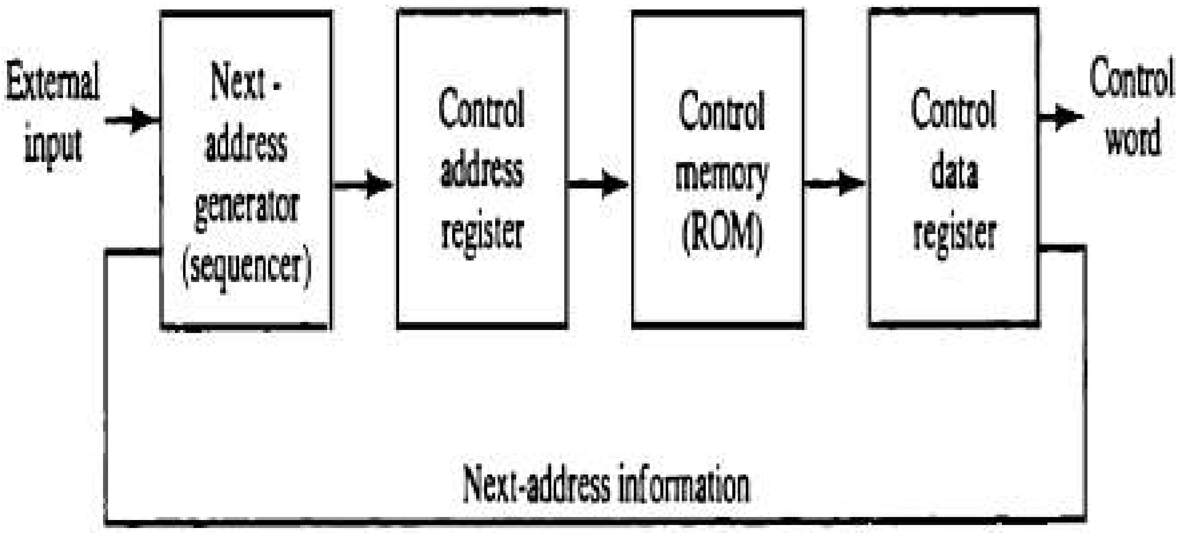
Micro-Programmed Control Unit:

→**Microprogramming** is a second alternative for designing control unit of digital computer (uses software for control). A control unit whose binary control variables are stored in memory is called a **micro-programmed control unit.** The control variables at any given time can be represented by a string of 1's and 0's called a **control word** (which can be programmed to perform various operations on the component of the system). Each word in control memory contains within it a microinstruction. The microinstruction specifies one or more micro-operations for the system. A sequence of microinstructions constitutes a **micro-program**. A memory that is part of a control unit is referred to as a **control memory.**

→A more advanced development known as **dynamic microprogramming** permits a micro-program to be loaded initially from an auxiliary memory such as a magnetic disk. Control units that use dynamic microprogramming employ a writable control memory; this type of memory can be used for writing (to change the micro-program) but is used mostly for reading.

→The general configuration of a micro-programmed control unit is demonstrated in the block diagram of **Figure**

Figure 7-1 Microprogrammed control organization.



→The control memory is assumed to be a ROM, within which all control information is permanently stored. The **control memory address register** specifies the address of the microinstruction and the **control data register** holds the microinstruction read from memory.

→The microinstruction contains a control word that specifies one or more micro-operations for the data processor. Once these operations are executed, the control must determine the next address. The location of the next microinstruction may be the one next in sequence, or it may be locate somewhere else in the control memory. For this reason it is necessary to use some bits of the present microinstruction to control the generation of the address of the next microinstruction. The next address may also be a function of external input conditions. While the micro-operations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction.

→The next address generator is sometimes called a **micro-program sequencer,** as it determines the address sequence that is read from control memory, the address of the next microinstruction can be specified several ways, depending on the sequencer inputs. Typical functions of a micro-program sequencer are incrementing the control address register by one, loading into the control address register an address from control memory, transferring an external address or loading an initial address to start the control operations.

COMPUTER ORGANIZATION CHAPTER-3

→The control data register holds the present microinstruction while the next address is computed and read from memory. The data register is some-times called a **pipeline register.** It allows the execution of the micro-operations specified by the control word simultaneously with the generation of the next microinstruction. This configuration requires a two-phase clock, with one clock applied to the address register and the other to the data register.

→The system can operate without the control data register by applying a single-phase clock to the address register. The control word and next-address information are taken directly from the control memory. It must be realized that a ROM operates as a combinational circuit, with the address value as the input and the corresponding word as the output. The content of the specified word in ROM remains in the output wires as long as its address value remains in the address register. No read signal is needed as in a random-access memory. Each clock pulse will execute the micro-operations specified by the control word and also transfer a new address to the control address register. In the example that follows we assume a single-phase clock and therefore we do not use a control data register. In this way the address register is the only component in the control system that receives clock pulses. The other two components: the sequencer and the control memory are combinational circuits and do not need a clock.

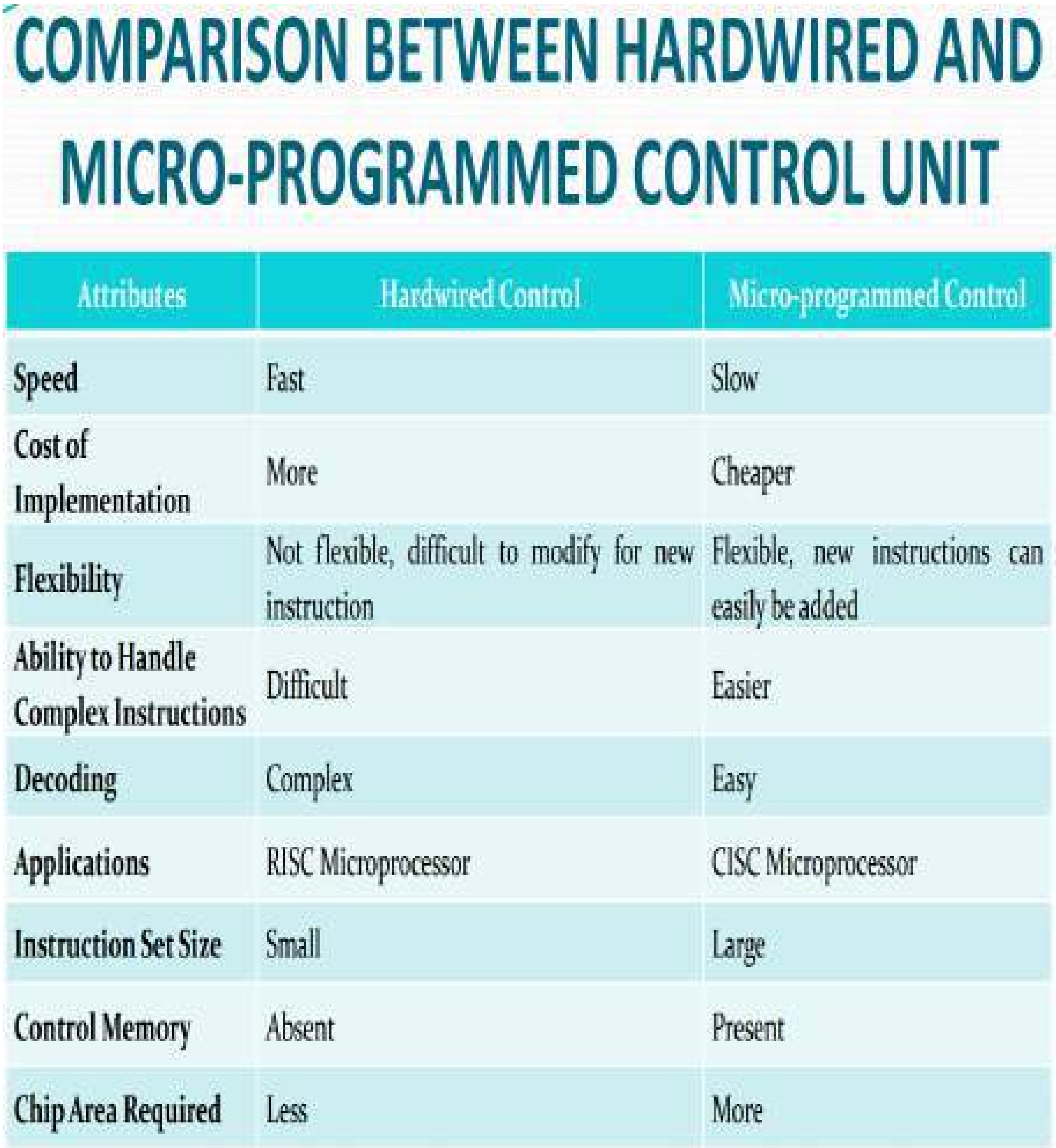
ADVANTAGES:

- The design of micro-program control unit is less complex because micro-programs are implemented using software routines.
- The micro-programmed control unit is more flexible because design modifications, correction and enhancement is easily possible.
- The new or modified instruction set of CPU can be easily implemented by simply rewriting or modifying the contents of control memory.
- The fault can be easily diagnosed in the micro-program control unit using diagnostics tools by maintaining the contents of flags, registers and counters.

DISADVANTAGES:

- The micro-program control unit is slower than hardwired control unit.
- That means to execute an instruction in micro-program control unit requires more time.

- The micro-program control unit is expensive than hardwired control unit in case of limited hardware resources.
- The design duration of micro-program control unit is more than hardwired control unit for smaller CPU.



COMPARISON BETWEEN HARDWIRED AND MICRO-PROGRAMMED CONTROL UNIT

Attributes	Hardwired Control	Micro-programmed Control
Speed	Fast	Slow
Cost of Implementation	More	Cheaper
Flexibility	Not flexible, difficult to modify for new instruction	Flexible, new instructions can easily be added
Ability to Handle Complex Instructions	Difficult	Easier
Decoding	Complex	Easy
Applications	RISC Microprocessor	CISC Microprocessor
Instruction Set Size	Small	Large
Control Memory	Absent	Present
Chip Area Required	Less	More

9. Address Sequencing:

→Microinstructions are stored in control memory in groups, with each group specifying **routine**. Each computer instruction has its own micro-program routine in control memory to generate the micro-operations that execute the instruction. Process of finding address of next micro-instruction to be executed is called address sequencing. Address sequencer must have capabilities of finding address of next micro-instruction in following situations:

- **In-line Sequencing**
- **Unconditional Branch**
- **Conditional Branch**
- **Subroutine call and return**
- **Looping**
- **Mapping from instruction op-code to address in control memory.**

→Steps the control must undergo during the execution of a single computer instruction:

- 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
- The control memory then goes through the routine to determine the effective address of the operand – AR holds operand address
- The next step is to generate the micro-operations that execute the instruction by considering the op-code and applying a *mapping*
- After execution, control must return to the fetch routine by executing an unconditional branch

→In summary, the address sequencing capabilities required in a control memory are:

1. Incrementing of the control addresses register.

2. Unconditional branch or conditional branch, depending on status bit conditions.

3. A mapping process from the bits of the instruction to an address for control memory.

4. A facility for subroutine call and return.

Figure 7-2 shows a block diagram of control memory and the associated hardware needed for selecting the next microinstruction address.

- Control address register (CAR) receives address of next micro instruction from different sources.
- Incrementer simply increments the address by one.
- In case of branching branch address is specified in one of the field of microinstruction.
- In case of subroutine call return address is stored in the register SBR which is used when returning from called subroutine.

Conditional Branching:

→The branch logic of **Figure. 7-2** provides decision making capabilities in control unit.

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

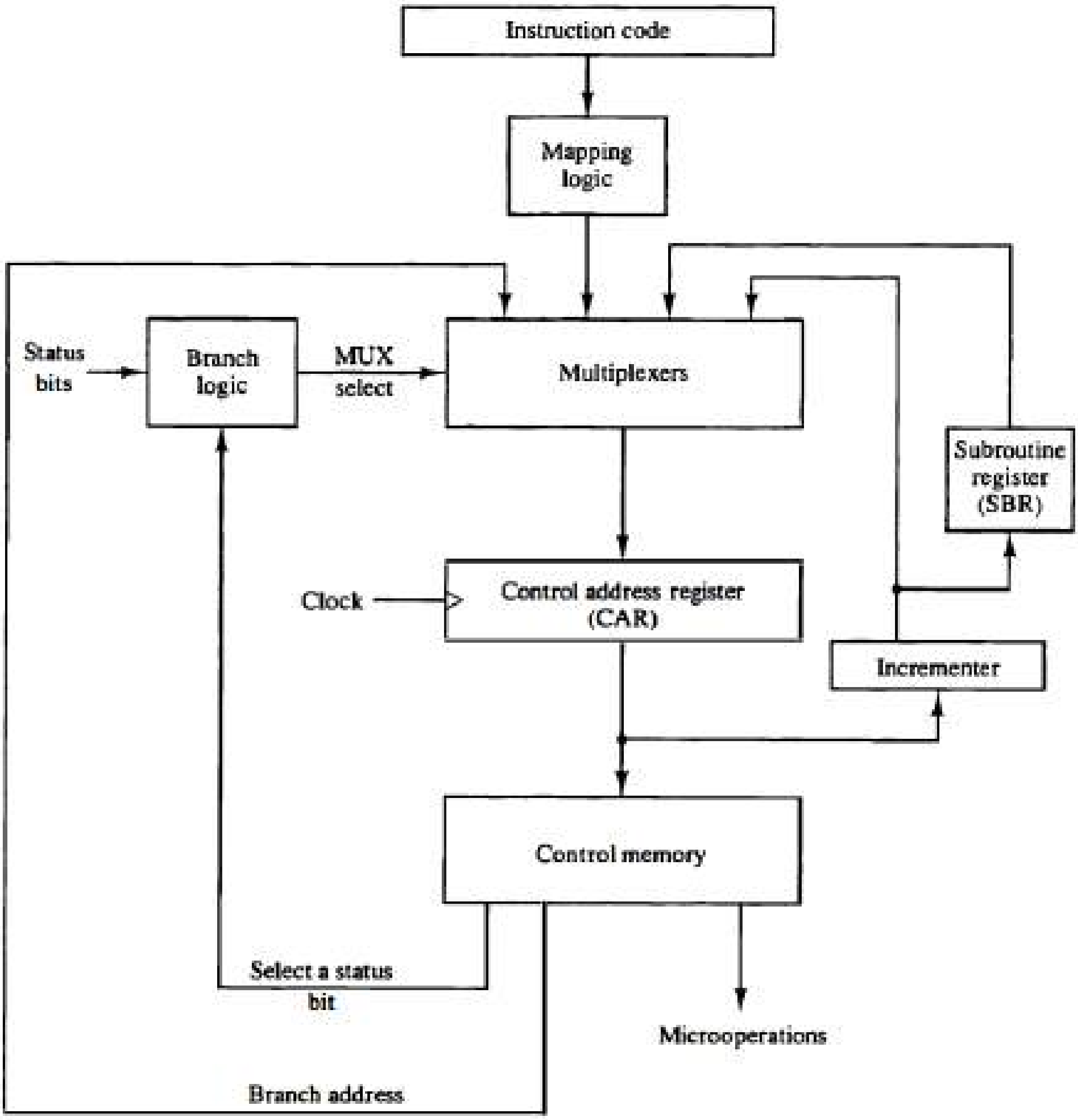


Figure 7-2 Selection of address for control memory.

→Simplest way of implementing branch logic hardware is to test the specified condition and branch to the indicated address if condition is met otherwise address resister is simply incremented. If Condition is true, h/w set the appropriate field of status register to 1. Conditions are tested for O (overflow), N (negative), Z (zero), C (carry), etc.

Unconditional Branching:

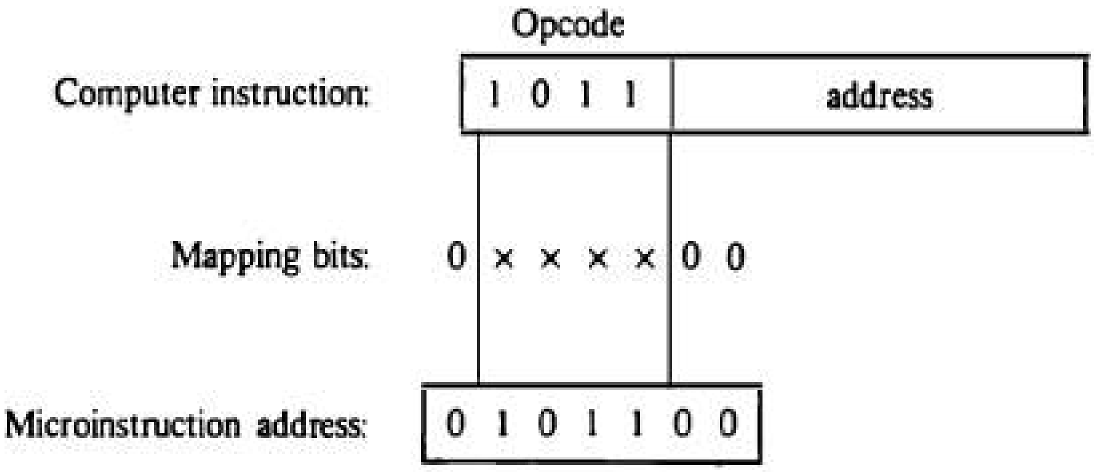
→For **unconditional branching,** fix the value of one status bit to be one load the branch address from control memory into the CAR.

Mapping of Instruction:

→Assuming operation code of 4-bits which can specify 16 (2^4) distinct instructions. Assume further and control memory has 128 words, requiring an address of 7-bits. Now we have to map 4-bit operation code into 7-bit control memory address. Thus, we have to map Op-code of an instruction to the address of the Microinstruction which is the starting microinstruction of its subroutine in memory.

→One simple mapping process that converts the **4-bit operation code** to a **7-bit address** for control memory is shown in **Figure.7-3**. This provides for each computer instruction a micro-program routine with a capacity of four microinstructions.

Figure 7-3 Mapping from instruction code to microinstruction address.



Subroutines:

→ Subroutines are programs that are used by other routines to accomplish a particular task. A **subroutine** can be called from any point within the main body of the micro-program. Frequently many micro-programs contain identical section of code. Microinstructions can be saved by employing subroutines that use common sections of microcode.

→For example, the sequence of microinstructions needed to generate the effective address of the operand for an instruction is common to all memory reference instructions. Thus, this sequence could be a subroutine that is called from within many other routines to execute the effective address computation.

→The **subroutine register** can then become the source for transferring the address for the return to the main routine.

→Subroutine resister is used to save a return address during a subroutine call which is organized in LIFO (last in, first out) stack.

10. Micro-program Example:

→Once we have a configuration of a computer and its micro-programmed control unit, the designer generates the microcode for the control memory. Code generation of this type is called **microprogramming** and is similar to **conventional machine language programming**. We assume here a simple digital computer similar (but not identical) to Manos' basic computer.

Computer Configuration:

→The block diagram of the computer is shown in **Figure 7-4**.It consists of two memory units: a main memory for storing instructions and data, and a control memory for storing micro-program. Four registers are associated with the processor unit and two with the control unit.

→**Four processor registers are:**

- **Program counter – PC**
- **Address register – AR**
- **Data register – DR**
- **Accumulator register - AC**

→**Two control unit registers are:**

- **Control address register – CAR**
- **Subroutine register – SBR**

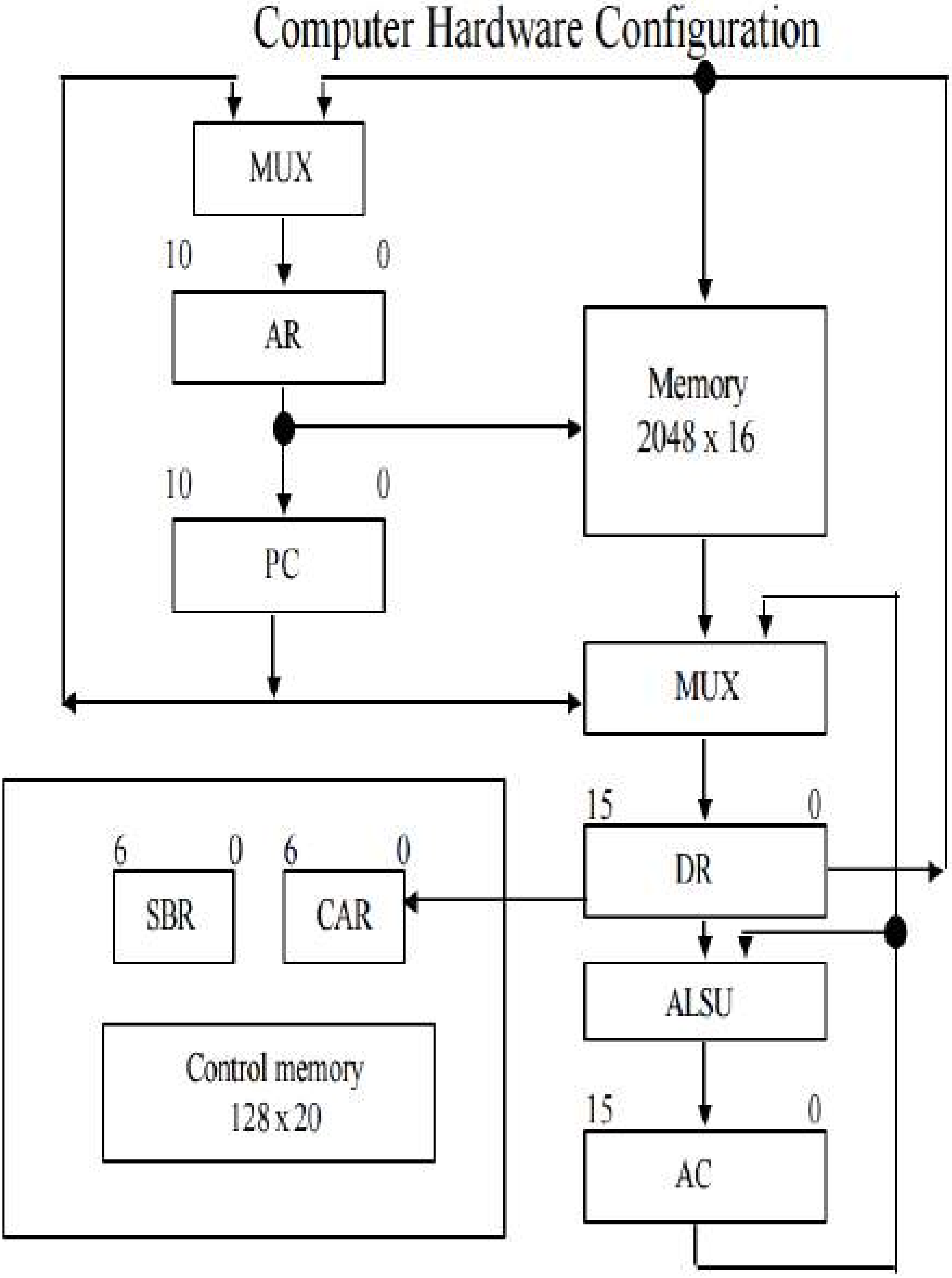


Figure. Computer Hardware Configuration

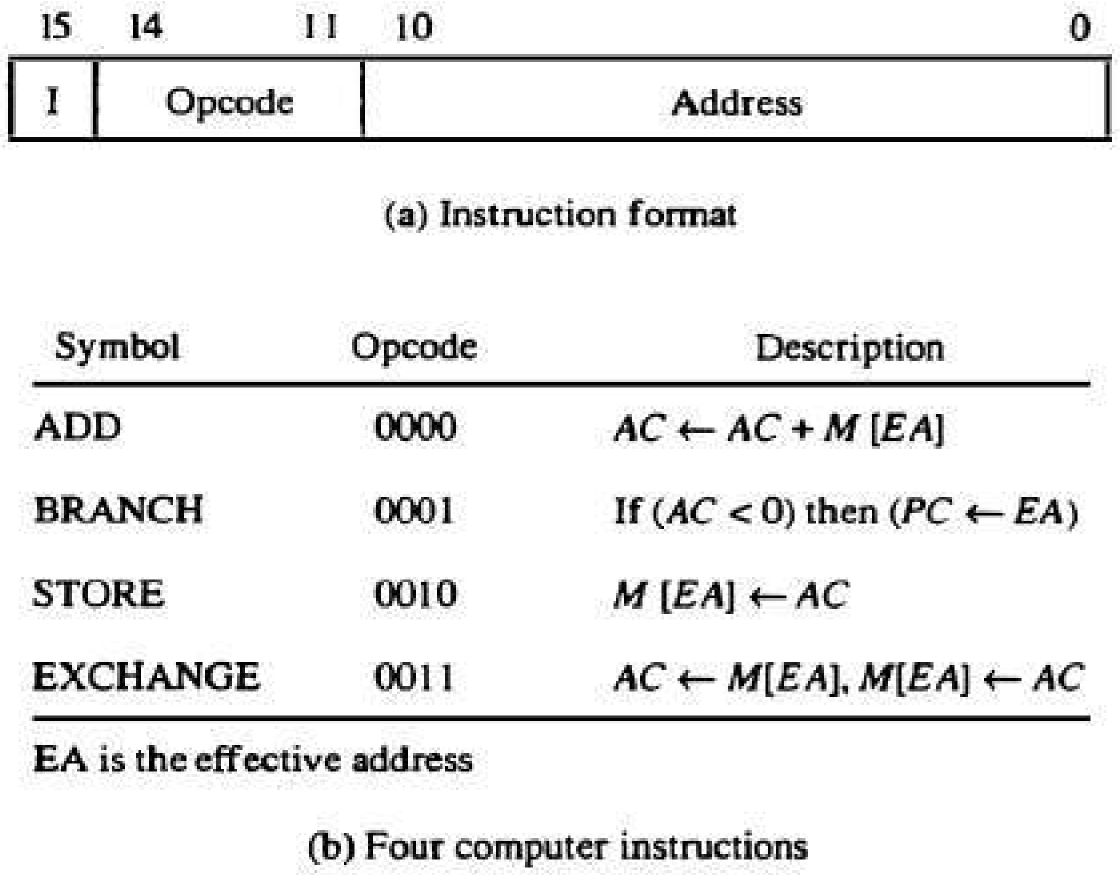
→The transfer of information among the registers in the processor is done through multiplexers rather than a common bus. DR can receive information from AC, PC, or memory. AR can receive information from PC or DR. PC can receive information only from AR. The arithmetic, logic, and shift unit performs micro-operations with data from AC and DR and places the result in AC. Note that memory receives its address from AR. Input data written to memory come from DR, and data read from memory can go only to DR.

→The computer instruction format is depicted in **Figure.** It consists of three fields:

- **1-bit field for indirect addressing.**
- **4-bit operation code.**
- **11-bit address field.**

→**Figure 7-5(b)** lists four of the 16 possible memory-reference instructions.

Figure 7-5 Computer instructions.



(a) Instruction format

Symbol	Opcode	Description
ADD	0000	$AC \leftarrow AC + M[EA]$
BRANCH	0001	If $(AC < 0)$ then $(PC \leftarrow EA)$
STORE	0010	$M[EA] \leftarrow AC$
EXCHANGE	0011	$AC \leftarrow M[EA], M[EA] \leftarrow AC$

EA is the effective address

(b) Four computer instructions

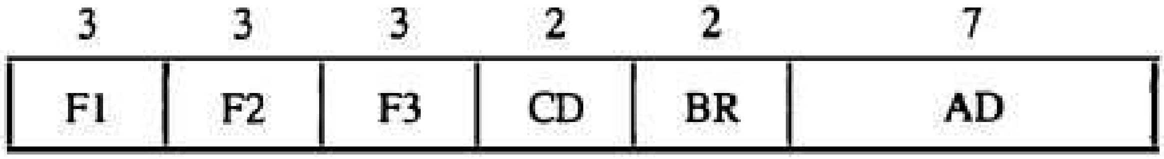
→The ADD instruction adds the content of the operand found in the effective address to the content of AC. The BRANCH instruction causes a branch to the effective address if the operand in AC is negative. The program proceeds with the next consecutive instruction if AC is not negative. The AC is negative if its sign bit (the bit in the leftmost position of the register) is a 1. The STORE instruction transfers the content of AC into the memory word specified by the effective address. The EXCHANGE instruction swaps the data between AC and the memory word specified by the effective address.

Microinstruction Format:

→The microinstruction format for the control memory is shown in **Figure.**

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

- Three fields F1, F2, and F3 specify micro-operations for the computer [3 bits each]
- The CD field selects status bit conditions [2 bits]
- The BR field specifies the type of branch to be used [2 bits]
- The AD field contains a branch address [7 bits]
-



3	3	3	2	2	7
F1	F2	F3	CD	BR	AD

F1, F2, F3: Microoperation fields

CD: Condition for branching

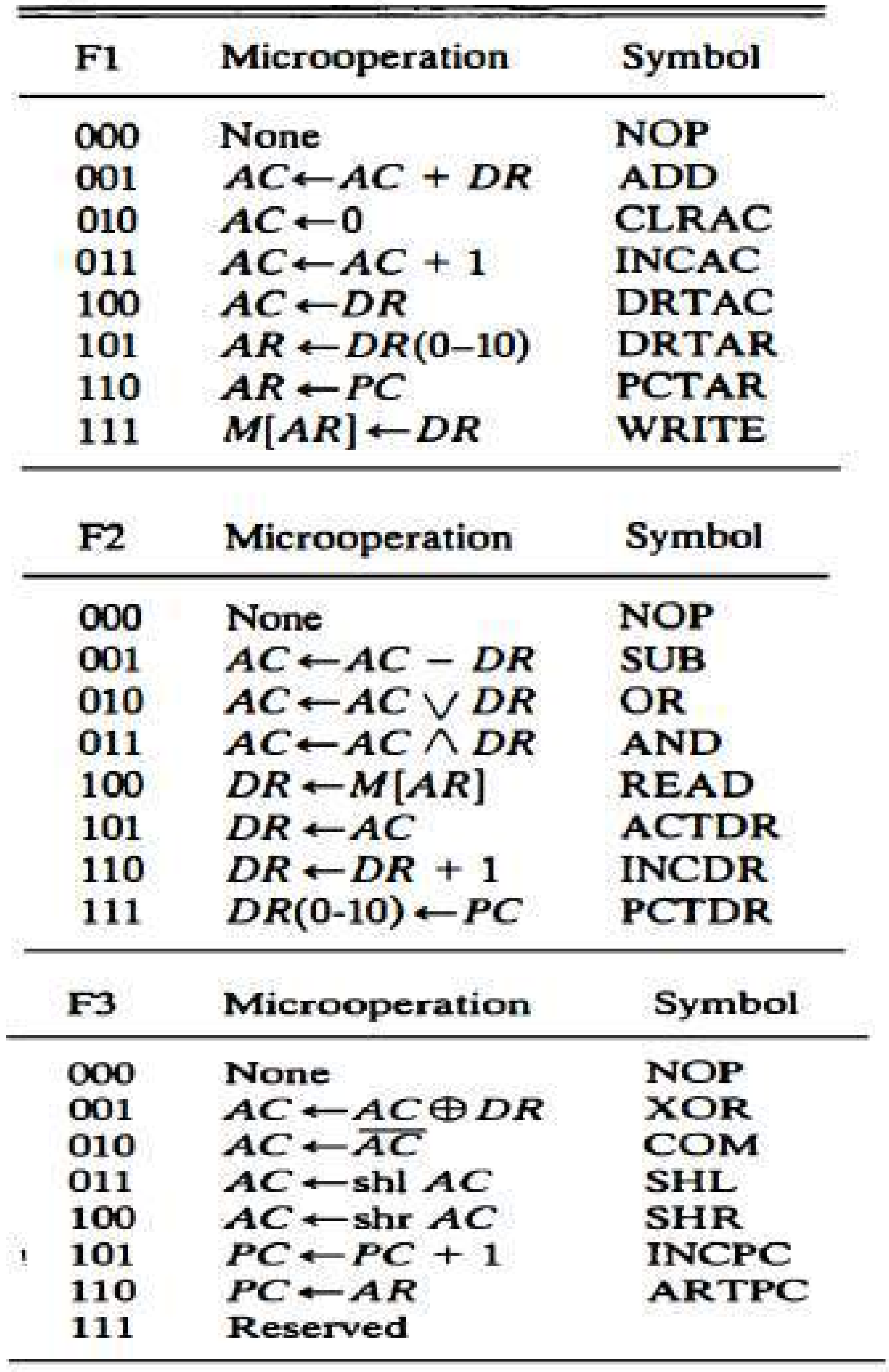
BR: Branch field

AD: Address field

Figure 7-6 Microinstruction code format (20 bits).

→The micro-operations are subdivided into three fields of three bits each. The three bits in each field are encoded to specify seven distinct micro-operations as listed in **Table 7-1**. This gives a total of 21 micro-operations. No more than three micro-operations can be chosen for a microinstruction, one from each field.

TABLE 7-1 Symbols and Binary Code for Microinstruction Fields



F1	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC + DR$	ADD
010	$AC \leftarrow 0$	CLRAC
011	$AC \leftarrow AC + 1$	INCAC
100	$AC \leftarrow DR$	DRTAC
101	$AR \leftarrow DR(0\text{–}10)$	DRTAR
110	$AR \leftarrow PC$	PCTAR
111	$M[AR] \leftarrow DR$	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC - DR$	SUB
010	$AC \leftarrow AC \vee DR$	OR
011	$AC \leftarrow AC \wedge DR$	AND
100	$DR \leftarrow M[AR]$	READ
101	$DR \leftarrow AC$	ACTDR
110	$DR \leftarrow DR + 1$	INCDR
111	$DR(0\text{-}10) \leftarrow PC$	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow \overline{AC}$	COM
011	$AC \leftarrow \text{shl } AC$	SHL
100	$AC \leftarrow \text{shr } AC$	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow AR$	ARTPC
111	Reserved	

→If fewer than three micro-operations are used, one or more of the fields will use the binary code 000 for NOP (No operation). As an illustration, a microinstruction can specify two simultaneous micro-operations from F2 and F3 and none from F1.

DR ← M [AR]   with F2 = 100

And   PC ← PC +1   with F3 = 101

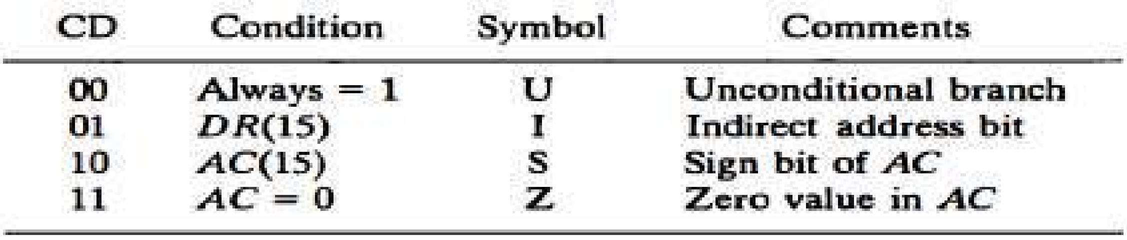
F1   F2   F3   =   000 100 101

→ The **condition field (CD)** is two bits to specify four status bit conditions.

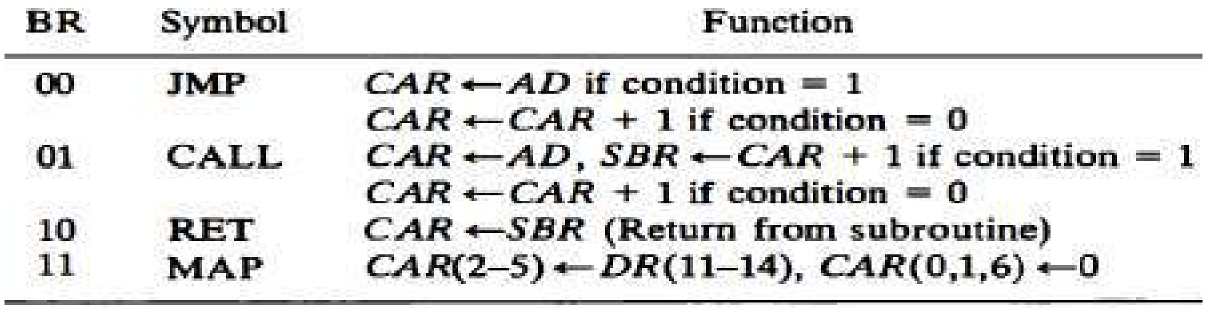
00 Always = 1	**U (Unconditional branch)**
01 DR (15)	**I (Indirect address bit)**
10 AC (15)	**S (Sign bit of AC)**
11 AC = 0	**Z (Zero value in AC)**

→The ***Branch field (BR)*** is two bits and is used with the address field to choose the address of the next microinstruction

BR = 00	**JMP**	**CAR ← AD if condition =1**
		CAR ← CAR + 1 if condition=0
BR = 01	**CALL**	**CAR ← AD, SBR ← CAR + 1 if condition =1**
		CAR ← CAR +1 if condition=0
BR = 10	**RET**	**CAR ← SBR (Return from subroutine)**
BR = 11	**MAP**	**CAR (2-5) ← DR (11-14), CAR (0, 1, 6) ← 0**



CD	Condition	Symbol	Comments
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*



BR	Symbol	Function
00	JMP	*CAR* ← *AD* if condition = 1
		CAR ← *CAR* + 1 if condition = 0
01	CALL	*CAR* ← *AD*, *SBR* ← *CAR* + 1 if condition = 1
		CAR ← *CAR* + 1 if condition = 0
10	RET	*CAR* ← *SBR* (Return from subroutine)
11	MAP	*CAR*(2–5) ← *DR*(11–14), *CAR*(0,1,6) ← 0

Symbolic Microinstructions:

→Each line of an assembly language micro-program defines a symbolic microinstruction and is divided into five fields: label, micro-operations, CD, BR, and AD.

The fields specify the following information:

1. The label field may be empty or it may specify a symbolic address. A label is terminated with a colon (:).
2. 2. The micro-operations field consists of one, two, or three symbols, separated by commas, from those defined in Table 7-1. There may be no more than one symbol from each F field. The NOP symbol is used when the microinstruction has no micro-operations. This will be translated by the assembler to nine zeros.
3. The CD field has one of the letters U, I, S, or Z.
4. The BR field contains one of the four symbols defined in **Table**
5. The AD field specifies a value for the address field of the microinstruction in one of three possible ways:

   a. With a symbolic address, this must also appear as a label.

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

   c. When the BR field contains a RET or MAP symbol, the AD field is left empty and is converted to seven zeros by the assembler.

→The symbol ORG defines the first address of a micro-program routine ORG 64 – places first microinstruction at control memory 1000000.

Fetch Routine:

→During FETCH Read an instruction from memory and decode the instruction and update PC.

→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. A convenient starting location for fetch routine is 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

→Write the symbolic micro program for the fetch routine as follows:

```
                ORG 64
Fetch:          PCTAR           U       JMP     NEXT
                READ, INCPC     U       JMP     NEXT
                DRTAR           U       MAP
```

→The translation of the symbolic micro-program to binary produces the following binary micro-program.

Binary 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

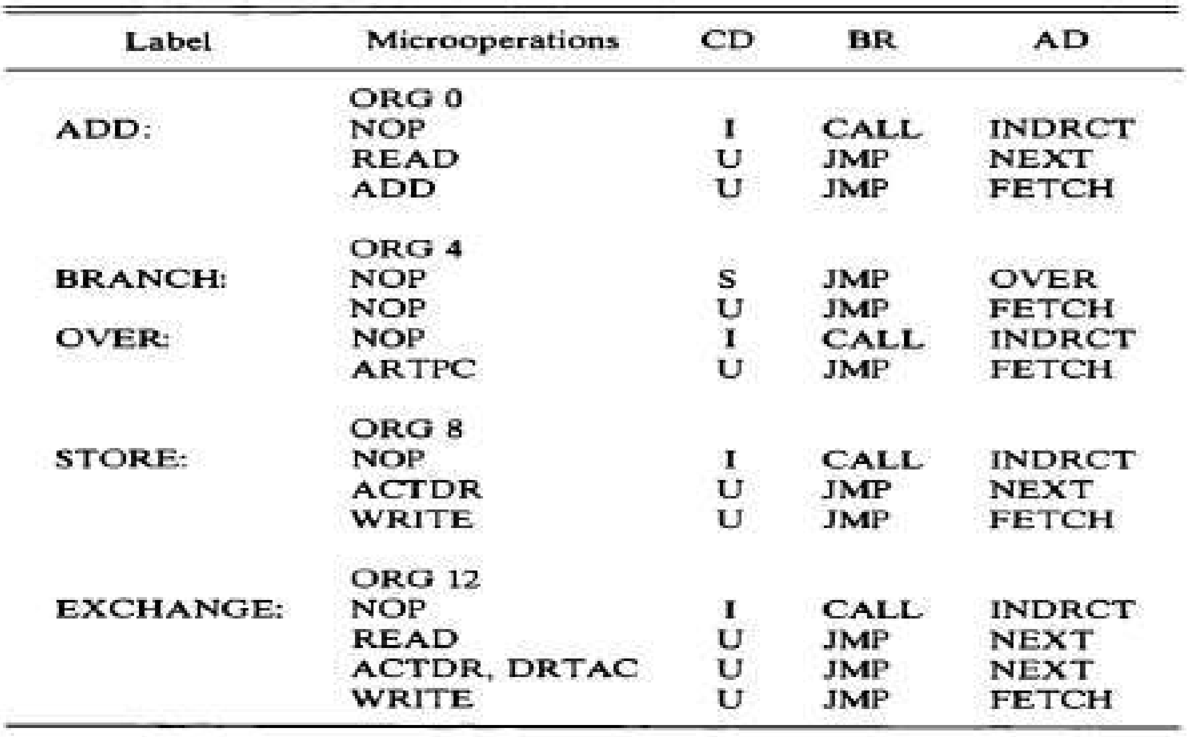
Symbolic Micro-program:

→The execution of the third MAP microinstruction in the fetch routine results in a branch to address 0XXXX00, where XXXX are the four bits of operation code. For example,

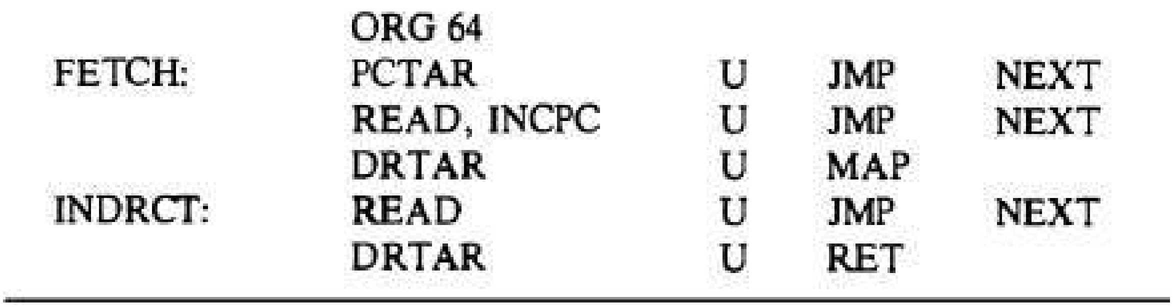
- Control Storage: 128 20-bit words
- The first 64 words: Routines for the 16 machine instructions
- The last 64 words: Used for other purpose (e.g., fetch routine and other subroutines)
- Mapping: Op-code XXXX into 0XXXX00, the first address for the 16 routines is 0(0 0000 00), 4(0 0001 00), 8(0 0010 00), 12, 16, 20, ...,60.

→This gives 4 words in control memory for each routine

TABLE 7-2 Symbolic Microprogram (Partial)



Label	Microoperations	CD	BR	AD
	ORG 0			
ADD:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
	ORG 4			
BRANCH:	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
OVER:	NOP	I	CALL	INDRCT
	ARTPC	U	JMP	FETCH
	ORG 8			
STORE:	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH
	ORG 12			
EXCHANGE:	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH



```
            ORG 64
FETCH:      PCTAR            U    JMP    NEXT
            READ, INCPC      U    JMP    NEXT
            DRTAR            U    MAP
INDRCT:     READ             U    JMP    NEXT
            DRTAR            U    RET
```

→Execution of ADD instructions

Indirect Routine:

→We need to calculate the indirect procedure for accessing the address of operand in all memory-reference instructions.

→So INDRCT routine has been placed in a subroutine that can be called in every memory-reference instruction

→It is only can be called if I=1 then a branch to NDRCT occurs (See Table)

→So in this cycle memory is accessed to get the effective address of operand

- For ADD instruction, the microinstructions in locations 1 and 2 will carry out.
- Reads operand from memory into DR
- Add contents of DR to AC
- Jumps to fetch routine

→Execution of BRANCH instructions

- It causes a branch to effective address of instruction if AC < 0 which means S = 1 Starts by checking the value of S. if S=0 then no branch occurs and next microinstruction causes a jump to fetch routine
- If S=1 then control goes to location OVER where a call to INDRCT takes place first if I=1 then control goes to where AR register points to.

→Execution of STORE instructions

- Uses INDRCT routine if I=1
- Content of AC is transferred to DR then memory write operation is initiated to store DR into MEM(AR)

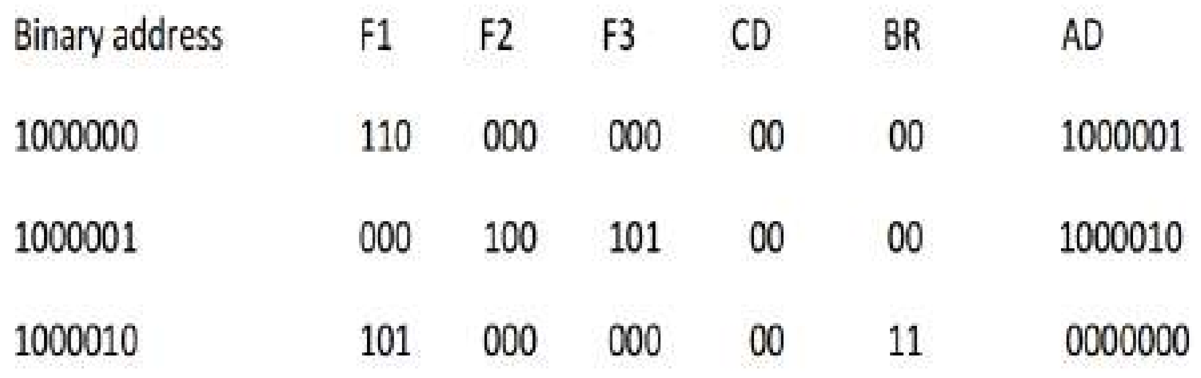
→Execution of EXCHANGE instructions

- Exchange content of MEM(AR) with AC
- Jumps to INDRCT routine to get the effective operand address
- Reads from memory the operand and direct it to DR
- AC and DR are exchanged in third microinstruction
- DR is written to memory MEM(AR)

Binary Micro-program:

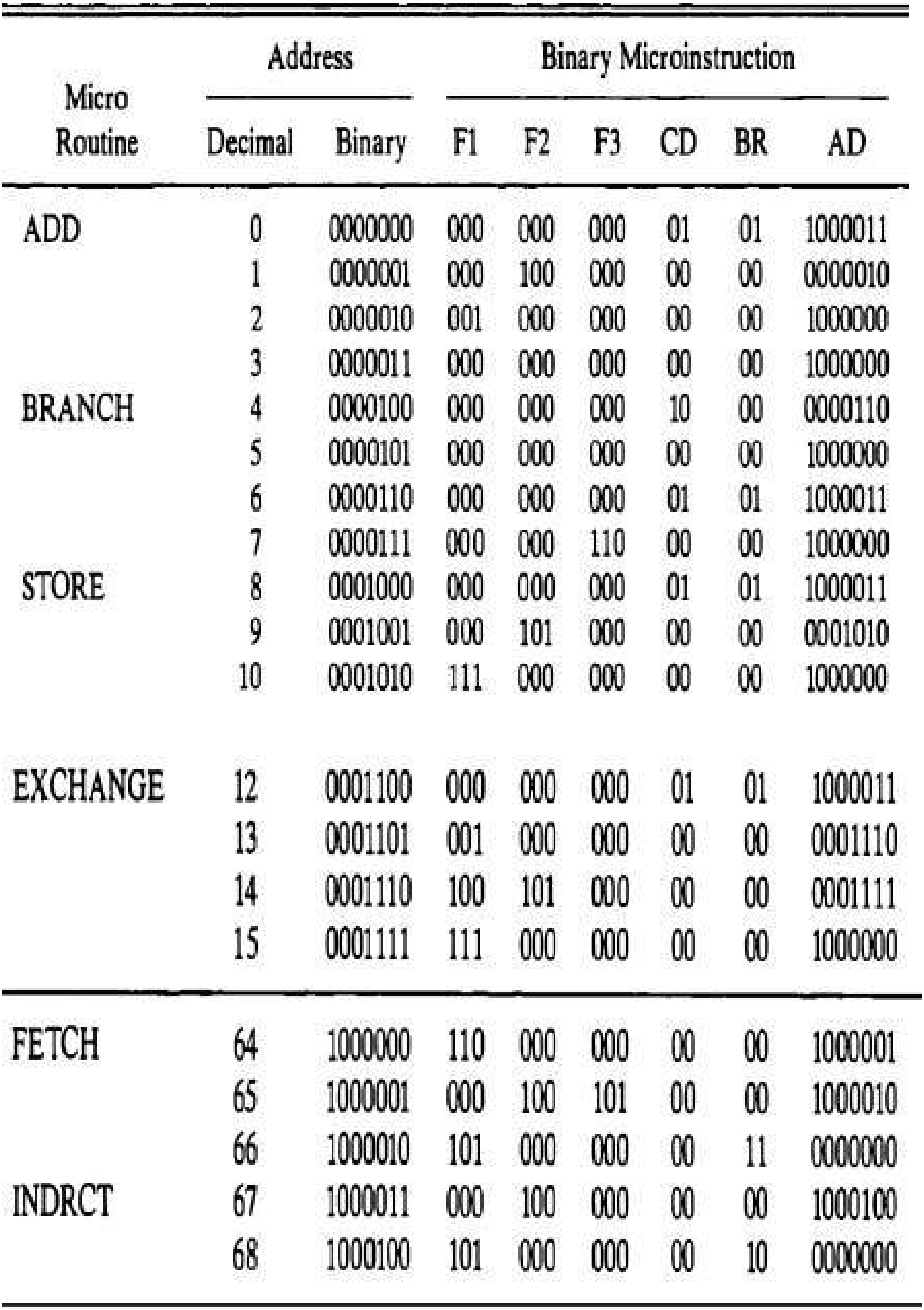
→The symbolic micro-program is a convenient form for writing micro-programs in a way that people can understand. But this is not a way that the micro-program is stored in memory. The symbolic program must be translated to binary either by means of an assembler program or by the user.

→Binary equivalent of a micro-program translated by an assembler for fetch cycle:



Binary 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

TABLE 7-3 Binary Microprogram for Control Memory (Partial)



Micro Routine	Address		Binary Microinstruction					
	Decimal	Binary	F1	F2	F3	CD	BR	AD
ADD	0	0000000	000	000	000	01	01	1000011
	1	0000001	000	100	000	00	00	0000010
	2	0000010	001	000	000	00	00	1000000
	3	0000011	000	000	000	00	00	1000000
BRANCH	4	0000100	000	000	000	10	00	0000110
	5	0000101	000	000	000	00	00	1000000
	6	0000110	000	000	000	01	01	1000011
	7	0000111	000	000	110	00	00	1000000
STORE	8	0001000	000	000	000	01	01	1000011
	9	0001001	000	101	000	00	00	0001010
	10	0001010	111	000	000	00	00	1000000
EXCHANGE	12	0001100	000	000	000	01	01	1000011
	13	0001101	001	000	000	00	00	0001110
	14	0001110	100	101	000	00	00	0001111
	15	0001111	111	000	000	00	00	1000000
FETCH	64	1000000	110	000	000	00	00	1000001
	65	1000001	000	100	101	00	00	1000010
	66	1000010	101	000	000	00	11	0000000
INDRCT	67	1000011	000	100	000	00	00	1000100
	68	1000100	101	000	000	00	10	0000000

11. Design of Control Unit:

→The number of control bits that initiate micro-operations can be reduced by grouping mutually exclusive variables into fields and encode k bits into 2k micro-operations.

Decoding of F fields:

→The 9-bits of the micro-operation field are divided into 3 subfields of 3 bits each. The control memory output of each subfield must be decoded to provide distinct micro-operations. The outputs of the decoders are connected to the appropriate inputs in the processor unit.

→**Figure 7-7** shows 3 decoders of 3X8 type, so each field gives 8control lines.

→E.g. when F1=101 (binary 5), next clock pulse transition transfers the content of DR(0-10) to AR (DRTAR). Similarly when F1=110(6), there is a transfer from PC to AR (PCTAR). Outputs 5 & 6 of decoder F1 are connected to the load inputs of AR so that when either is active information from multiplexers is transferred to AR.

Arithmetic-Logic-shift Unit:

→Arithmetic logic shift unit instead of using gates to generate control signals, is provided inputs with outputs of decoders (AND, ADD and ARTAC).

→Output from the 3 decoders will generate control signals to indicate the micro-operation performed

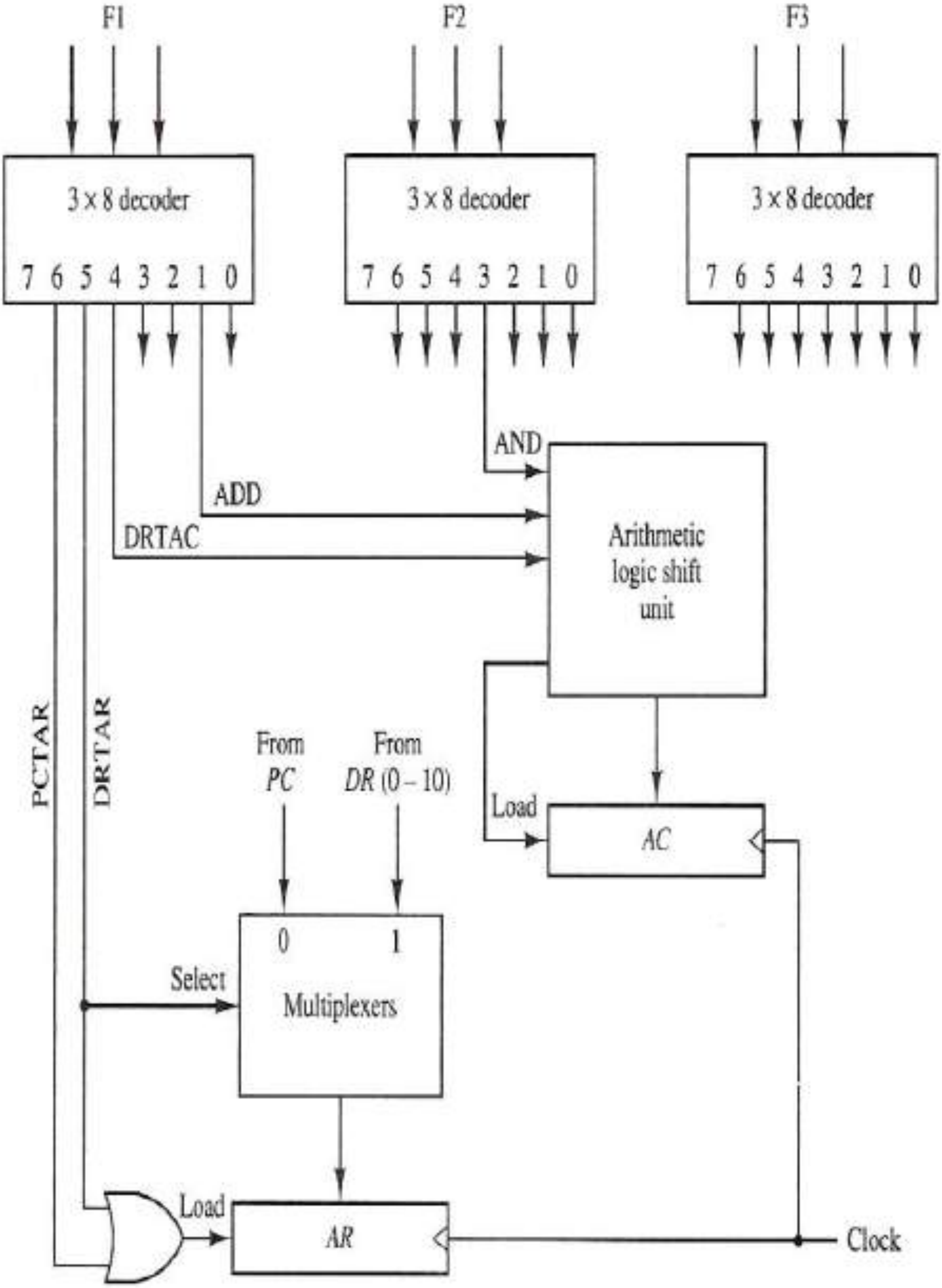


Figure: Decoding of micro-operation fields .Micro-program Sequencer

→Basic components of a micro-programmed control unit are control memory and the circuits that select the next address. This address selection part is called a micro-program sequencer. The purpose of micro-program sequencer is to load CAR so that microinstruction may be read and executed. Commercial sequencers include within the unit an internal resister stack to store addresses during micro-program looping and subroutine calls.

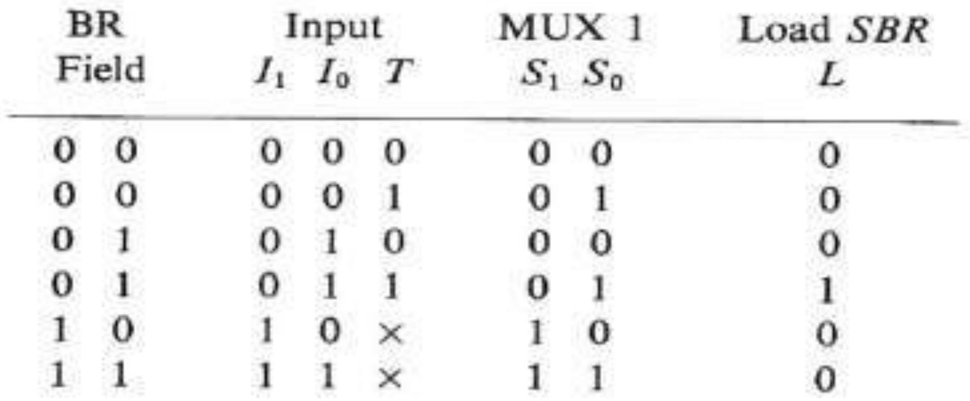
Design of input logic:

→The input logic circuit in **Fig. 7-8** has three inputs $I_{0,}$ $I_{1,}$ and T, and three outputs, $S_{0,}$ $S_{1,}$ and L. Variables $S_{0,}$ and S_1 select one of the source addresses for CAR.Variable L enables the load input in SBR. The binary values of the two selection variables determine the path in the multiplexers.

- MUX1 selects an address from one of four sources of and routes it into CAR.
- MUX2 tests the value of selected status bit and result is applied to input logic circuit.
- Output of CAR provides address for the control memory

→E.g. when S1S0=10, MUX input number 2 is selected and establishes a transfer path from SBR to CAR.

→Internal structure of a typical micro-program sequencer for a control memory is shown in the **Figure.** It consists of input logic circuit having following truth table.



BR Field		Input			MUX 1		Load *SBR*
		I_1	I_0	T	S_1	S_0	L
0	0	0	0	0	0	0	0
0	0	0	0	1	0	1	0
0	1	0	1	0	0	0	0
0	1	0	1	1	0	1	1
1	0	1	0	×	1	0	0
1	1	1	1	×	1	1	0

Fig: Input Logic Truth for Microprogram Sequencer

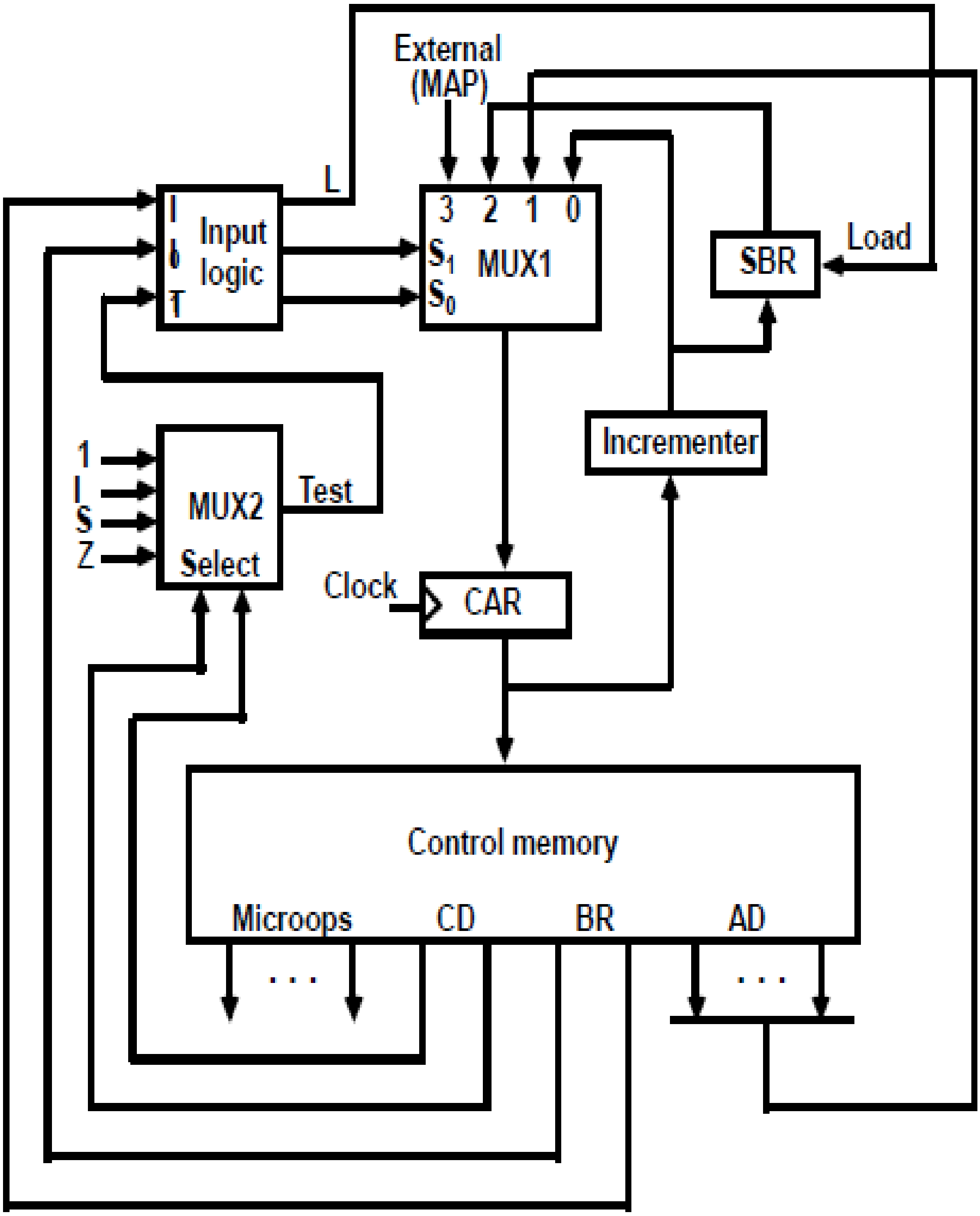


Figure: Micro-program sequencer for a control memory

→The truth table can be used to obtain the simplified Boolean functions for the input logic circuit:

$$S_0 = I_0$$
$$S_1 = I_0 I_1 + IO'T$$
$$L = I_0' I_1 T$$

COMPUTER ORGANIZATION CHAPTER-3

IMPORTANT QUESTIONS UNIT-3

[1] Explain various instruction formats used in general purpose computers?

[2] What are the different types of interrupts? Explain.

[3] Explain various data transfer instructions?

[4] Explain about RISC?

[5] Explain in detail the influence of number of number of addresses on computer programs and evaluate the arithmetic statements for different address instructions?

[6] Explain about stack organization?

[7] How many times does the control unit refer to memory when it fetches and executes an indirect addressing mode instruction if the instruction is (a) a computational type requiring an operand from memory; (b) a branch type?

[8] Explain different types of addressing?

[9] Discuss the implementation of a typical 64 word STACK in memory and give its PUSH and POP operations?

[10] Compare and contrast RISC and CISC computers?

[11] The memory unit of a computer has 256K words of 32 bits each. The computer has an instruction format with four fields: an operation code field, a mode field to specify one of seven addressing modes, a register address field to specify one of 60 processor registers, and a memory address. Specify the instruction format and the number of bits in each field if the in instruction is in one memory word.

[12] Explain about the micro-program sequencer for a control unit?

[13] Draw the block diagram for design of control unit and explain its functionality?

[14] Hardwired control unit is faster than micro-programmed control unit. Justify the statement.

[15] Explain control memory and symbolic micro-program?

[16] Explain the Fetch routine?

[17] Describe how micro-instructions are arranged in control memory and how they are interpreted?

(a) Draw the block diagram of micro-program sequencer for a control memory and explain its operation?

(b) Distinguish between micro-program and hardwired control?

(c) What is the difference between a microprocessor and a micro-program? Is it possible to design a microprocessor without a micro-program?

COMPUTER ORGANIZATION CHAPTER-3

[18] (a) Formulate a mapping procedure that provides eight consecutive microinstructions for each routine. The operation code has six bits and the control memory has 2048 words.

(b) A computer has 16 registers, an ALU (arithmetic logic unit) with 32 operations, and a shifter with eight operations, all connected to a common bus system.

(i)Formulate to a control word for a micro-operation.

(ii)Specify the number of bits in each field of the control word and give a general encoding scheme.

[19] Explain the difference between hardwired and micro-programmed control?

[20] Write in brief micro-programmed control and hardwired control.

[21] What is control word?

[22] Discus about pipeline register?

[23] Control memory unit?

[24] Name the different micro-instruction formats.

[25] What is the purpose of micro-program sequencer?

[26] What are the different stack operations? Explain.

[27] Explain in brief about Conditional Branch Instructions and Indirect Addressing Mode.

[28] What do you understand by addressing mode?

[29] Perform the logic AND, OR, and XOR with the two binary strings 1001 1100 and 10101010.

[30] Convert the following arithmetic expressions from reverse Polish notation to infix notation. A B C*/ D-E F / +

[31] State the relative advantages and disadvantages of direct and indirect addressing mode? Convert (A+ B)*[C* (D + E) +F] into reverse Polish notation.

Memory Organization: Memory Hierarchy, Main Memory, Auxiliary Memory, Associative Memory, Cache Memory, Virtual Memory.
Input-Output Organization: Peripheral Devices, Input-Output Interface, Asynchronous data transfer, Modes of Transfer, Priority Interrupts, Direct Memory Access

1. Memory Hierarchy:

→The memory unit that communicates directly with the CPU is called ***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 memory hierarchy system consists of all storage devices employed in a computer system from the slow but high-capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory accessible to the high speed processing logic.

→CPU Register - also known as Internal Processor Memory. The data or instructions which have to be executed are kept in these registers.

→The **Cache Memory** is employed in computer system 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 result that processing speed is limited primarily by the speed of main memory. The cache is used for storing segments of programs currently being executed in the CPU and temporary data frequently needed in the present calculations.

→By making program and data available at a rapid rate, it is possible to increase the performance rate of the computer.

→The typical access time ratio between cache and main memory is about 1to 7~10. Auxiliary memory access time is usually 1000 times that of main memory.

→Multiprogramming– Many operating systems are designed to enable the CPU to process a number of independent programs concurrently – known as multiprogramming. Sometimes a program is too long to be accommodated in total space available in main memory. A program with its data normally resides in auxiliary memory. When a program or a segment of program is to be executed, it is transferred to main memory to be executed by the CPU. The part of the computer system that supervises the flow of information between auxiliary and main memory is called the **memory management system**.

→**Figure 1** illustrates the components in a typical memory hierarchy.

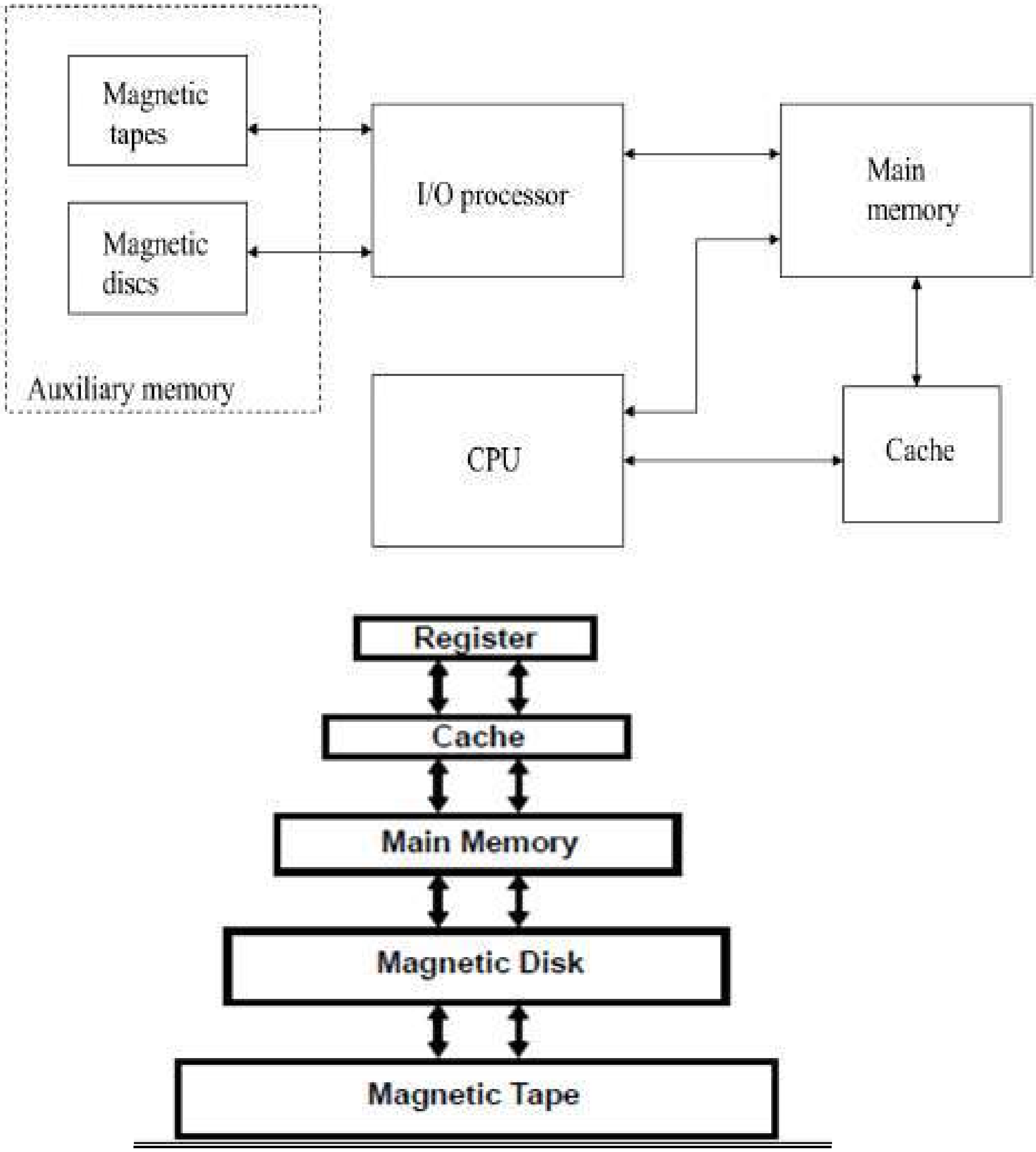


Figure 1 Memory hierarchy in computer system

2. Main Memory:

→Main memory is central storage unit in computersystem. It is a relatively large and fast memory used to store programs and data during the computer operation. The principal technology used for main memory is based on semiconductor integrated circuits.

Random access memory (RAM):

→The integrated circuit chips are available in two possible operating models: Static and Dynamic.

→The**Static RAM (SRAM):**consists essentially of internal flip-flops that store the binary information. The dynamic RAM stores the binary information in the form of electronic charges that are applied to capacitors. The stored charge on the capacitors tends to discharge with time and the capacitors must be periodically recharged by refreshing the dynamic memory. The static RAM is easier to use and has shorter read and write cycles and used in cache.

→The**Dynamic RAMs (DRAMs):**are used for implementing the main memory.DRAMs stores data as form of electronic charges in small capacitors. Capacitors are provided by CMOS transistors. Needs refreshing periodically as charges on small capacitor discharge soon (need electronic control unit for that).

→DRAM compared to SRAM offer reduced power consumption and larger capacity. But SRAM are faster.

→ROM is different type of main memories. Used to store programs and data that does not change at all (programs, tables, etc.)

→The ROM portion of main memory is needed for storing an initial program called a **bootstrap loader** (start loading operating systems).

→ ROMs are used to startup any computer.

→ROM and RAM chips are available in different sizes. And usually we have to combine many chips to increase size.

RAM and ROM Chips:

1. RAM Chip:

→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 block diagram of a RAM chip is shown in **Figure 12-2.**

→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 two chips select (CS) control inputs are for enabling the chip only when it is selected by the microprocessor.

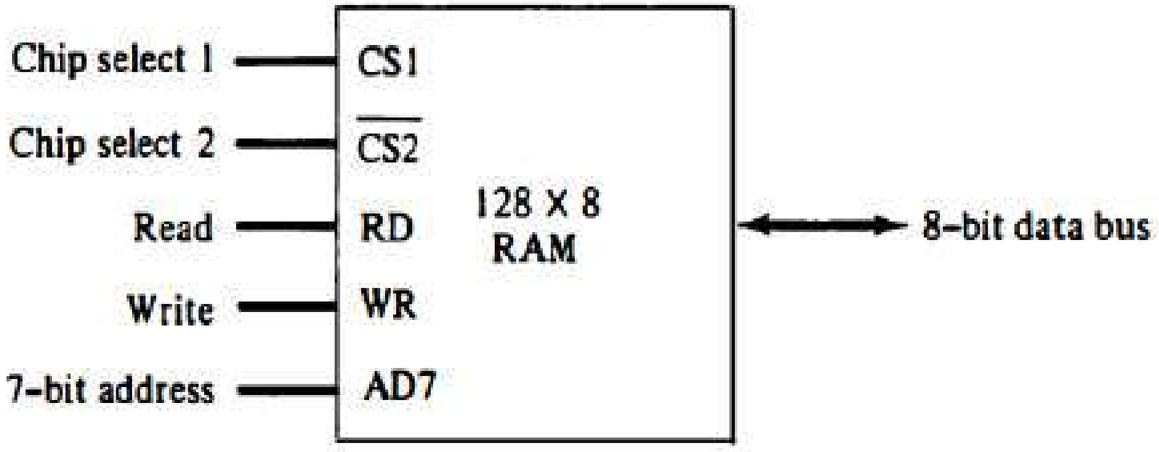
→The function table listed in Fig. 12-2(b) specifies the operation of the RAM chip.

→When **CS1=1 and** $\overline{CS2}$ **= 0**, the memory can be placed in a write or read mode.

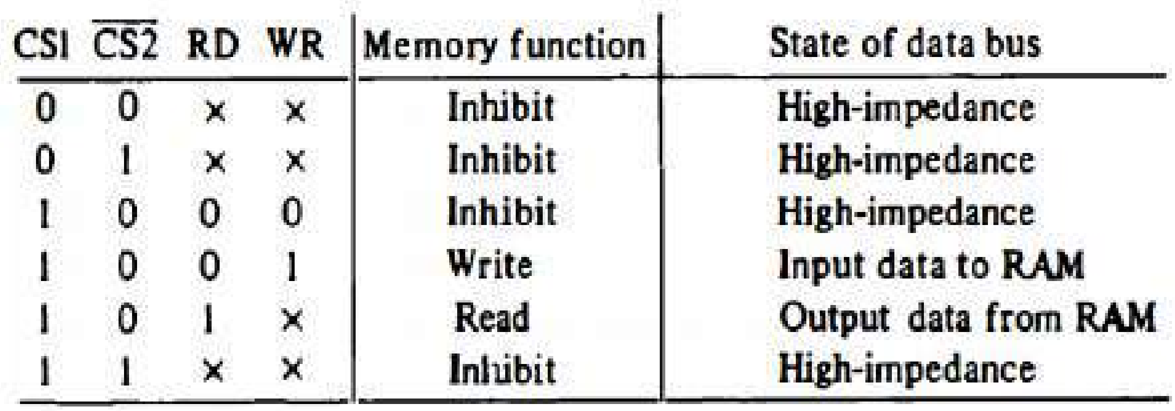
→When the WR input is enabled, the memory stores a byte from the data bus into a location specified by the address input lines.

→ When the RD input is enabled, the content of the selected byte is placed into the data bus.

Figure 12-2 Typical RAM chip.



(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

2. ROM Chip:

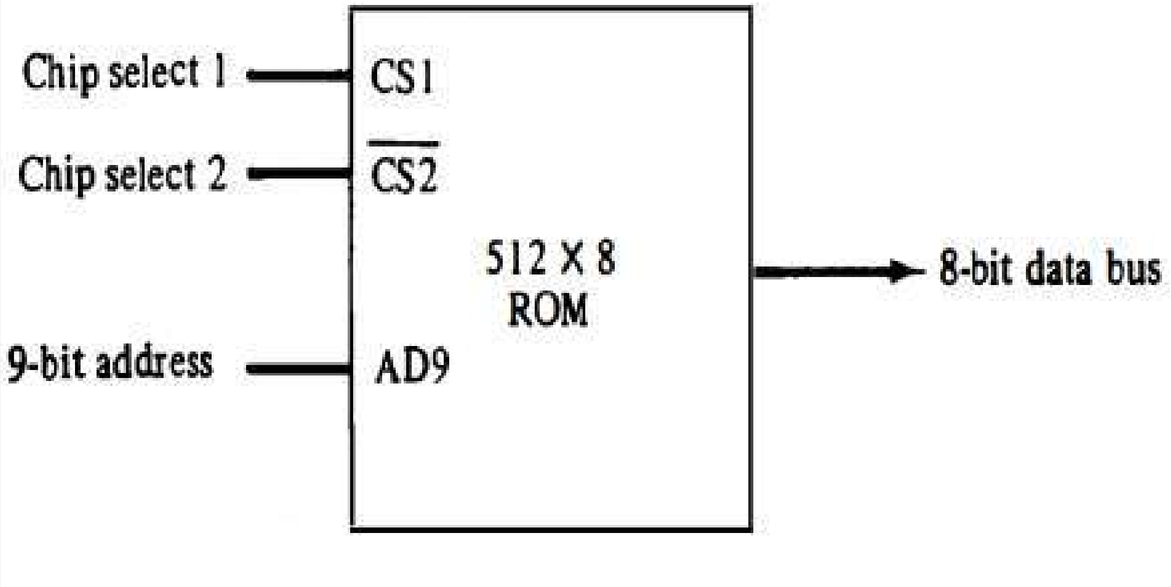


Figure 12-3 Typical ROM chip.

→Since the ROM can only read, the data bus can only be in an output mode.
→The block diagram of a ROM chip is shown in Figure 12-3.
→No need of READ and WRITE control. Same sized RAM and ROM chip, it is possible to have more bits of ROM than of RAM, because the internal binary cells in ROM occupy less space than in RAM.

→The nine address lines in the ROM chip specify any one of the 512 bytes stored in it. The chip select inputs must be CS1=1 and $\overline{CS2}$ =0 for the unit to operate. Otherwise, data bus is in a high-impedance state.

Memory Address Map:

→*Memory Address Map* is a pictorial representation of assigned address space for each chip in the system.

→Assume computer system with 512 bytes of RAM and 512 bytes of ROM. The memory address map is shown in **TABLE 12-1**.

→The RAM chips have 128 bytes and need seven address lines. The ROM chip has 512 bytes and needs nine address lines. The X's are always assigned to low-order bus lines: lines 1 through 7 for the RAM and lines 1 through 9 for the ROM.
→The table clearly shows that the nine low-order bus lines constitute a memory space for RAM equal to 2^9 = 512 bytes.
→When line 10 is 0, the CPU selects a RAM, and when this line is equal to 1, it selects the ROM.

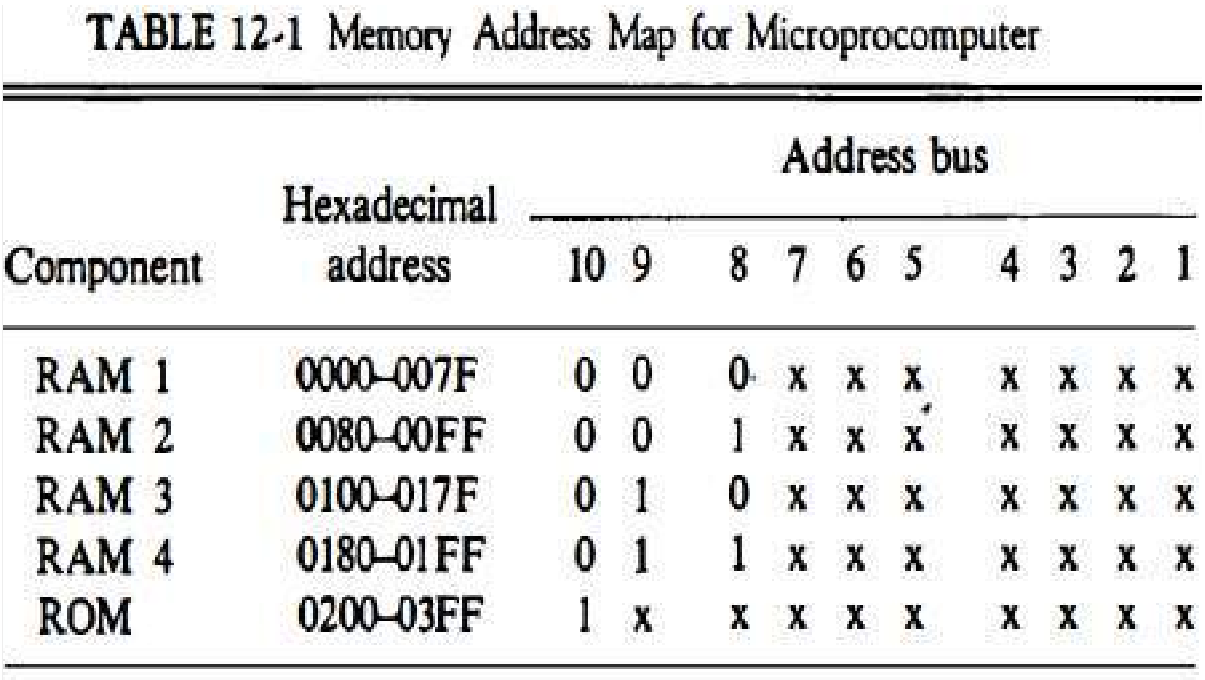


TABLE 12-1 Memory Address Map for Microprocomputer

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

Memory connection to CPU:

→RAM and ROM chips are connected to a CPU through the data and address buses.

→The connection of memory chips to the CPU is shown in **Figure 12-4**. This configuration gives a memory capacity of 512 bytes of RAM and 512 bytes of ROM. It implements the memory map of **Table 12-1**.

→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.

→When the address lines 8 and 9 are equal to 00, the first 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.

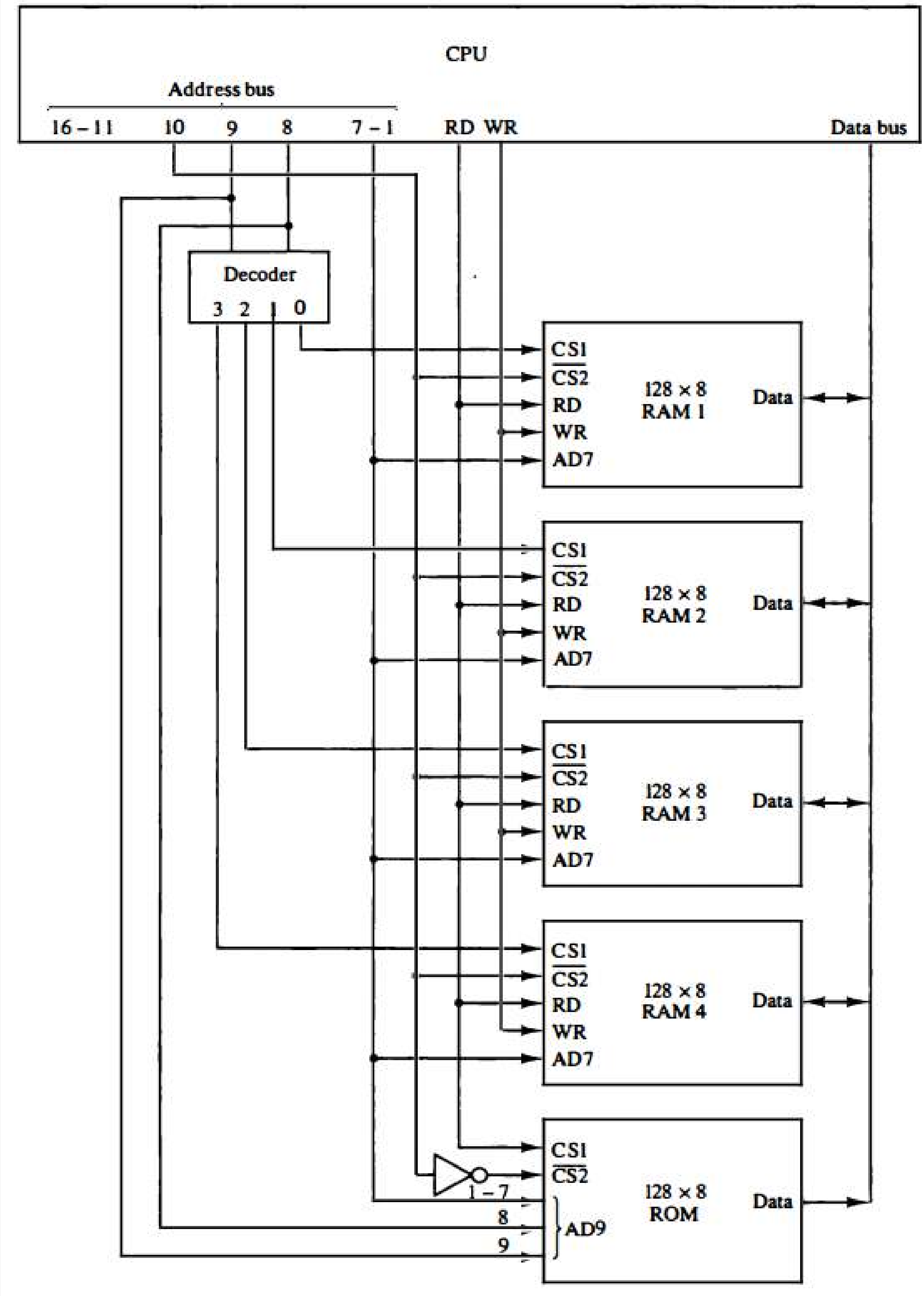


Figure 12-4 Memory connection to the CPU.

3. Auxiliary Memory:

→The most common auxiliary memory devices used in computer system are magnetic disk and tapes. Other components used, but not as frequently, are magnetic drums, magnetic bubble memory, and optical disks. 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 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.

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 for access purposes.

→Bits are stored in magnetized surface in spots along concentric circles called t**racks**.

→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 subdivision of one disk surface into tracks and sectors is shown in **Figure. 12-5**.

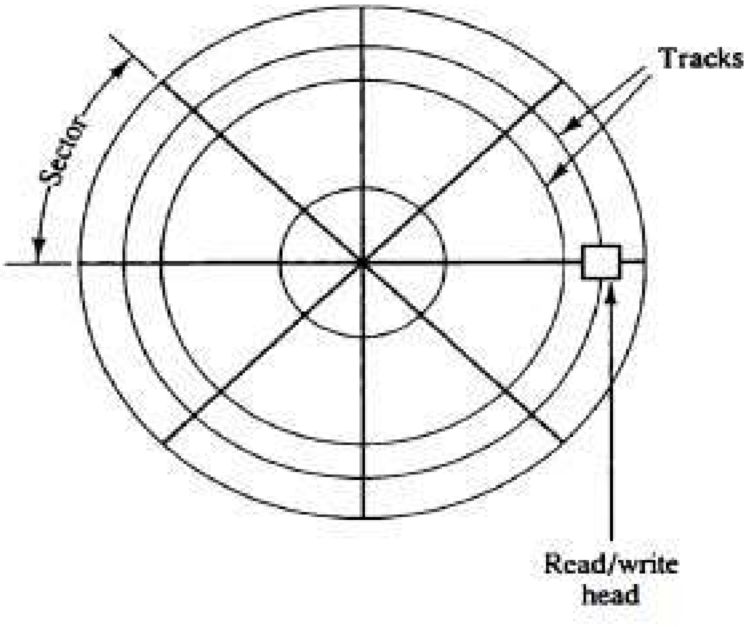
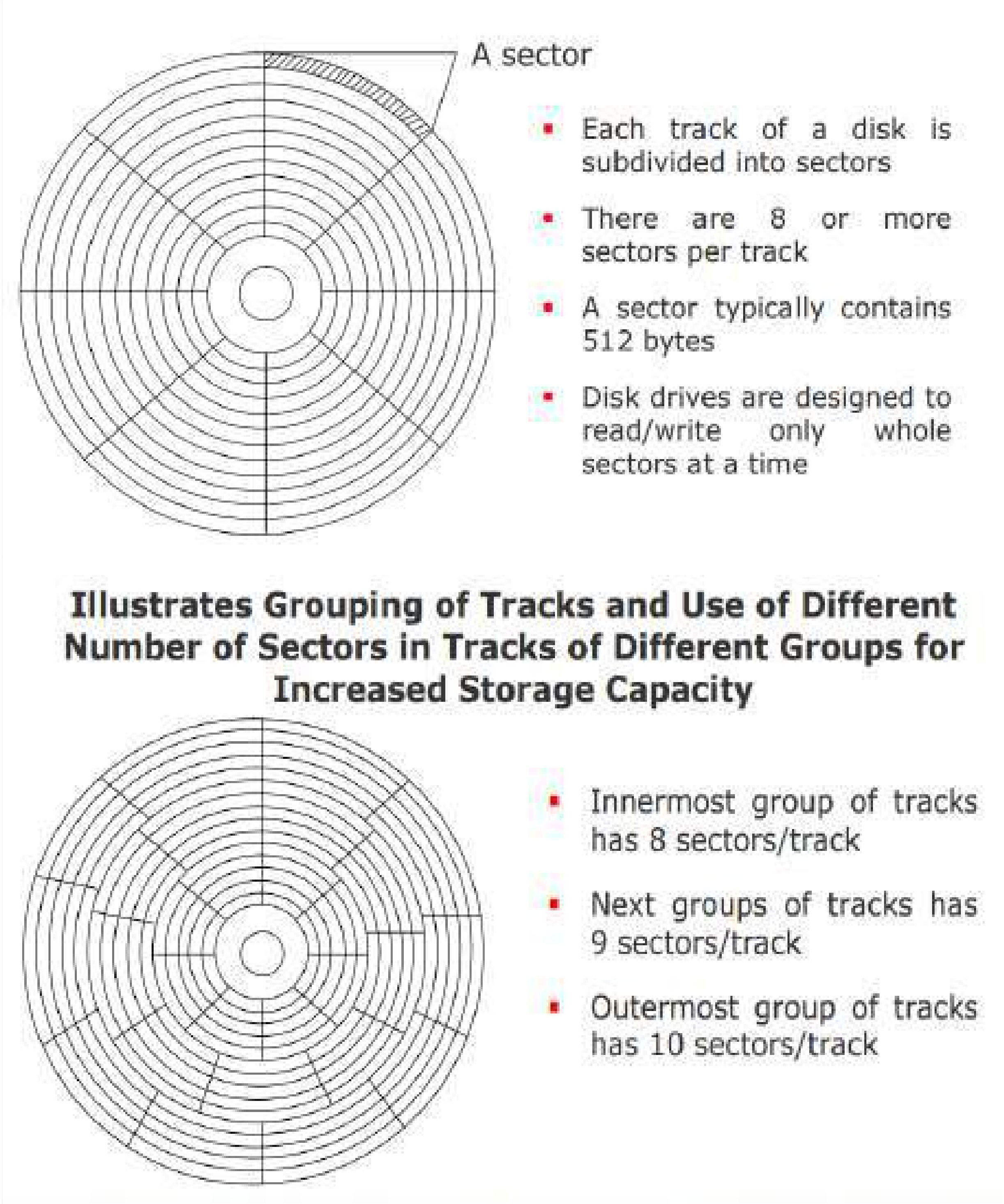


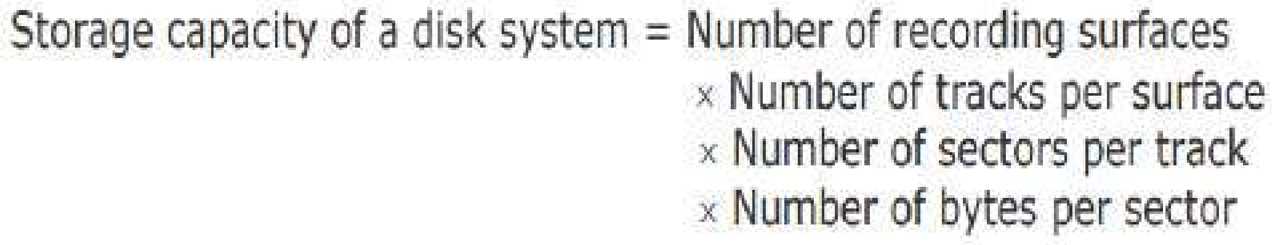
Figure 12-5 Magnetic disk.

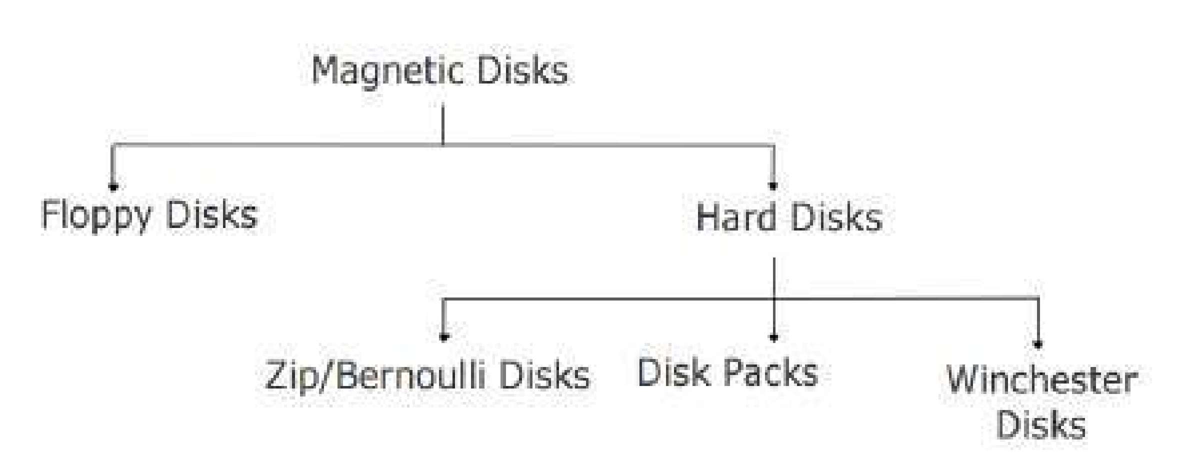


- Each track of a disk is subdivided into sectors
- There are 8 or more sectors per track
- A sector typically contains 512 bytes
- Disk drives are designed to read/write only whole sectors at a time

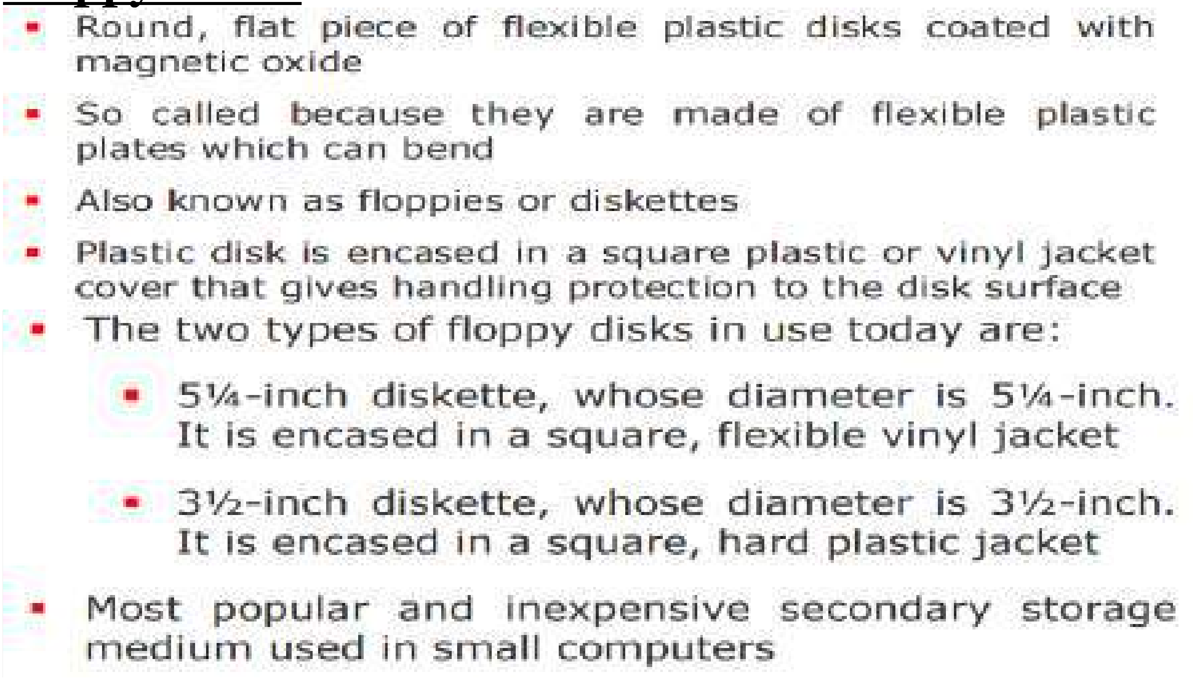
Illustrates Grouping of Tracks and Use of Different Number of Sectors in Tracks of Different Groups for Increased Storage Capacity

- Innermost group of tracks has 8 sectors/track
- Next groups of tracks has 9 sectors/track
- Outermost group of tracks has 10 sectors/track


Storage capacity of a disk system = Number of recording surfaces
× Number of tracks per surface
× Number of sectors per track
× Number of bytes per sector

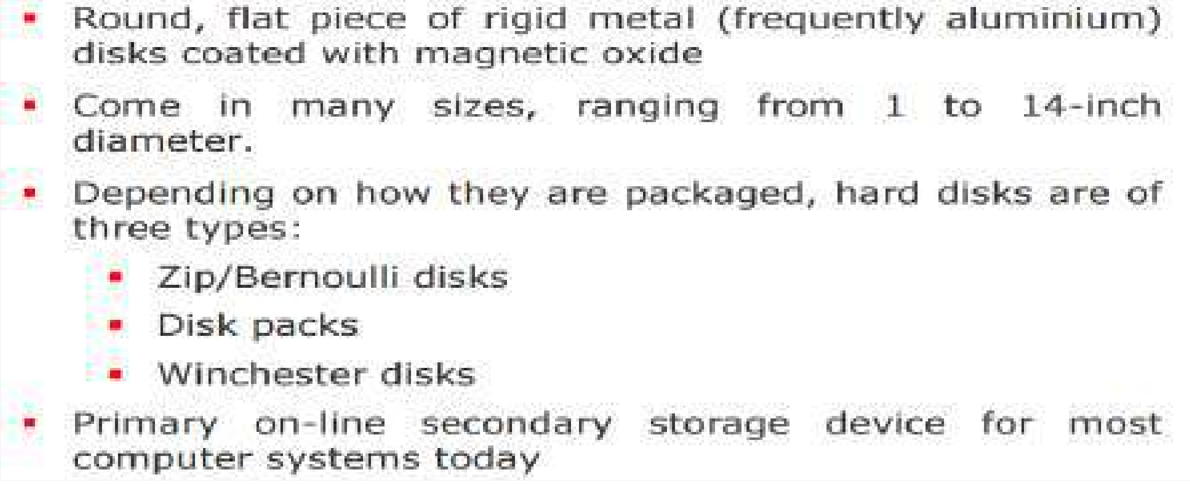


Floppy Disks:



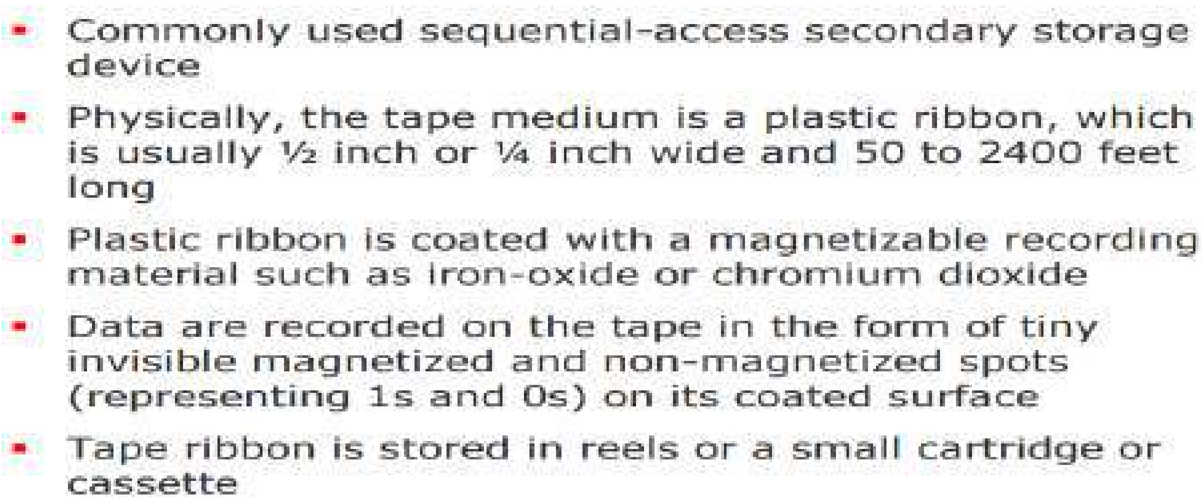
- Round, flat piece of flexible plastic disks coated with magnetic oxide
- So called because they are made of flexible plastic plates which can bend
- Also known as floppies or diskettes
- Plastic disk is encased in a square plastic or vinyl jacket cover that gives handling protection to the disk surface
- The two types of floppy disks in use today are:
  - 5¼-inch diskette, whose diameter is 5¼-inch. It is encased in a square, flexible vinyl jacket
  - 3½-inch diskette, whose diameter is 3½-inch. It is encased in a square, hard plastic jacket
- Most popular and inexpensive secondary storage medium used in small computers

Hard Disks:



- Round, flat piece of rigid metal (frequently aluminium) disks coated with magnetic oxide
- Come in many sizes, ranging from 1 to 14-inch diameter.
- Depending on how they are packaged, hard disks are of three types:
  - Zip/Bernoulli disks
  - Disk packs
  - Winchester disks
- Primary on-line secondary storage device for most computer systems today

1. Magnetic Tapes:



- Commonly used sequential-access secondary storage device
- Physically, the tape medium is a plastic ribbon, which is usually ½ inch or ¼ inch wide and 50 to 2400 feet long
- Plastic ribbon is coated with a magnetizable recording material such as iron-oxide or chromium dioxide
- Data are recorded on the tape in the form of tiny invisible magnetized and non-magnetized spots (representing 1s and 0s) on its coated surface
- Tape ribbon is stored in reels or a small cartridge or cassette

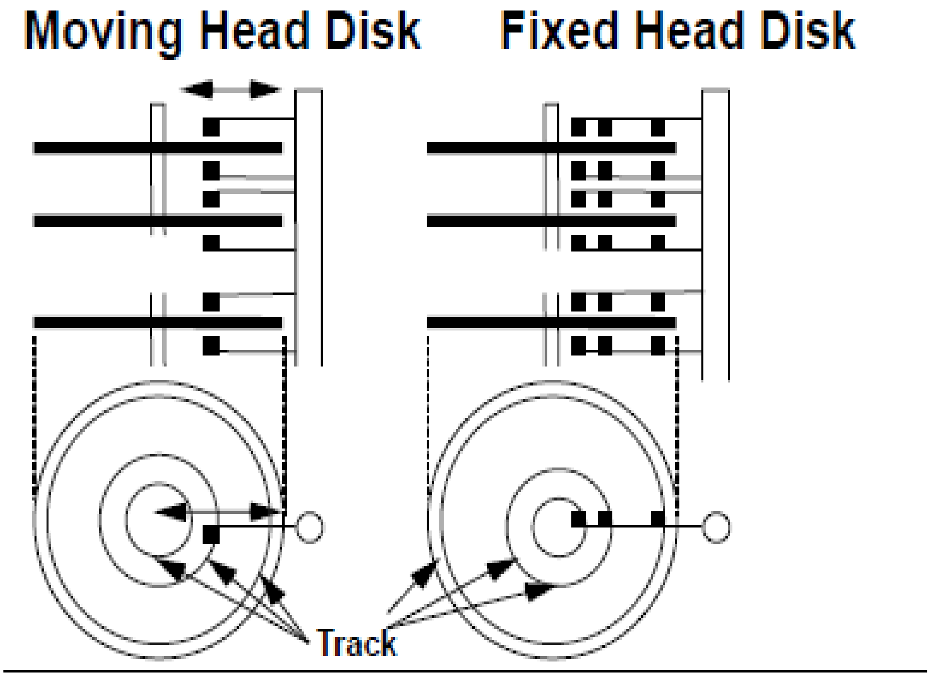


Figure. Magnetic Tape

→Bits are recorded as magnetic spots on the tape along several parallel tracks(7 to 9 tracks to form character with parity).
→Read/write heads are mounted one on 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.
→Data are recorded in records (number of characters) followed by gaps between record for synchronization.
→Each record on tape has an identification bit pattern at the beginning and end.
→Records are identified by reading ID bit patterns.

4. Associative Memory:

→The time required to find an object stored in memory can be reduced considerably if the objects are based on their contents, not on their locations. A memory unit accessed by the content is called an **associative memory.**

→A memory unit is 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.

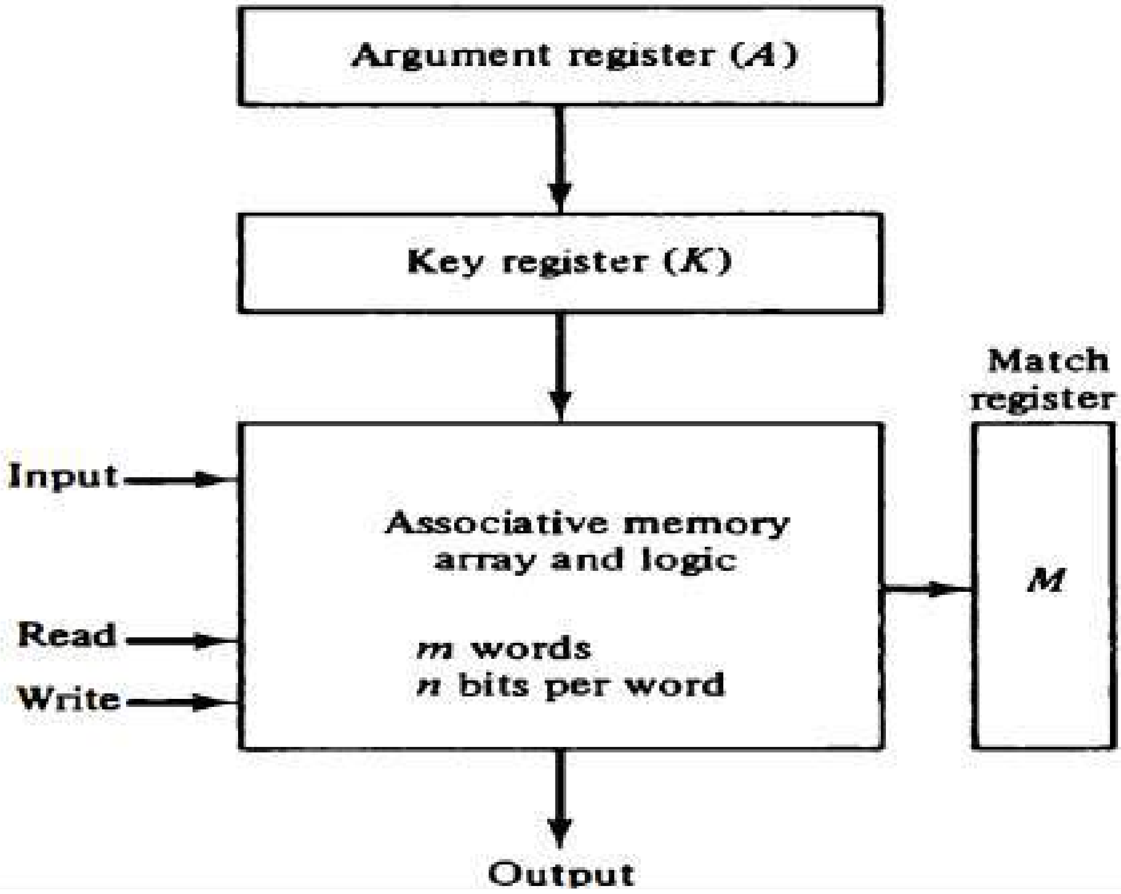
• The memory is capable of finding an empty unused location to store the word.

• When a word is to be read from an associative memory, the content of the word, or part of the word, is specified.

• An associative memory is more expensive than 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, figure bellow show Block diagram of associative memory.

Hardware Organization:

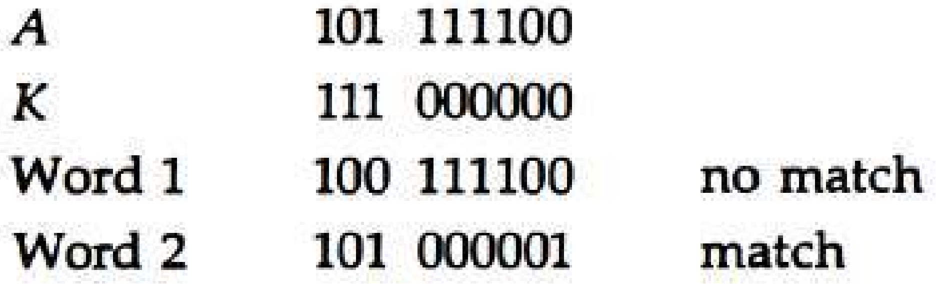
Figure 12-6 Block diagram of associative memory.



→The block diagram of an associative memory is shown in **Figure. 12-6.** It consists of a memory array and logic for m 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. 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.

→As 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.



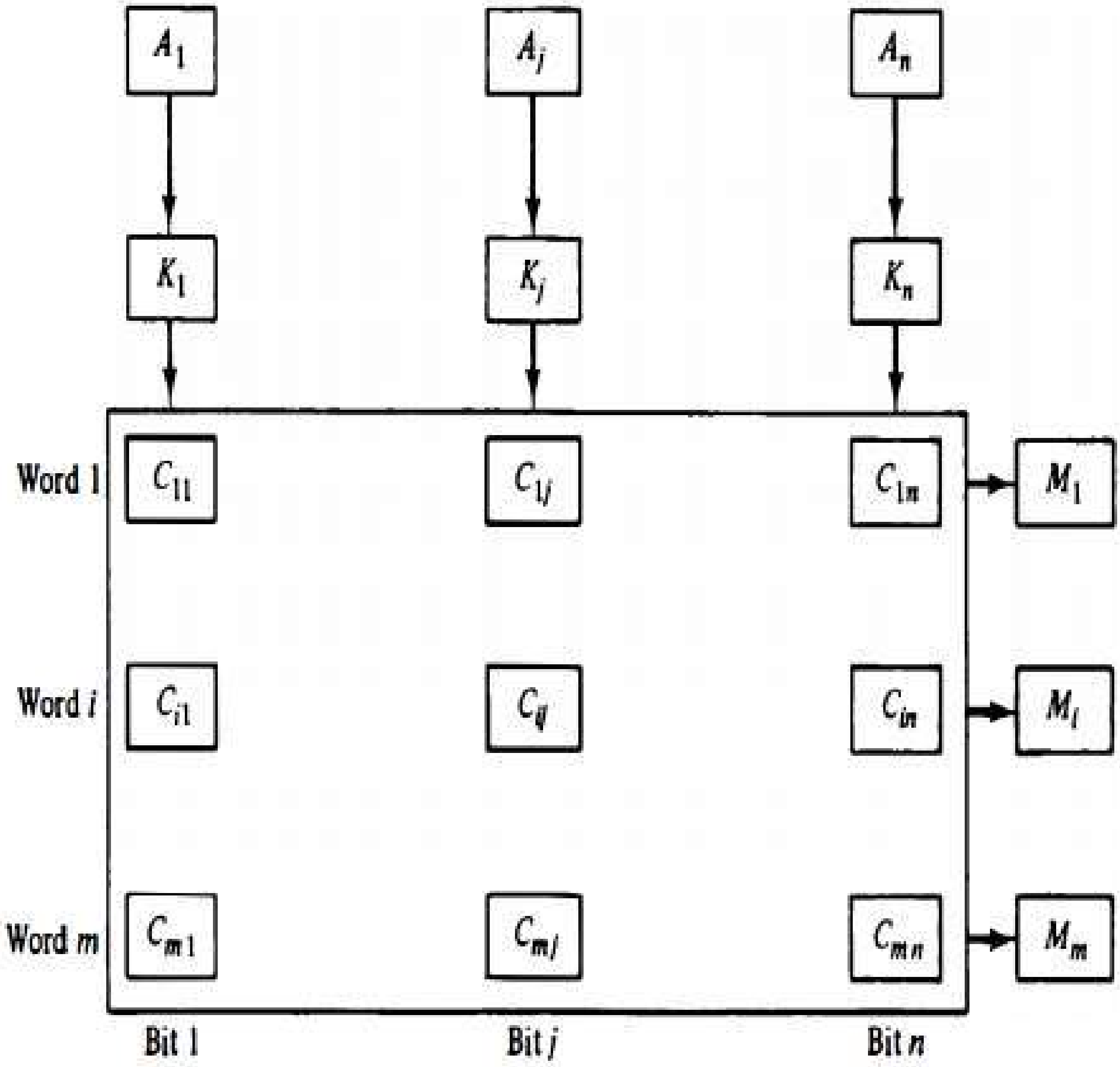
A	101 111100	
K	111 000000	
Word 1	100 111100	no match
Word 2	101 000001	match

Word 2 matches the unmasked argument field because the three left most bits of the argument and the words are equal.

→The relation between the memory array and external registers in an associative memory is shown in **Fig. 12-7.**

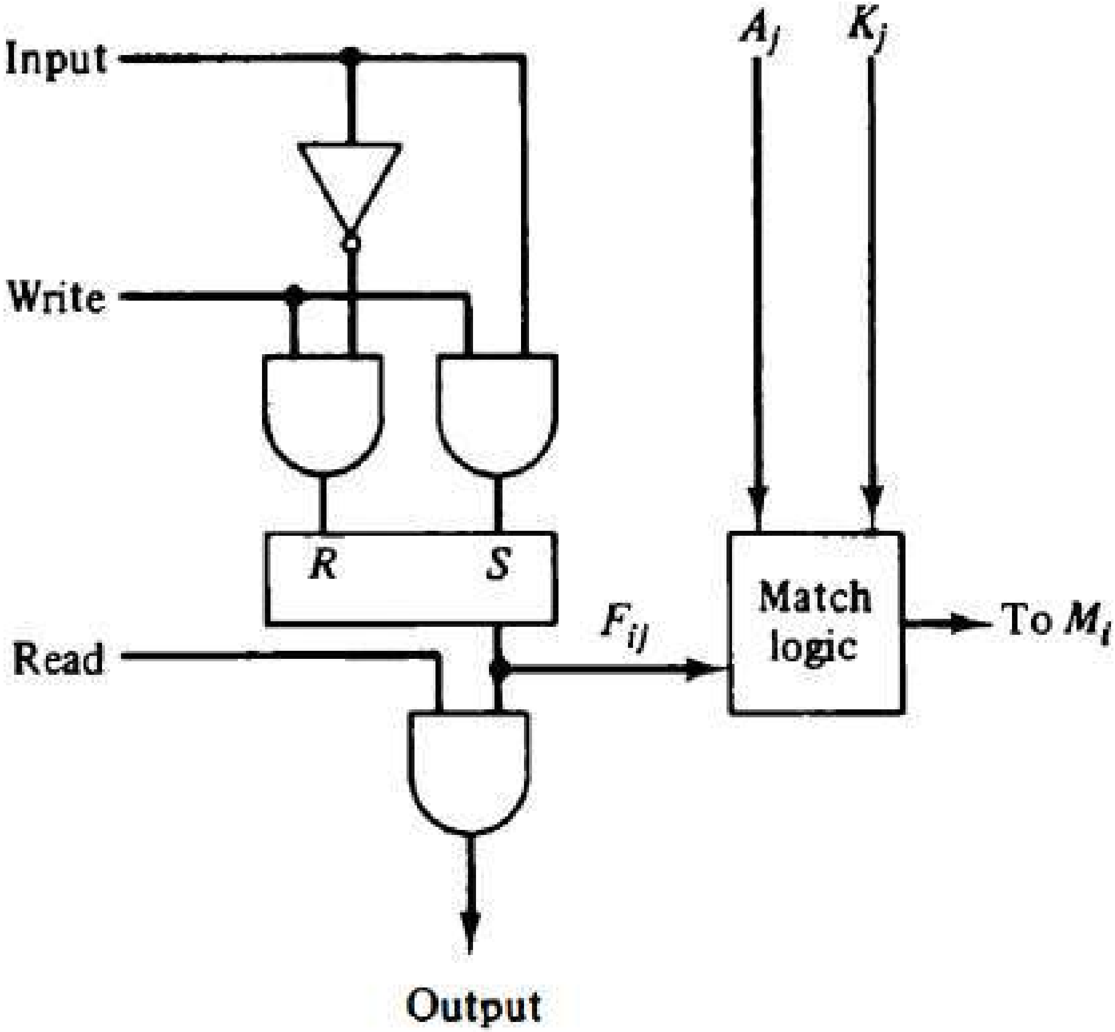
→The cell C_{ij} is the cell for bit j in word i. A bit A_j in the argument register is compared with all the bits in column j of the array provided that K_j =1.This is done for all columns j=1, 2… n. If a match occurs between all the unmasked bits of the argument and the bits in word i, the corresponding bit M_i in the match register is set to 1. If one or more unmasked bits of the argument and the word don't match, M_i is cleared to 0.

Figure 12-7 Associative memory of m word, n cells per word.



→The internal organization of a typical cell C_{ij} is shown in **Fig. 12-8**. It consists of a flip-flop storage element F_{ij} and the circuit 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 ce4ll with the corresponding unmasked bit of the argument and provides an output for the decision logic that sets the bit in M_i.

Figure 12-8 One cell of associative memory.



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 1or 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_i = 1 if the pair of bits in position j are equal; otherwise, x_i = 0. For a word i to be equal to the argument in A we must have all x_i 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

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

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

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:

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

When $K_j = 1$, we have $K_j' = 0$ and $x_j + 0 = x_j$. When $K_j = 0$, then $K_j' = 1$ and x; + 1 = 1. A term ($x_j + K_j'$) will be in the 1 state if its pair of bits is 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:

$$M_i = (x_1 + K_1')(x_2 + K_2')(x_3 + K_3') \cdots (x_n + K_n')$$

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 π 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 matching one word is shown in **Figure 12-9**. 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 match occurs and 0 if no match occurs. Note that if the key register contains all 0's, output M, will be a 1 irrespective of the value of A or the word. This occurrence must be avoided during normal operation.

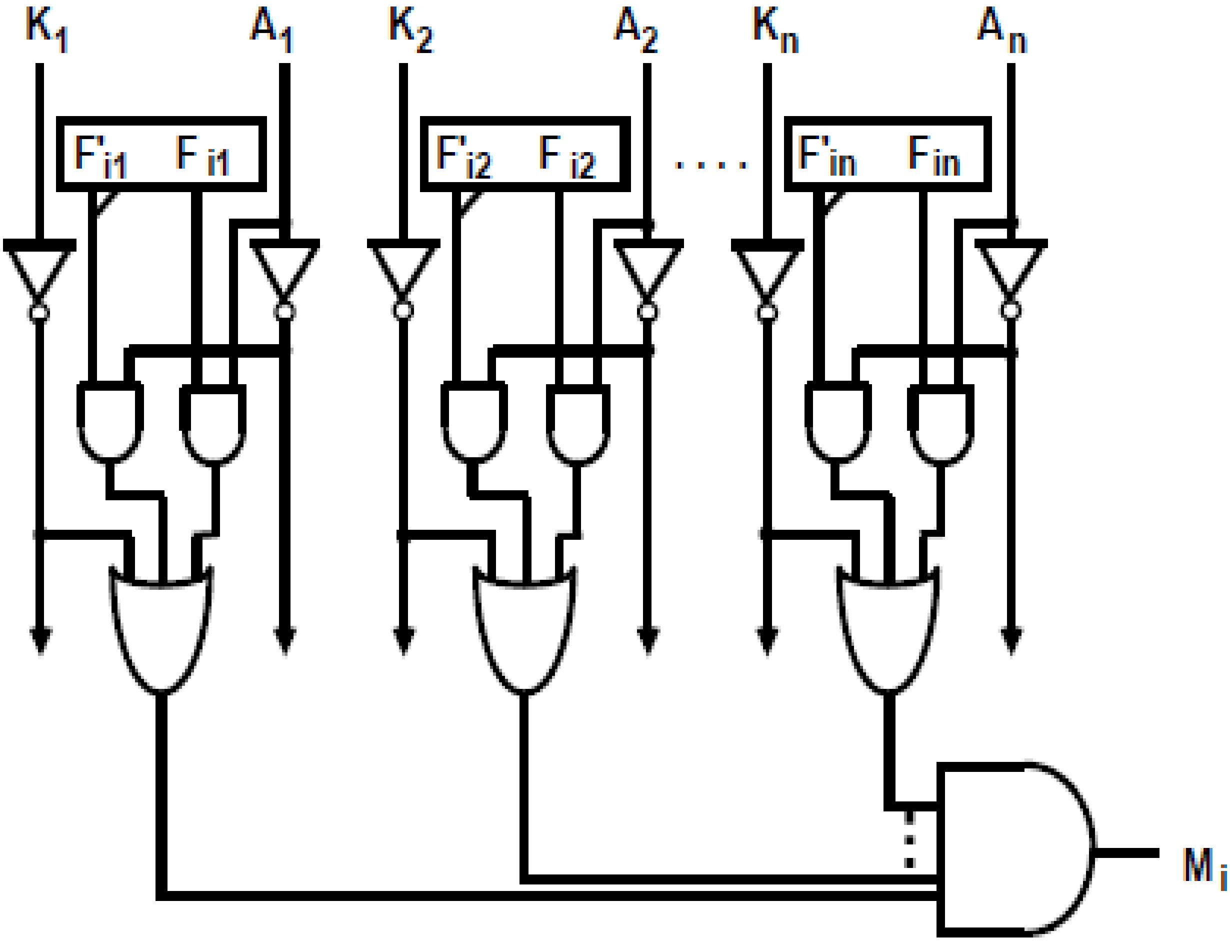


Figure 12-9Match logic for one word of associative memory.

Read Operation:

→In the read operation all the matched words are read in sequence by applying read signal to each word line whose corresponding M_i bit is logic 1.In application where no two identical items are stored in the memory, only one word may match the unmasked argument field.In such case we can use M_i output directly as a read signal for the corresponding word.The contents of the matched word will be presented automatically at the output lines and no special signal is needed.

Write Operation:

→An associative memory must have a write capability for storing the information to be searched. Writing in an associate memory can take different forms, depending upon the application. 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 take the memory device a random access memory for writing and a content addressable memory for reading.

→The advantage here is 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 $m=2^d$.

5. Cache Memory:

→**Locality of reference**is defined as an analysis has shown that references to memory at given interval of time is confined within few localized areas in memory.

→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 of 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 12-10**.

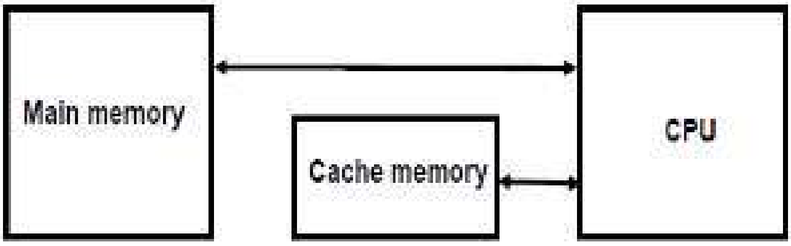


Figure 12-10 Example of cache memory.

Hit Ratio: The performance of cache memory is frequently measured in terms of quantity called hit ratio. The ratio of the number of hits divided by the total CPU references (hits + misses) to memory.

- **hit** : the CPU finds the word in the cache (0.9)
- **miss** : the word is not found in cache (CPU must read main memory)

→Cache memory access time = 100 ns, main memory access time = 1000 ns and
hit ratio = 0.9

→The basic characteristic of cache memory is its fast access time. The transformation of data from main memory to cache memory is referred to as *mapping* process.

→Three types of mapping techniques are:

- **Associative mapping**
- **Direct mapping**
- **Set associative mapping**

Associative mapping:

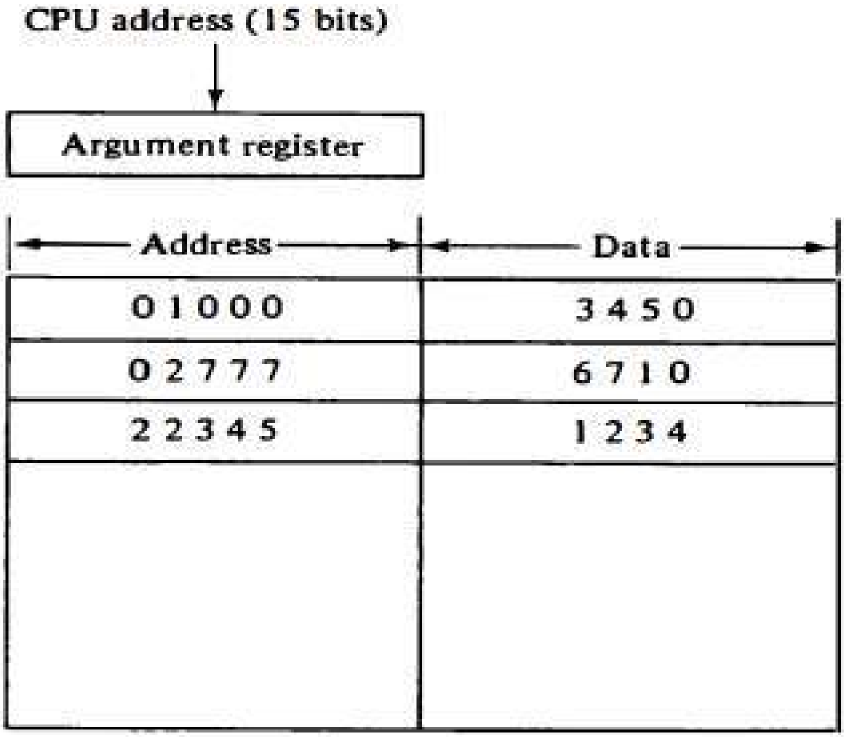
→The fastest and most flexible cache organization uses an associative memory. This organization is illustrated in **Figure 12-11**.

→The associative memory stores both the address and content (data) of the memory word. This permits any location in cache to store any 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.

Figure 12-11 Associative mapping cache (all numbers in octal).



Address	Data
01000	3450
02777	6710
22345	1234

→If the address is found, the corresponding 12-bit data is read and sent to CPU. If no match occurs, the main memory is accessed for the word. The address-data pair is then transferred to the associative cache memory.

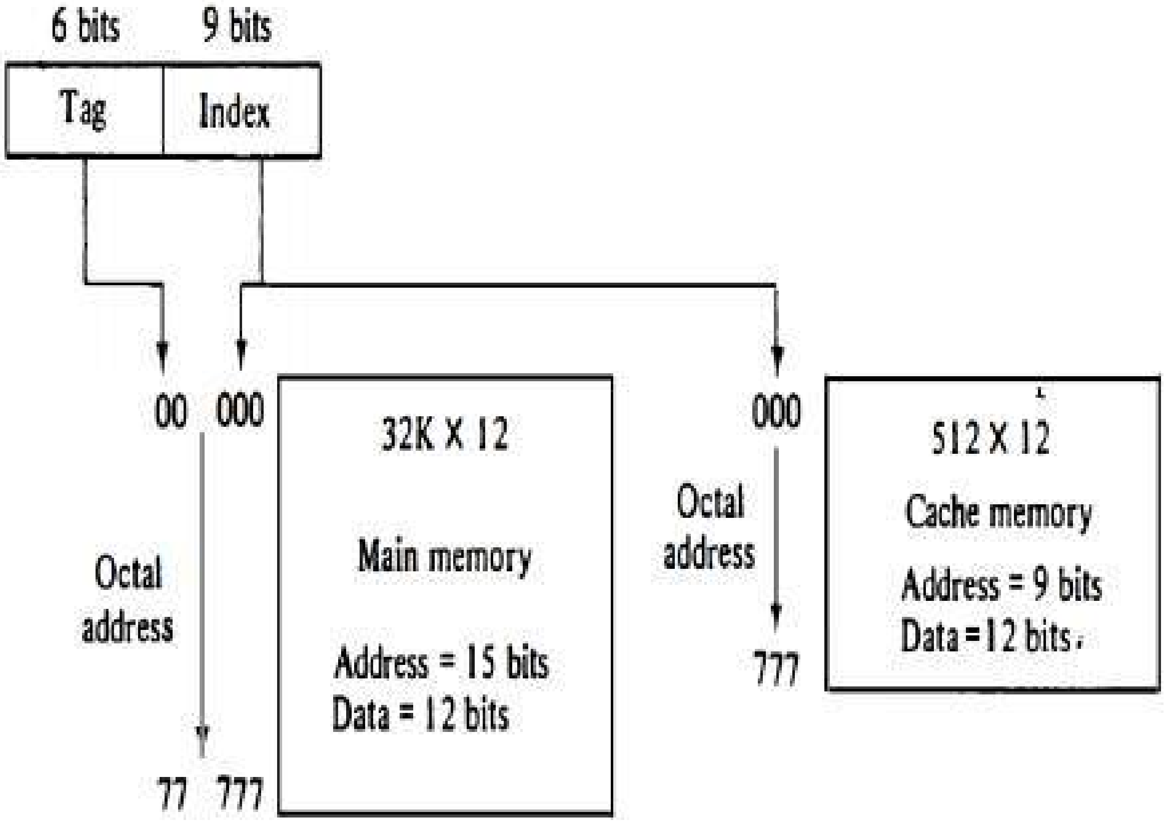
→If the cache is full, an address-data pair must be displaced to make room for a pair that is needed and not presently in the cache.

→This constitutes a first-in first-out (FIFO) replacement policy.

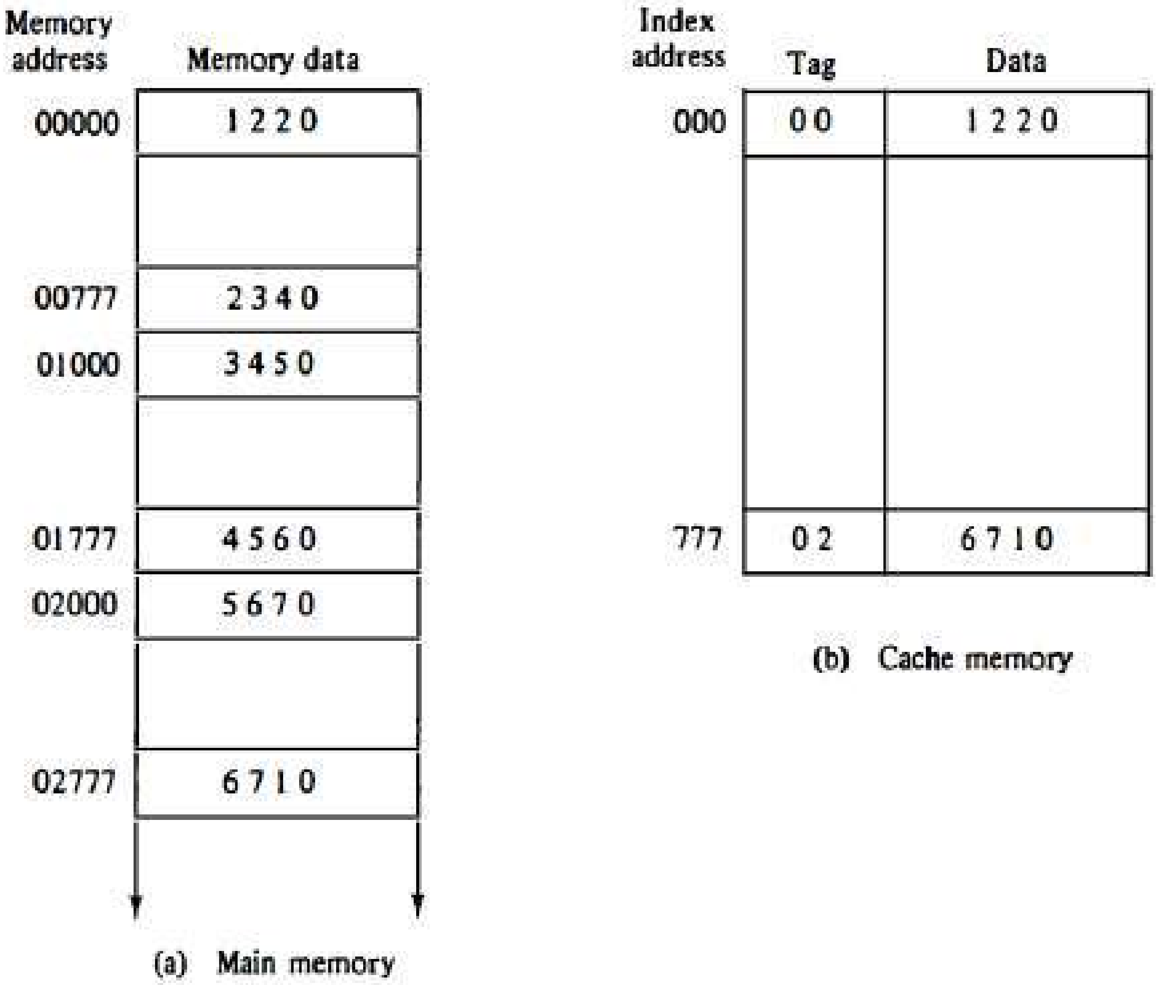
Direct mapping:

→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 **Figure. 12-12**.

Figure 12-12 Addressing relationships between main and cache memories.



→The CPU address (15 bits the number of address bits (n) required to access the 32Kx12 main memory as example) is divided into two fields. The least significant bits constitute the **index field** (nine bits the number of address bite (k) required to access the 512x12 cache memory as example) and the remaining six bits from the **tag field** (n-k). The figure shows that main memory needs an address that includes both the tag and the index bits.



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

(a) Main memory

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

(b) Cache memory

Figure 12-13 Direct mapping cache organization.

→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 12-13(b).**

→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.

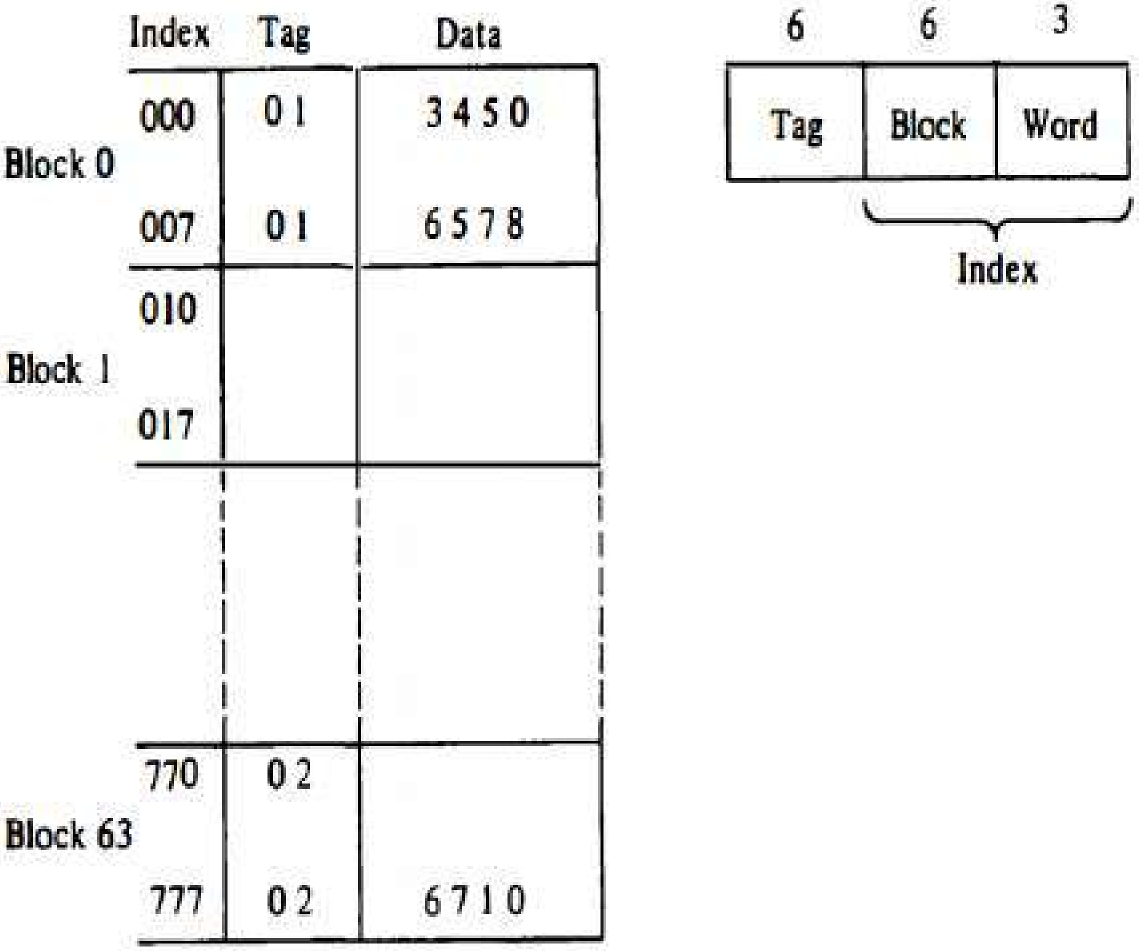
→When CPU generates a memory request, the index field is used for the address to access the cache. The tag field of the CPU address is compared with the tag in the word read from the cache.

→If the two tags match, there is a hit and the desired data word is in cache. If there is no match, there is a miss and the required word is read from main memory.It is then stored in the cache together with the new tag, replacing the previous value.

→The disadvantage of direct mapping is that the hit ratio can drop considerably if two or more words whose addresses have the same index but different tags are accessed repeatedly. However, this possibility is minimized by the fact that such words are relatively far apart in the address range (multiples of 512 locations in this example).

→Consider the numerical example shown in **Figure 12-13**. 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 used 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 direct-mapping example just described uses a block size of one word. The same organization but using a block size of B words is shown in **Figure 12-14**. The index field is now divided into two parts: the block field and the word field. In a 512-word cache there are 64 blocks of 8 words each, since 64 x 8 = 512.



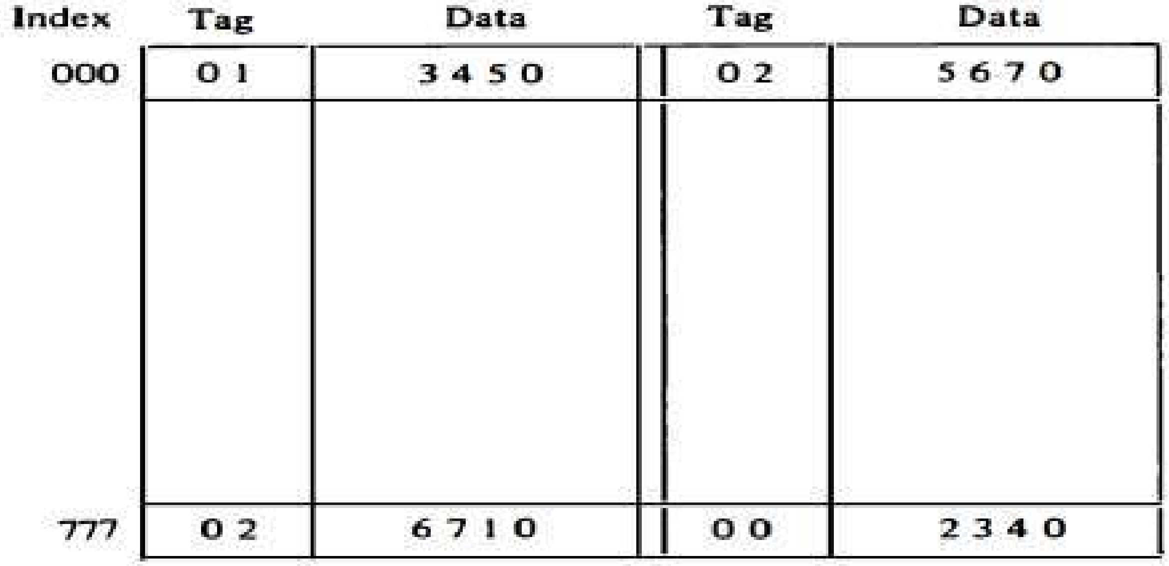
	Index	Tag	Data
Block 0	000	01	3450
	007	01	6578
Block 1	010		
	017		
Block 63	770	02	
	777	02	6710

Figure 12-14 Direct mapping cache with block size of 8 words.

Set-Associative mapping:

→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 **Figure 12-15**. Each indexaddress refers to two data words and their associated tags. For 32Kxl2 main memory and 512x12 cache memory 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.



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

Figure 12-15 Two-way set-associative mapping cache.

Writing into Cache:

→ An important aspect of cache organization is concerned with memory write requests. When the CPU finds a word in cache during a read operation, the main memory is not involved in the transfer. However, if the operation is a write, there are two ways that the system can proceed.

→The simplest and most commonly used procedure is to update 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**. This method has the advantage that main memory always contains the same data as the cache. This characteristic is important in systems with direct memory access transfers. It ensures that the data residing in main memory are valid at all times so that an I/O device communicating through DMA would receive the most recent updated data.

→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 word is removed from the cache it is copied into main memory. The reason for the write-back method is that during the time a word resides in the cache, it may be updated several times; however, as long as the word remains in the cache, it does not matter whether the copy in main memory is out of date, since requests from the word are filled from the cache.

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, but 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. The introduction of the valid bit means that a word in cache is not replaced by another word unless the valid bit is set to 1 and a mismatch of tags occurs. If the valid bit happens to be 0, the new word automatically replaces the invalid data. Thus the initialization condition has the effect of forcing misses from the cache until it fills with valid data.

By Sreenu Konda (9490970060)

6.Virtual Memory:

→In a memory hierarchy system, programs and data are first stored in auxiliary memory. Portions of a program or 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.

→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.

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. The address space is allowed to be larger than the memory space in computers with virtual memory.

→As example for 1024k auxiliary memory and 32k main memory then

Virtual address bits =20	Physical address bits =15
Address space =1024k	Memory space = 32k

→In a multi-program 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 are moved from auxiliary memory into main memory as shown in **Figure 12-16.**

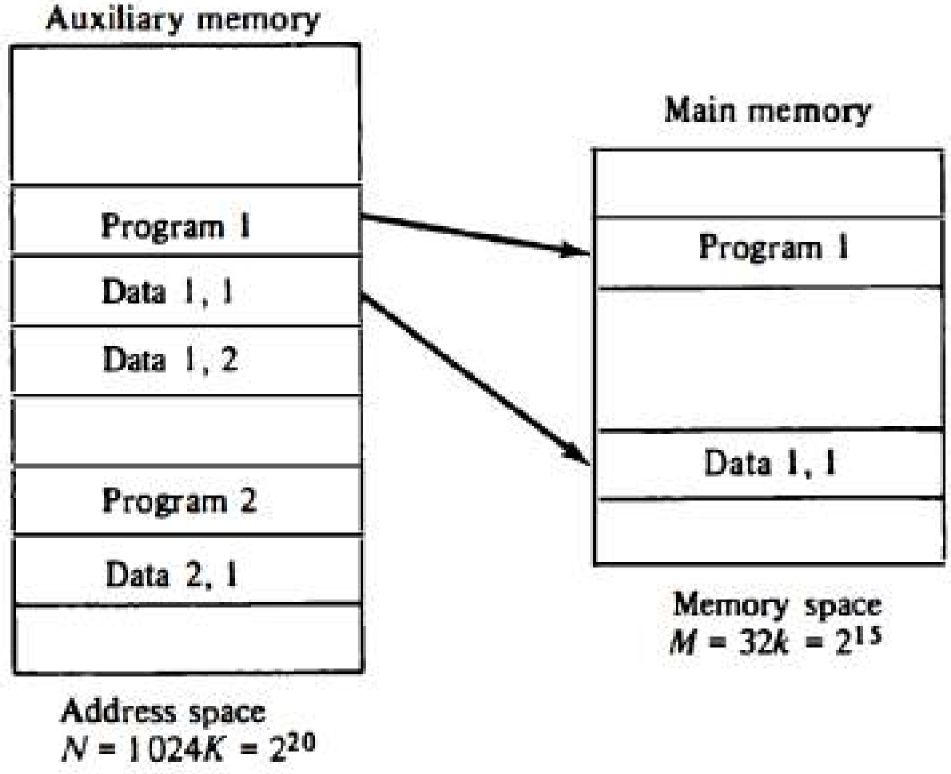
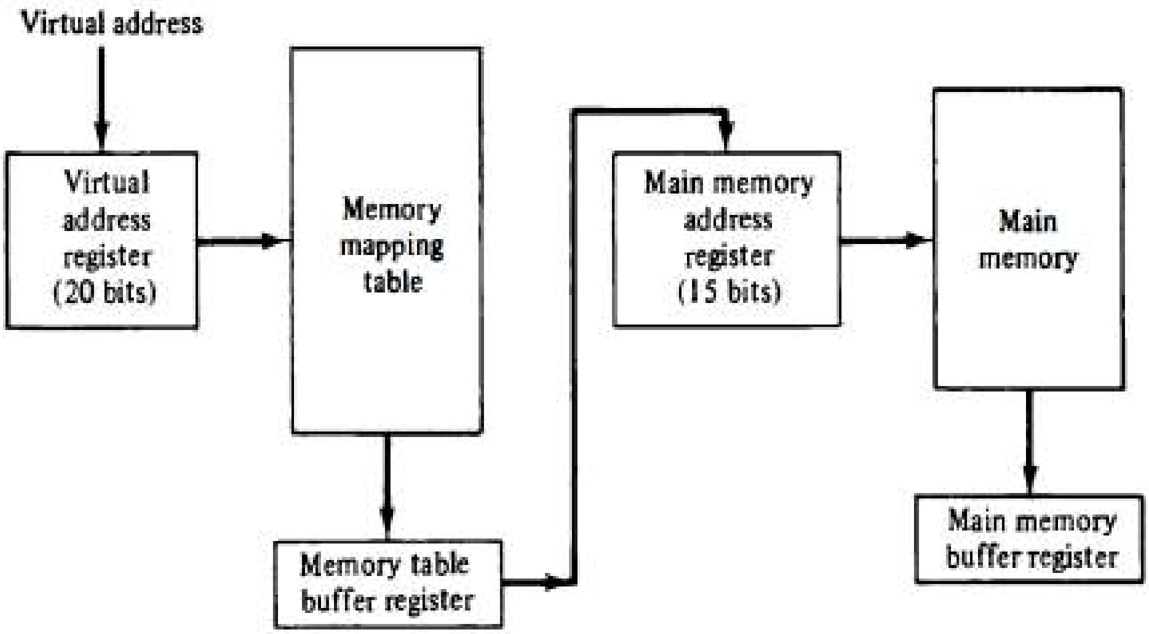


Figure 12-16 Relation between address and memory space in a virtual memory system.

→A table is then needed, as shown in Figure 12-17, to map a virtual address of 20 bits to a physical address of 15 bits. The mapping is a dynamic operation which means that every address is translated immediately as a word is referenced by CPU.

→The mapping table may be stored in a separate memory as shown **Figure 12-17** or in main memory. In the first case, an additional memory unit 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.

Figure 12-17 Memory table for mapping a virtual address.


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

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.

→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 **Figure 12-18**. At any given time, up to four pages of address space may reside in main memory in any one of the four blocks.

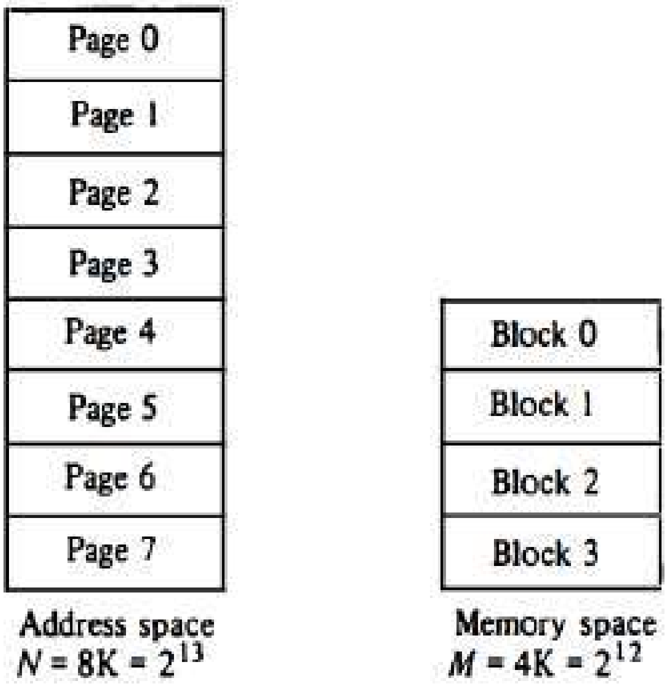


Figure 12-18 Address space and memory space split into groups of 1K words.

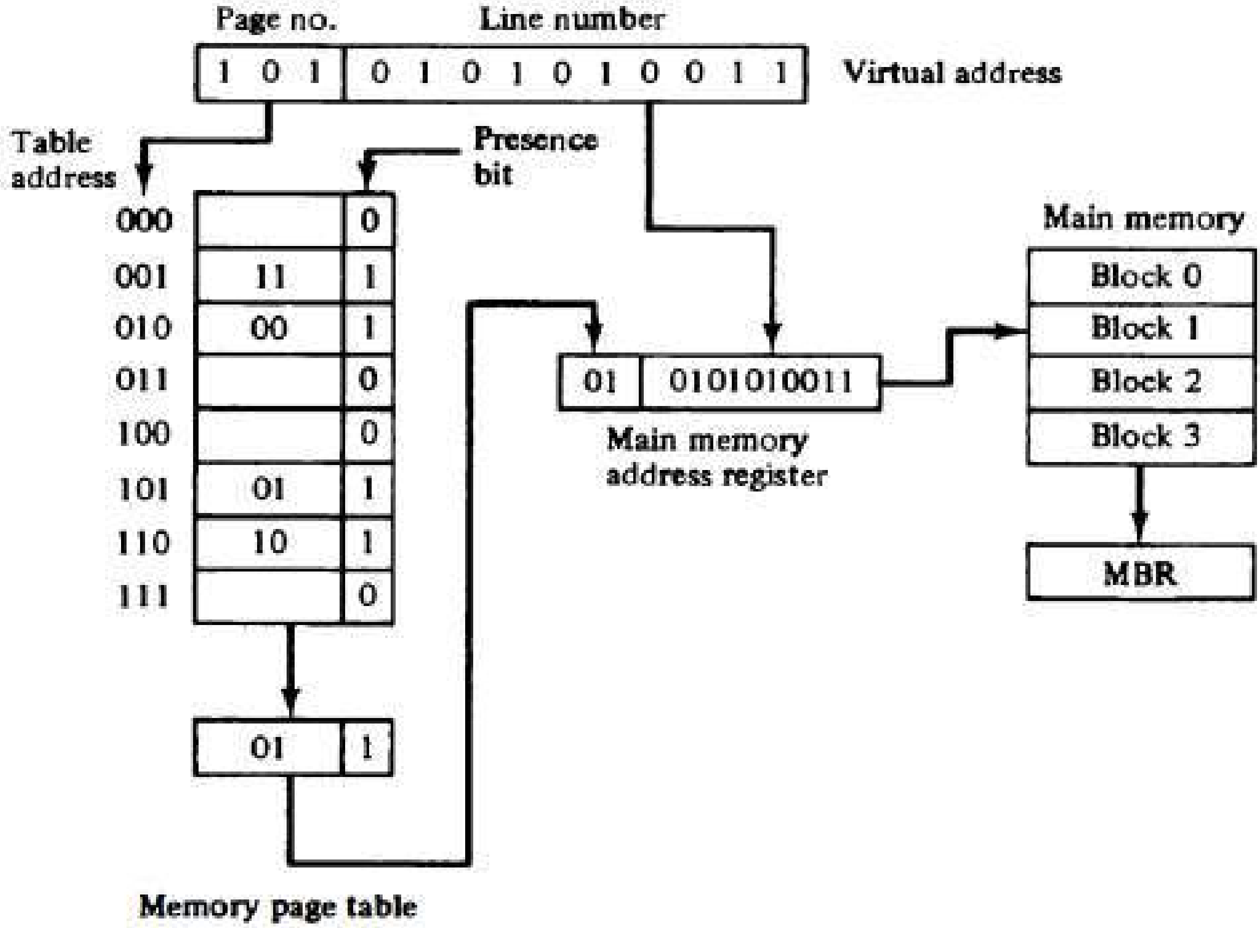
→The mapping from address space to memory space is facilitated if each virtual address is considered to be represented by two numbers: a page number address and a line (word) within the page. In a computer with 2 words per page, p bits are used to specify a line address (word) and the remaining high-order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

→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.

→The organization of the memory mapping table in a paged system is shown in Figure bellow. This figure for 8k x 12 auxiliary memory and 4k x 12 main memory with block size = page size = 1k

→The organization of memory mapping table in page system is shown in **Figure 12-19**. The memory-page table consists of eight words, one for each page. The address in the page table denotes the page number and the content of the word gives the block number where that page is stored in main memory. The table shows that pages 1, 2, 5, and 6 are now available in main memory in blocks 3, 0, 1, and 2, respectively. A presence bit in each location indicates whether the page has been transferred from auxiliary memory into main memory.

Figure 12-19 Memory table in a paged system.



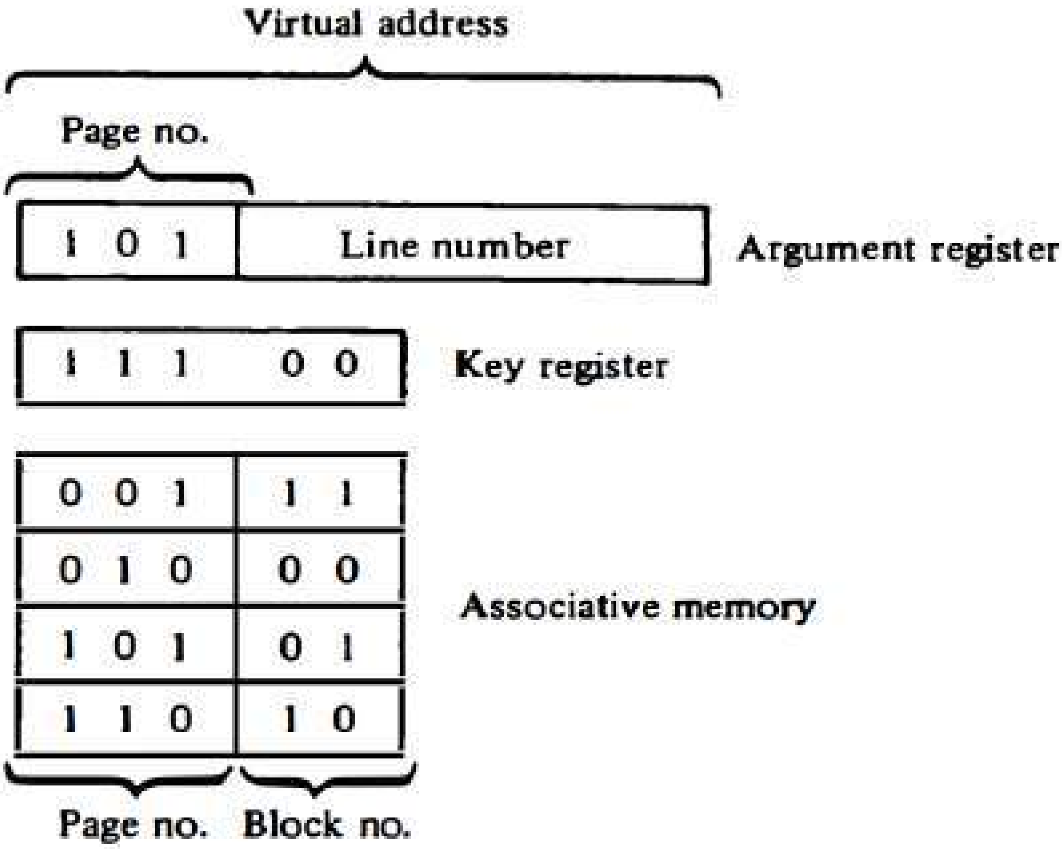
<u>Associative Memory Page Table:</u>

→A random-access memory page table is inefficient with respect to storage utilization.

→A more 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.

→<u>Consider again the case of eight pages and four blocks as in the example of **Figure 12-19**. We replace the random access memory-page table with an associative memory of four words as shown in **Figure 12-20**</u>. Each entry in the associative memory array consists of two fields. The first three bits specify a field for 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 register 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.

Figure 12-20 An associative memory page table.



Replacement algorithm:

→When a program starts execution, one or more pages are transferred into main memory and the page table is set to indicate their position. 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 a miss occur in a Cache memory and the Cache is full, it is necessary to replace one word with new word from main memory. The most common replacement algorithms are:-

1- Random Replacement: Select the item randomly.

2- 2- FIFO (First-In First-Out): Select the item has been in the Cache the longest.

3- 3- LRU (Least Recently Used): Select the item that has been least recent used by the CPU.

7. Peripheral Devices:

I/O subsystem:

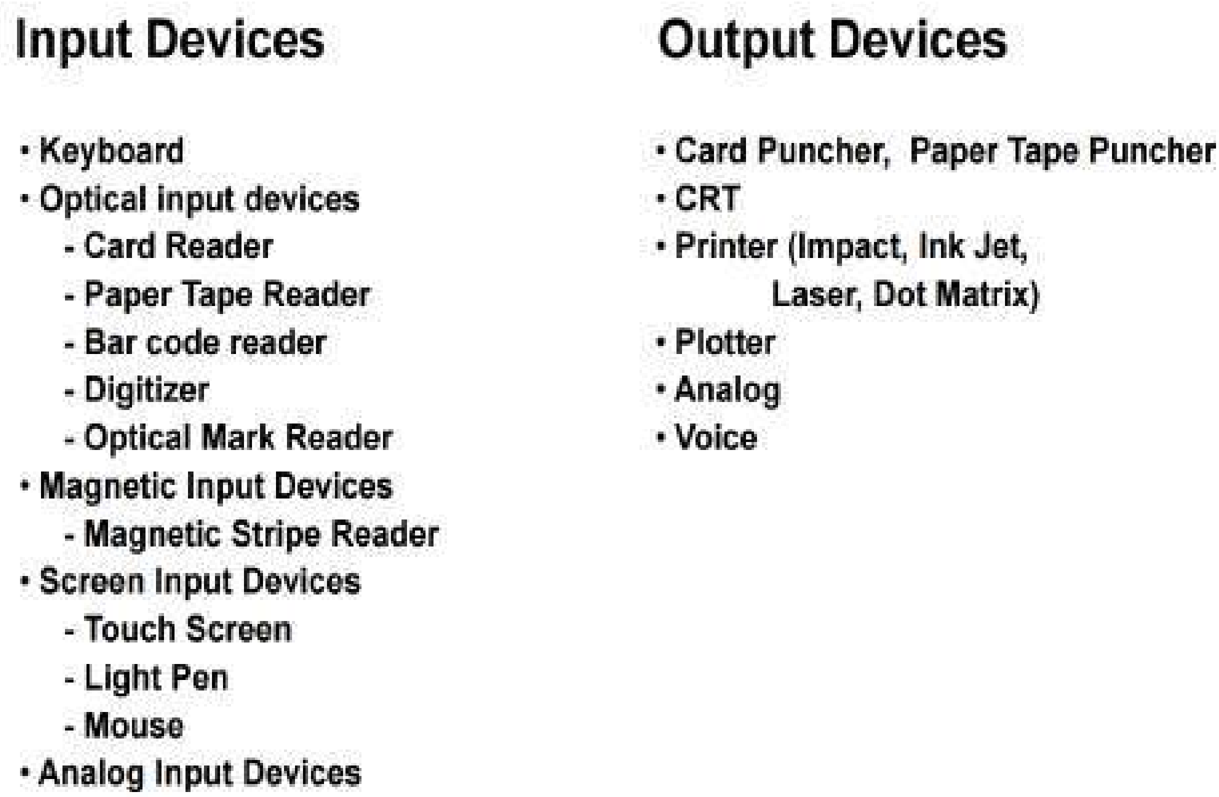
→The *input-output subsystem* (also referred as I/O) proves an efficient mode of communication between the central system and outside environment. Data and programs must be entered into the computer memory for processing and result of processing must be must be recorded or displayed for the user.

Peripheral devices:

→Any input/output devices connected to the computer are called **peripheral devices**.

→**Input devices** are used to put the information into computer. With the help of input devices we can store information in memory so that CPU can use it. Program or data is read into main memory from input device or secondary storage under the control of CPU input instruction.

→**Output devices** are used to output the information from computer. If some results are evaluated by computer and it is stored in computer, then with the help of output devices, we can present it to the user. Output data from the main memory go to output device under the control of CPU output instruction.



Input Devices

- Keyboard
- Optical input devices
  - Card Reader
  - Paper Tape Reader
  - Bar code reader
  - Digitizer
  - Optical Mark Reader
- Magnetic Input Devices
  - Magnetic Stripe Reader
- Screen Input Devices
  - Touch Screen
  - Light Pen
  - Mouse
- Analog Input Devices

Output Devices

- Card Puncher, Paper Tape Puncher
- CRT
- Printer (Impact, Ink Jet, Laser, Dot Matrix)
- Plotter
- Analog
- Voice

→The computer **keyboard,** based on the typewriter keyboard, contains keys f or entering letters, numbers, and punctuation marks, as well as keys to change th e meaning of other keys. The function keys perform tasks that vary fromprogram to program.

→The **mouse** is a device that is rolled on the desktop to move the cursor onthe screen. A ball on the bottom of the mouse translates the device's movements to sensors within the mouse and then through the connecting port to the computer.

→The **printer** puts text or other images produced with a computer onto paper or other surfaces. Printers are either impact or nonimpact devices.

→The **joystick** is a pointing device used principally for games.
→The **light pen** performs the same functions as a mouse or trackball, but it is held up to the screen, where its sensors detect the presence of pixels and send a signal through a cable to the computer.

→The graphics, or **digitizing**, tablet is a pad with electronics beneath the surface which is drawn upon with a pointed device, called a stylus. The shapes drawn appear on the monitor's screen.
→The Cathode Ray Tube (***CRT***) contains an electronic gun that sends an electronic beam to a phosphorescent screen in front of the tube.

→The **monitor** is the device on which images produced by the computer operator or generated by the program are displayed on a cathode-ray tube (CRT).

→**Magnetic tapes** are used mostly for storing files of data: for example, company's payroll system.
→**Magnetic disks** have high-speed rotational surfaces coated with magnetic material.

ASCII Alphanumeric Characters:

→Standard binary code for the alphanumeric characters is **ASCII** (American Standard Code for Information Interchange).
→ASCII uses 7 bits to code 128 characters as shown in **Table11-1**.

- 94 printable characters and 34 non printable characters.

Printable characters consist of:

- 26 uppercase letters - (A through Z)
- 26 lowercase letters - (a through z)
- 10 numerals – (0 through 9)
- 32 special printable characters such as %, *, and $.

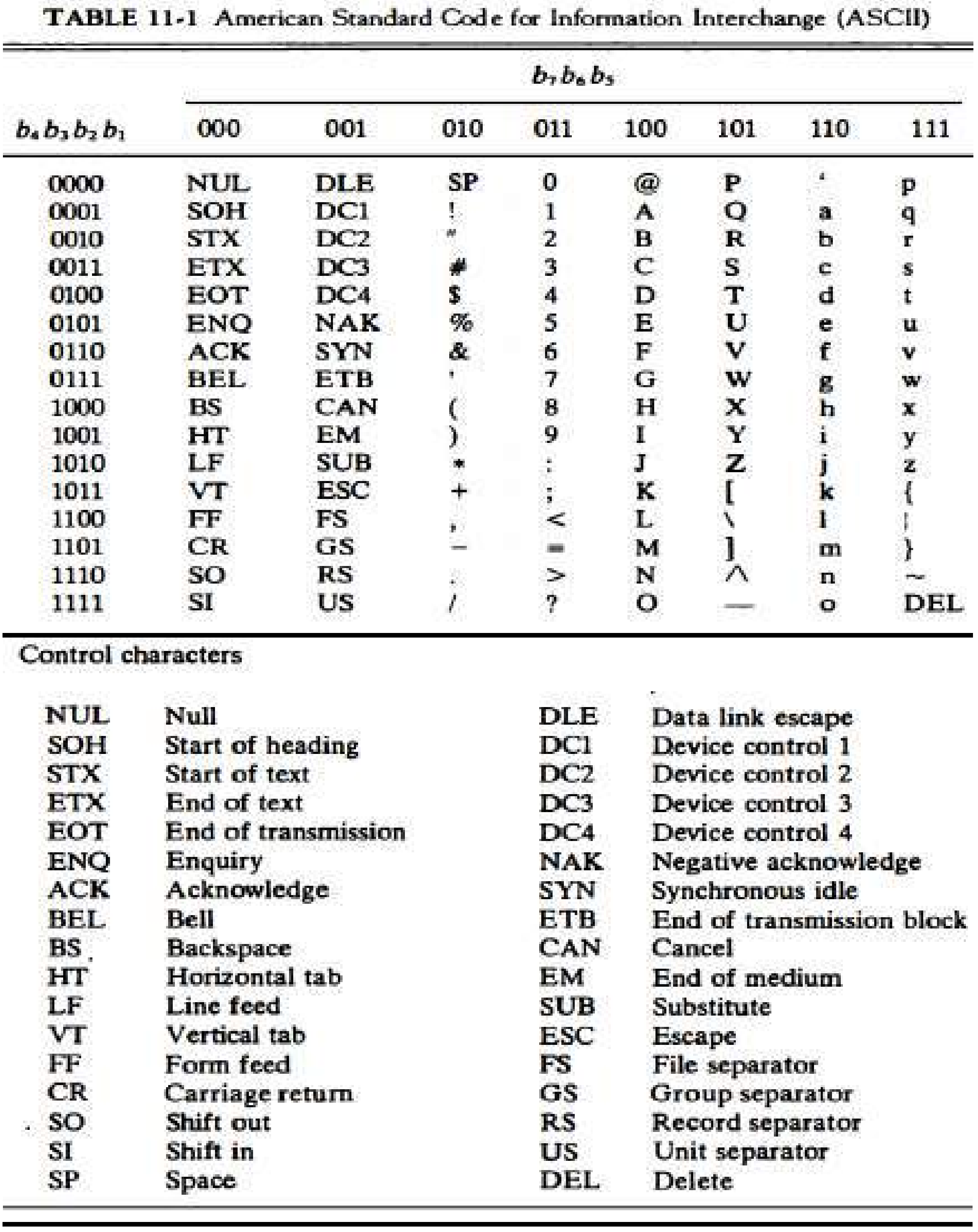


TABLE 11-1 American Standard Code for Information Interchange (ASCII)

	$b_7 b_6 b_5$							
$b_4 b_3 b_2 b_1$	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	'	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	¦
1101	CR	GS	−	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	—	o	DEL

Control characters

NUL	Null	DLE	Data link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of transmission	DC4	Device control 4
ENQ	Enquiry	NAK	Negative acknowledge
ACK	Acknowledge	SYN	Synchronous idle
BEL	Bell	ETB	End of transmission block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete

8.Input-Output Interface:

→Input-output interface provides a method for transferring information between internal storage and external I/O devices.

→The main purpose of I/O interface is to resolve differences between CPU and the peripherals.

The major differences between the central computer and each peripheral are:

1. Peripherals are electromechanical and electromagnetic devices and their manner of operation is different from the operation of the CPU and memory.
2. The data transfer rate of peripherals is usually slower than the transfer rate of the CPU.
3. Data codes and formats in peripherals differ from 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.

→To resolve these differences, computer systems include special hardware components between the CPU and peripherals to supervise and synchronize all input and output transfers. These components are called **interface** units because they interface between the processor bus and the peripheral device.

I/O Bus and Interface Modules:

→A typical communication link between the processor and several peripherals is shown in **Figure. 11-1.**

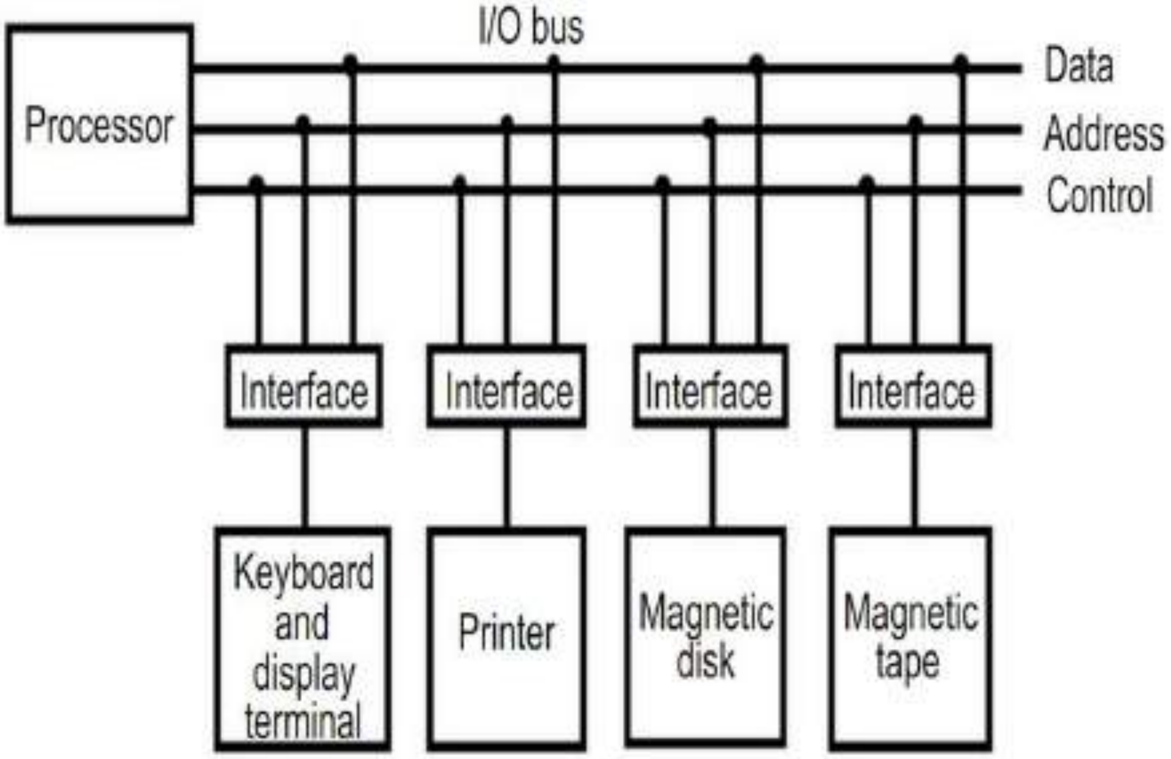


Figure 11-1 Connection of I/O bus to input-output devices.

→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. When the interface detects its own address, it activates the path between the bus lines and the device that it controls.

→**Functions of an interface are as below:** -

- Decodes the device address (device code).
- Decodes the I/O commands (operation or function code).
- Provides signals for the peripheral controller.
- Synchronizes the data flow and
- Supervises the transfer rate between peripheral and CPU or Memory.

→At the same time that the address is made available in the address lines, the processor provides a function code in the control lines(I/O command) which is one of the types:-

1. A **control command** is issued to activate the peripheral and to inform it what to do. For (a magnetic tape unit may be instructed to backspace tape by one record, to rewind the tape, or to start the tape moving in forward direction).
2. **A status command** is used to test various status conditions in the interface and the peripheral (the computer may wish to check the status of the peripheral before a transfer is initiated).
3. **A data output command** causes the interface to respond by transferring from the bus into one of its registers.
4. **Data input command** is the opposite of the data output. In this case interface receives an item of data from the peripheral and places it in its buffer register.

I/O versus Memory Bus:

→**I/O Bus:** Communication between CPU and all interface units is via a common I/O bus. An interface connected to a peripheral device may have a number of data registers, a control register, and a status register. A command is passed to the peripheral by sending to the appropriate interface register

→**Memory bus:** used for information transfers between CPU and the MM (main memory). I/O bus is for information transfers between CPU and I/O devices through their I/O interface.

→There are three ways that computer buses a 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.

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. The memory read and memory write control lines are enabled during a memory transfer. This configuration isolates all I/O interface addresses from the addresses assigned to memory and is referred to as the **isolated I/O method** for assigning addresses in a common bus.

→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**. The computer treats an interface register as being part of the memory system. The assigned addresses for interface registers cannot be used for memory words, which reduce the memory address range available.

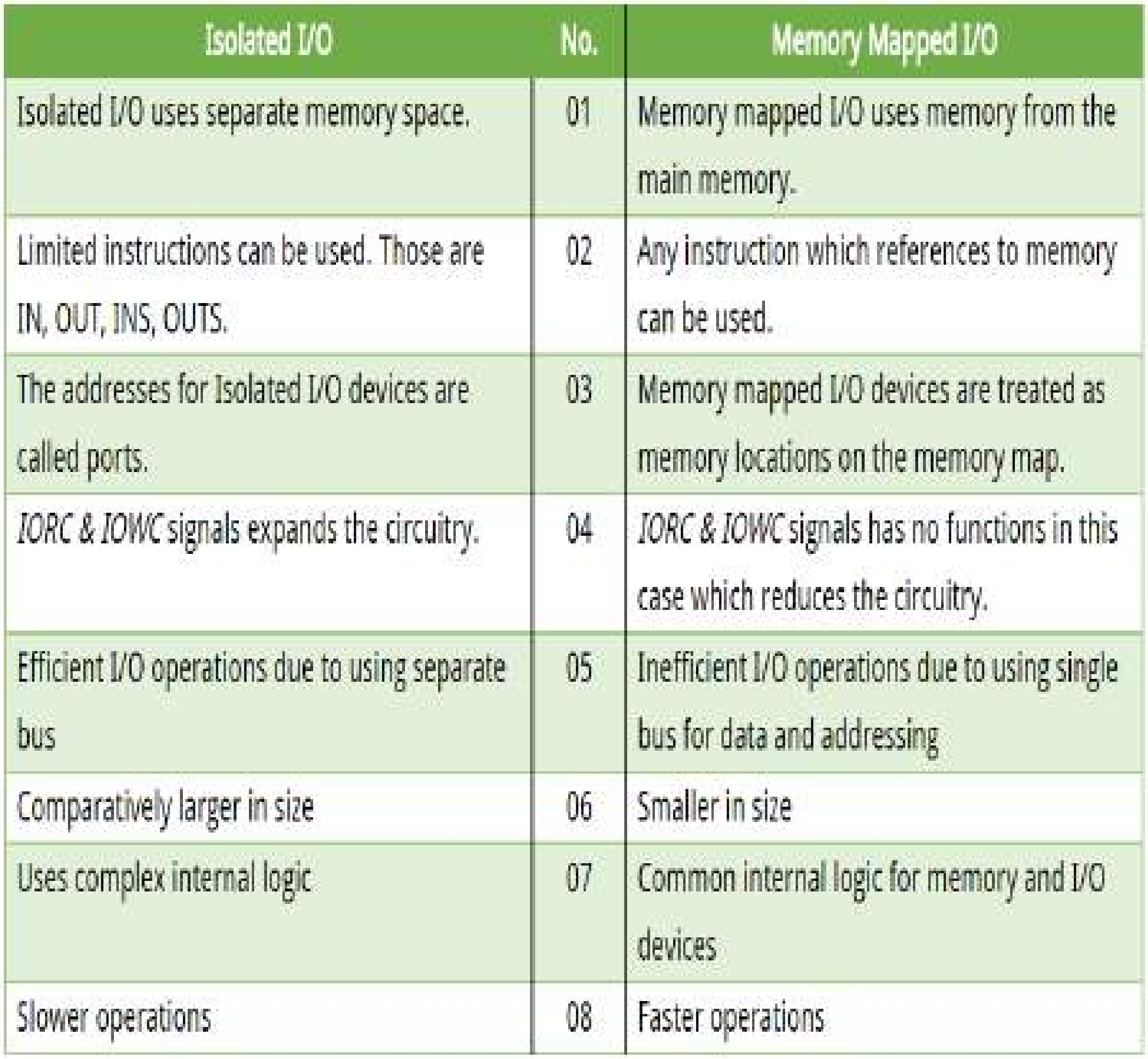
- In a memory-mapped I/O organization there are no specific inputs or output instructions.
- In a typical computer, there are more memory-reference instructions than I/O instructions. With memory mapped I/O all instructions that refer to memory are also available for I/O.

Differences between Isolated I/O and Memory Mapped I/O.

→**Memory Mapped I/O and Isolated I/O** are two methods of performing input-output operations between CPU and installed peripherals in the system. Memory mapped I/O uses the same address bus to connect both primary memory and memory of hardware devices. Thus the instruction to address a section or portion or segment of RAM can also be used to address a memory location of a hardware device.

→On the other hand, **isolated I/O** uses separate instruction classes to access primary memory and device memory. In this case, I/O devices have separate address space either by separate I/O pin on CPU or by entire separate bus. As it separates general memory addresses with I/O devices, it is called **isolated I/O.**

Differences Between Isolated I/O and Memory Mapped I/O:



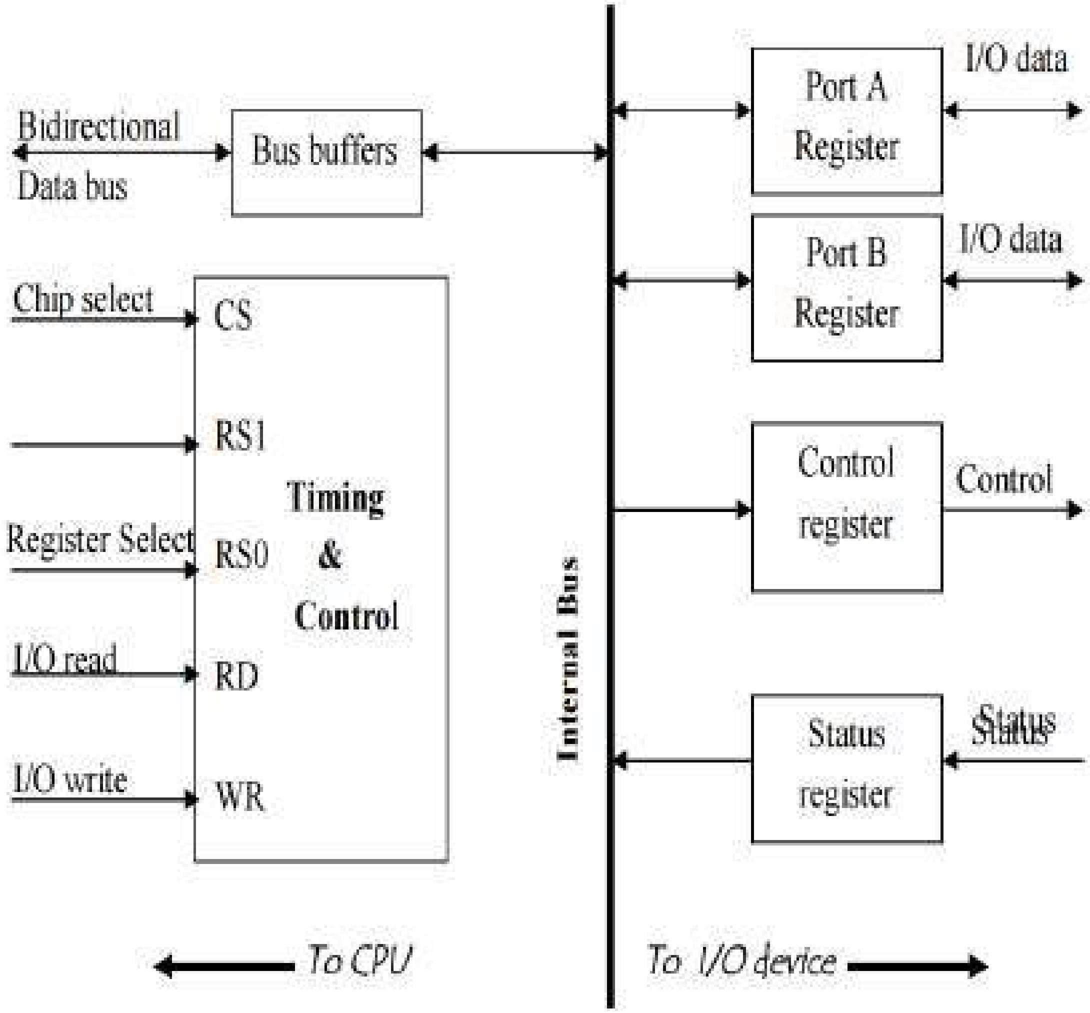
Isolated I/O	No.	Memory Mapped I/O
Isolated I/O uses separate memory space.	01	Memory mapped I/O uses memory from the main memory.
Limited instructions can be used. Those are IN, OUT, INS, OUTS.	02	Any instruction which references to memory can be used.
The addresses for Isolated I/O devices are called ports.	03	Memory mapped I/O devices are treated as memory locations on the memory map.
IORC & *IOWC* signals expands the circuitry.	04	*IORC* & *IOWC* signals has no functions in this case which reduces the circuitry.
Efficient I/O operations due to using separate bus	05	Inefficient I/O operations due to using single bus for data and addressing
Comparatively larger in size	06	Smaller in size
Uses complex internal logic	07	Common internal logic for memory and I/O devices
Slower operations	08	Faster operations

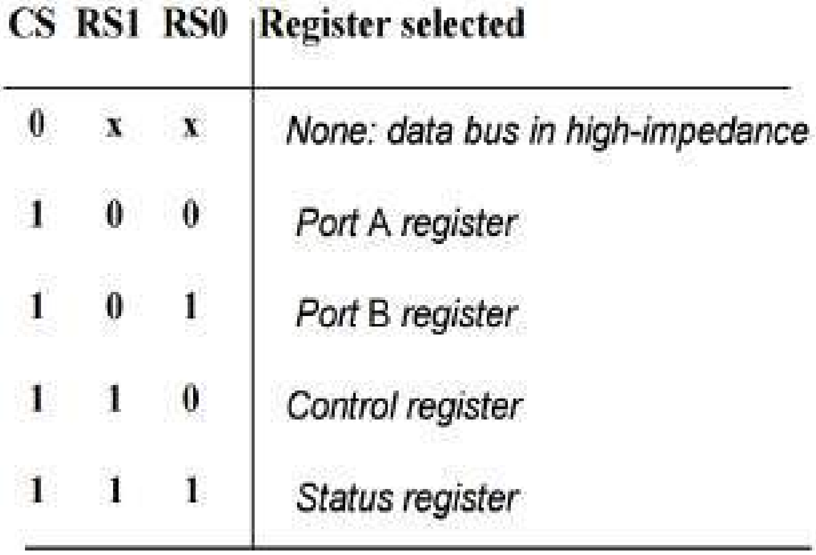
Example of I/O Interface:

→An example of an I/O interface unit is shown in block diagram form in **Figure 12-2.** It consists of two data registers called ports, a control register, a status register and bus buffers, and timing and control circuits. The interface communicates with the CPU through the data bus. The chip select and register select input determine the address assigned to the interface. The I/O read and

writes are control lines that specify an input or output, respectively. The four registers communicate directly with the I/O device attached to the interface.
→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 RS0 and RS1 are usually connected to the two least significant lines of the address bus. These two inputs select one of the four registers in the interface.

→Bidirectional lines represent both data in and out from the CPU. Information in each port can be assigned a meaning depending on the mode of operation of the I/O device: Port A = Data; Port B =Command; Port C = Status. CPU initializes (loads) each port by transferring a byte to the Control Register. CPU can define the mode of operation of each port.





CS	RS1	RS0	Register selected
0	x	x	*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*

Figure 11-2.Example of I/O Interface unit.

9. Asynchronous Data Transfer:

→In a computer system, CPU and an I/O interface are designed independently of each other.When internal timing in each unit is independent from the other and when registers in interface and registers of CPU uses its own private clock.
→In that case the two units are said to be asynchronous to each other. CPU and I/O device must coordinate for data transfers.

Strobe Control: This is one way of transfer i.e. by means of strobe pulse supplied by one of the units to indicate to the other unit when the transfer has to occur.
Handshaking: This method is used 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.

METHODS USED IN ASYNCHRONOUS DATA TRANSFER:

- **Strobe Control**
- **Handshaking**

Strobe Control:

→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. The strobe may be activated by either source or the destination unit. The Strobe signal is disabled indicates that the data bus does not contain valid data. New valid data will be available only after the strobe is enabled again.

→Strobe control method of data transfer uses a single control signal for each transfer. The strobe may be activated by either the source unit or the destination unit.

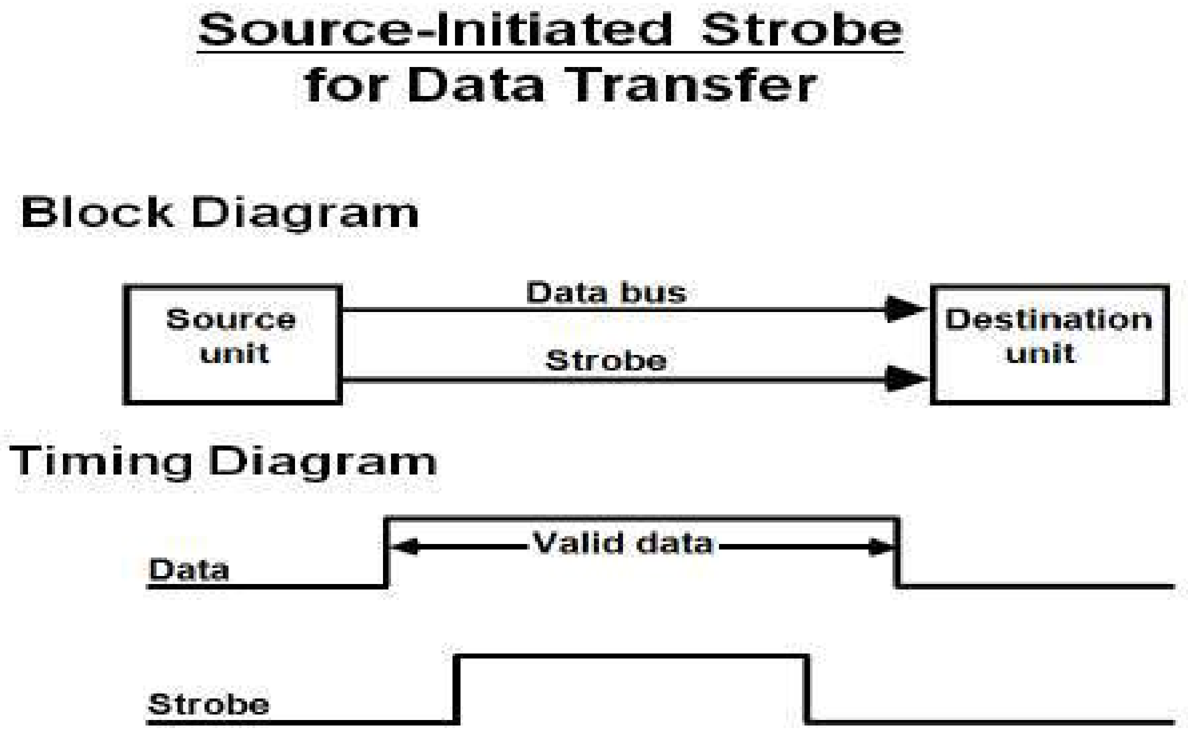
- **Source Initiated Strobe**
- **Destination Initiated Strobe**

SOURCE INITIATED STROBE:

→The data bus carries the binary information from source unit to the destination unit as shown below.

→The strobe is a single line that informs the destination unit when a valid data word is available in the bus. The source unit first places the data on the bus. After a brief delay to ensure that the data settle to a steady value, the source activities the strobe pulse.

→The information of the data bus and the strobe signal remain in the active state for a sufficient time period to allow the destination unit to receive the data. The source removes the data from the bus for a′ brief period of time after it disables its strobe pulse.



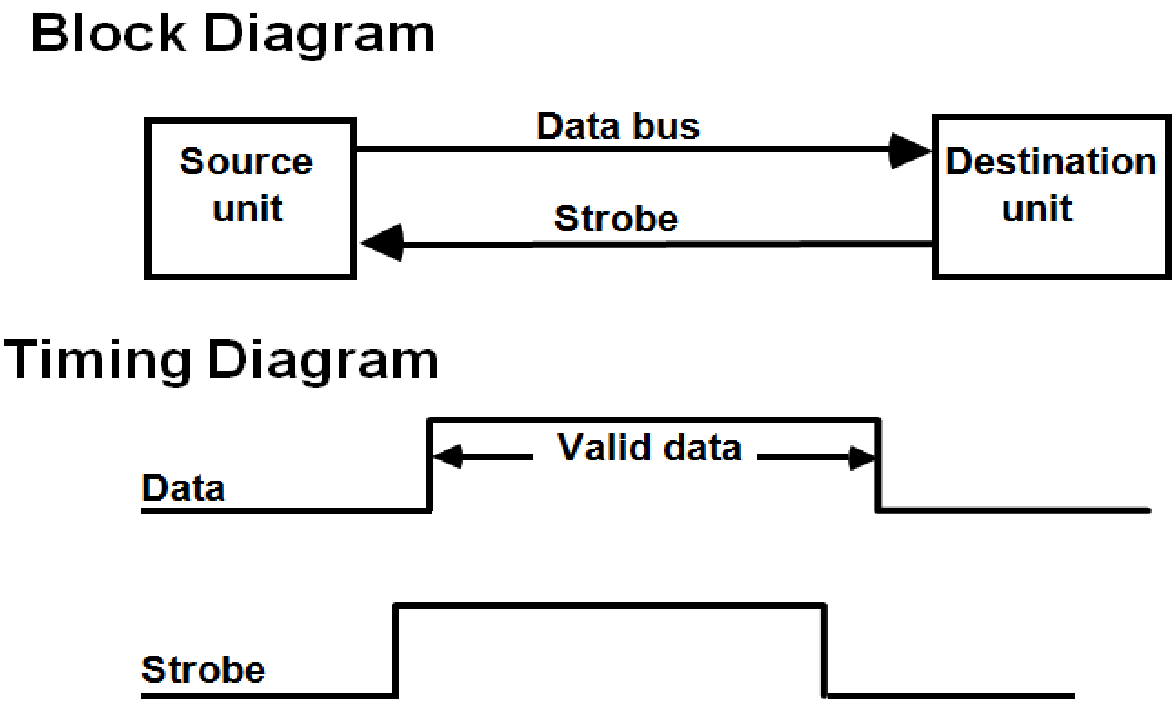
Source-Initiated Strobe for Data Transfer

DESTINATION INITIATED STROBE:

→First, the destination unit activates the strobe pulse, informing the source to provide the data. The source unit responds by placing the requested binary information on the unit to accept it.

→The data must be valid and remain in the bus long enough for the destination unit to accept it. The falling edge of the strobe pulse can be used again to trigger a destination register. The destination unit then disables the strobe. The source removes the data from the bus after a predetermined time interval.

Destination-Initiated Strobe
for Data Transfer



HANDSHAKING:

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 handshaking may be activated by either source or the destination unit.

→The **disadvantage** of the strobe method is that the source unit that initiates 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 that initiates the transfer has no way of knowing whether the source unit has actually placed the data on the bus. The handshaking method solves this problem by introducing a second control signal that provides a reply to the unit that initiates the transfer.

→There are two control lines in handshaking technique:

- **Source to destination unit.**
- **Destination to source unit.**

SOURCE INITIATED TRANSFER:

→Handshaking signals are used to synchronize the bus activities. 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 exchange of signals between two units.

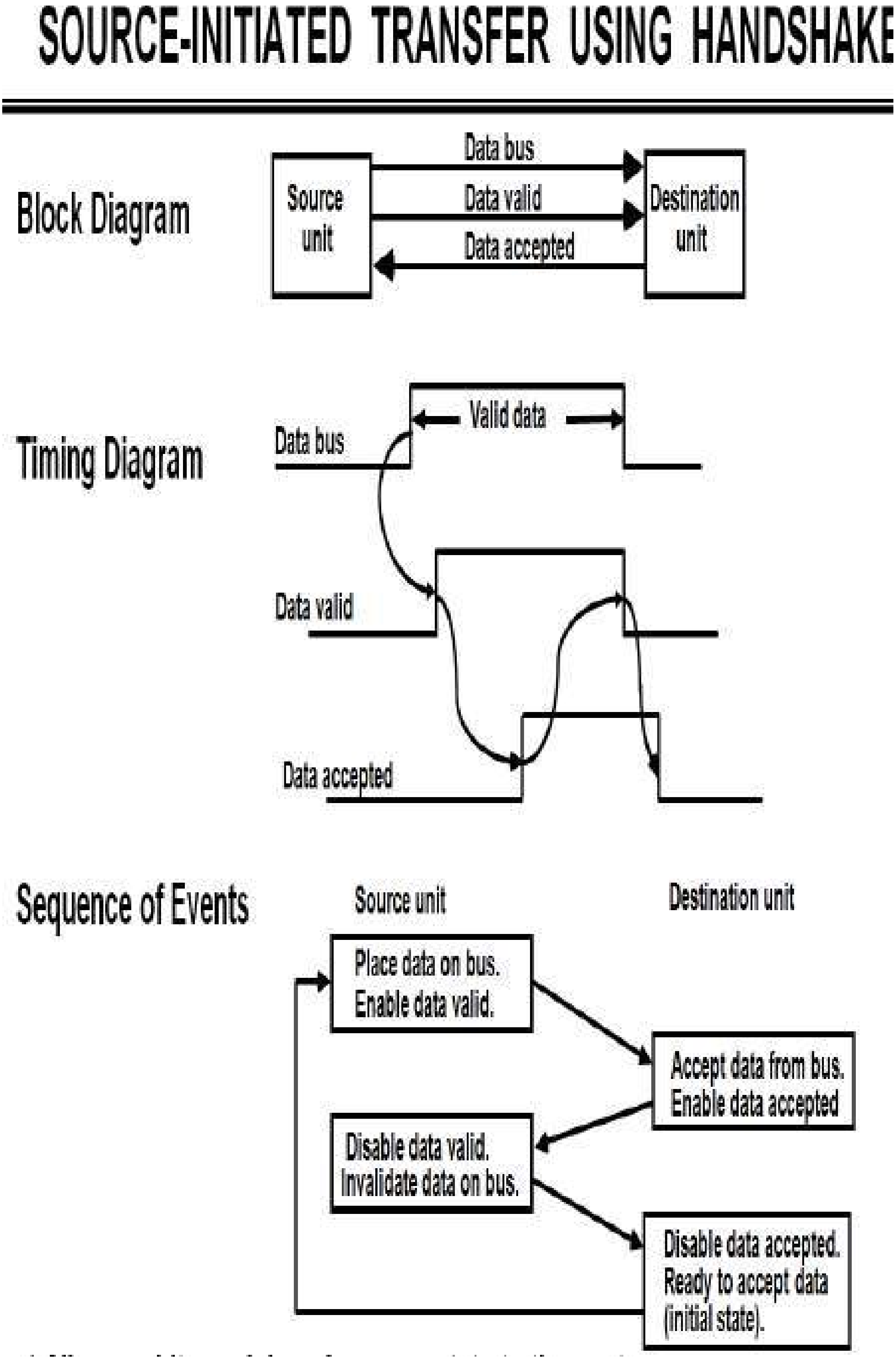
SOURCE INITIATED TRANSFER USING HANDSHAKING:

→The sequence of events:

- The source unit initiates the transfer by placing the data on the bus and enabling its data valid signal.
- The data accepted signals are 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.
- The destination unit the disables its data accepted signal and the system goes into its initial state.

DESTINATION INITIATED TRANSFER USING HANDSHAKING:

- In this case the name of the signal generated by the destination unit is ready for data.
- The source unit does not place the data on the bus until it receives the ready for data signal from the destination unit.
- The handshaking procedure follows the same pattern as in source initiated case. The sequence of events in both the cases is almost same except the ready for signal has been converted from data accepted in case of source initiated.



SOURCE-INITIATED TRANSFER USING HANDSHAKE

Place data on bus.
Enable data valid.

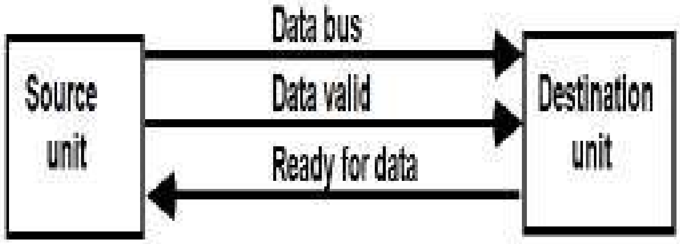
Accept data from bus.
Enable data accepted

Disable data valid.
Invalidate data on bus.

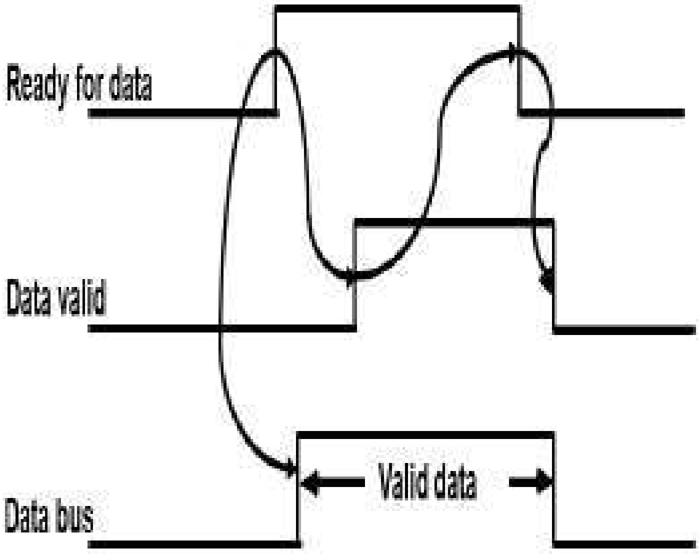
Disable data accepted.
Ready to accept data
(initial state).

DESTINATION-INITIATED TRANSFER USING HANDSHAKE

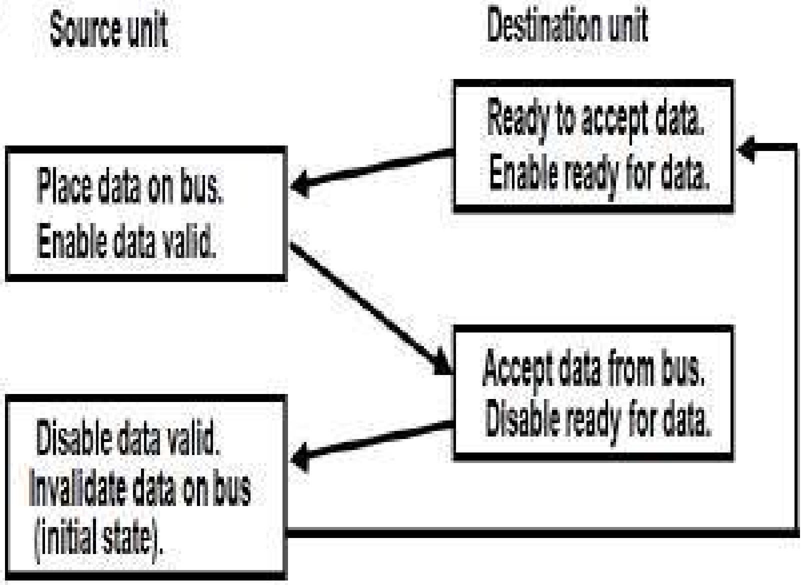
Block Diagram



Timing Diagram



Sequence of Events


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).

Asynchronous Serial Transfer:

→A serial asynchronous data transmission technique used in many interactive terminals employs special bits which are inserted at both ends of the character code.

→With this technique each character consists of three parts:

- Start bit: is always a 0 and is used to indicate the beginning of a character.
- Character bits: data.
- Stop bits: is always a 1.

→An example of this format is shown in **Figure. 11-7.**

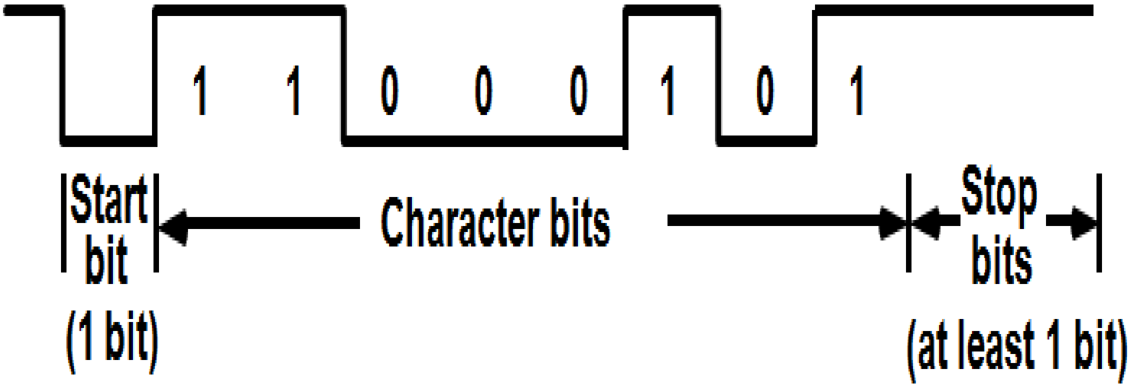


Figure 11-7 Asynchronous serial transmission

→A transmitted character can be detected by the receiver from knowledge of the 4 transmission rules:

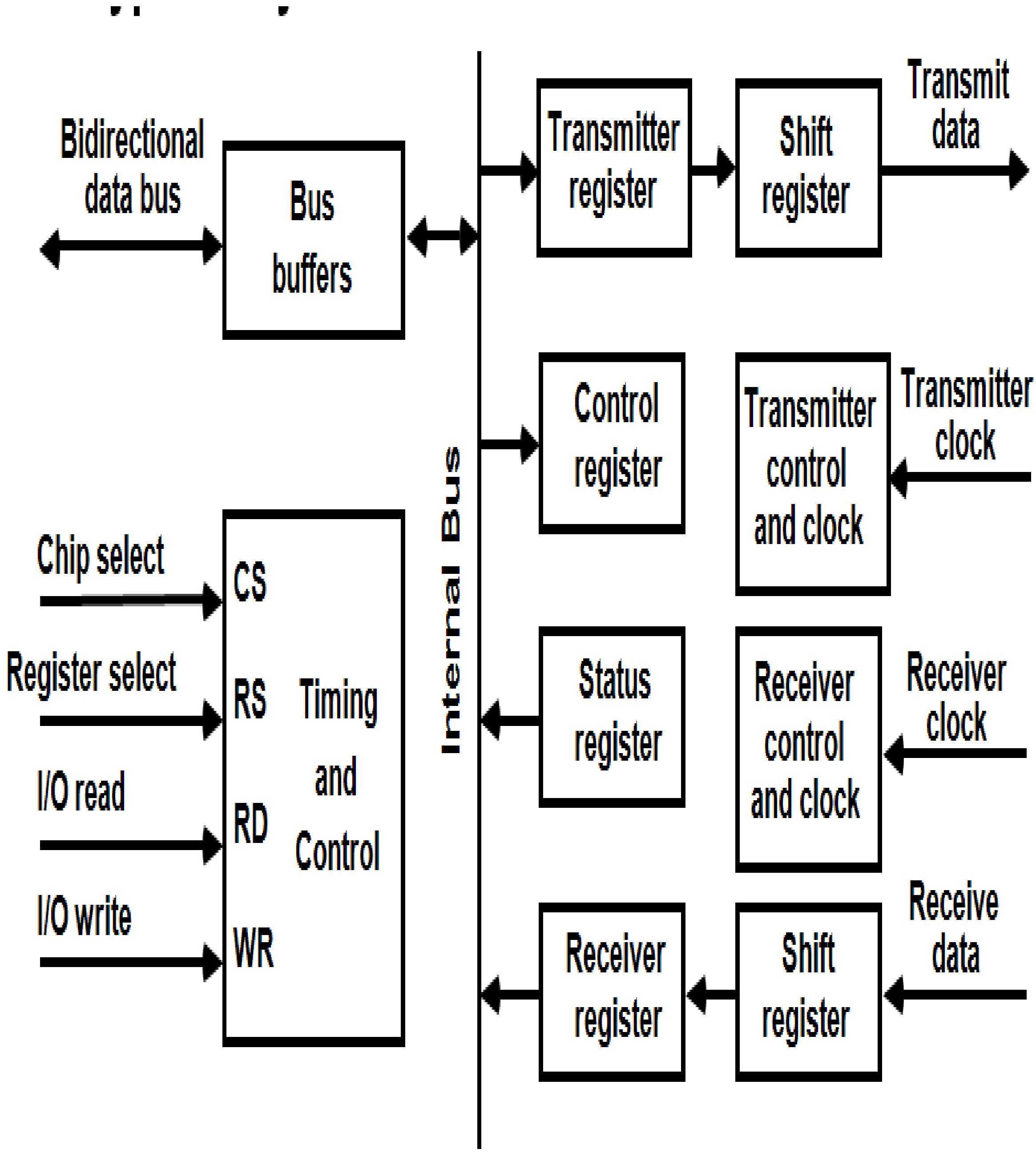
1. When a character is not being sent, the line is kept in the 1-state (idle state).
2. The initiation of a character transmission is detected from the *Start Bit,* which is always a 0.
3. The character bits always follow the *Start Bit.*
4. After the last bit of the character is transmitted, a *Stop Bit* is detected whenthe line returns to the 1-state for at least 1 bit time.

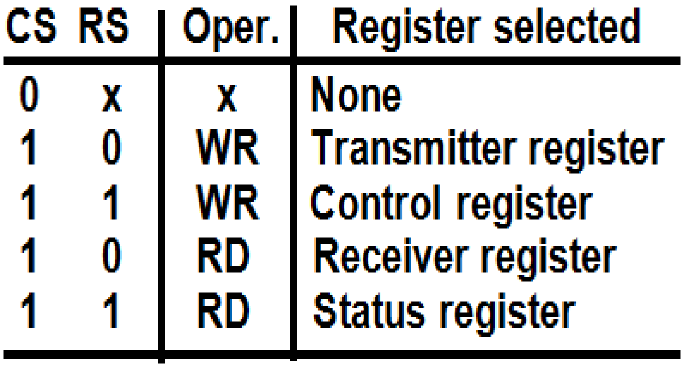
→As **illustration,** consider the serial transmission of a terminal whose transfer rate is 10 characters per second. Each transmitted character consists of a start bit, eight information bits, and two stop bits, for a total of 11 bit. Ten characters per second that means each character take 0.1 s for transfer. Since there are 11 bits to be transmitted, it follows that the bit time is 9.09 ms.

→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.
→10 characters per second with an 11 bit format have a transfer rate of 110 baud.

Asynchronous Communication Interface:
(UNIVERSAL ASYNCHRONOUS RECEIVER-TRANSMITTER -UART):

→The block diagram of A typical asynchronous communication interface is shown in**Figure. 11-8 .** It functions as both a transmitter and receiver.





CS	RS	Oper.	Register selected
0	x	x	None
1	0	WR	Transmitter register
1	1	WR	Control register
1	0	RD	Receiver register
1	1	RD	Status register

Figure 11-8 Block diagram of a typical asynchronous communication interface.

→It works as both a receiver and a transmitter. Its operation is initialized by CPU by sending a byte to the control register.

Transmitter:

→The operation of the transmitter portion of the interface is as follows. The CPU reads the status register and checks the flag. If the transmitter register is empty, the CPU transfers a character to the transmitter register and the interface clears the flag to mark the register full. The first bit in the transmitter shift register is set to 0 to generate a start bit. The character is transferred in parallel from the transmitter register to the shift register and the appropriate numbers of stop bits are appended into the shift register.

→The transmitter register is then marked empty. The character can now be transmitted one bit at a time by shifting the data in the shift register at the specified baud rate. The CPU can transfer another character to the transmitted register after checking the flag in the status register. The interface is said to be double buffered because a new character can be loaded as soon as the previous one starts transmission.

→The **transmitter register** accepts a data byte from CPU through the data bus and transferred to a shift register for serial transmission.

Receiver:

→The operation of the receiver portion of the interface is similar.

→The **receive portion** receives information into another shift register, and when a complete data byte is received it is transferred to receiver register. CPU can select the receiver register to read the byte through the data bus. Data in the status register is used for input and output flags.

First In First Out Buffer (FIFO):

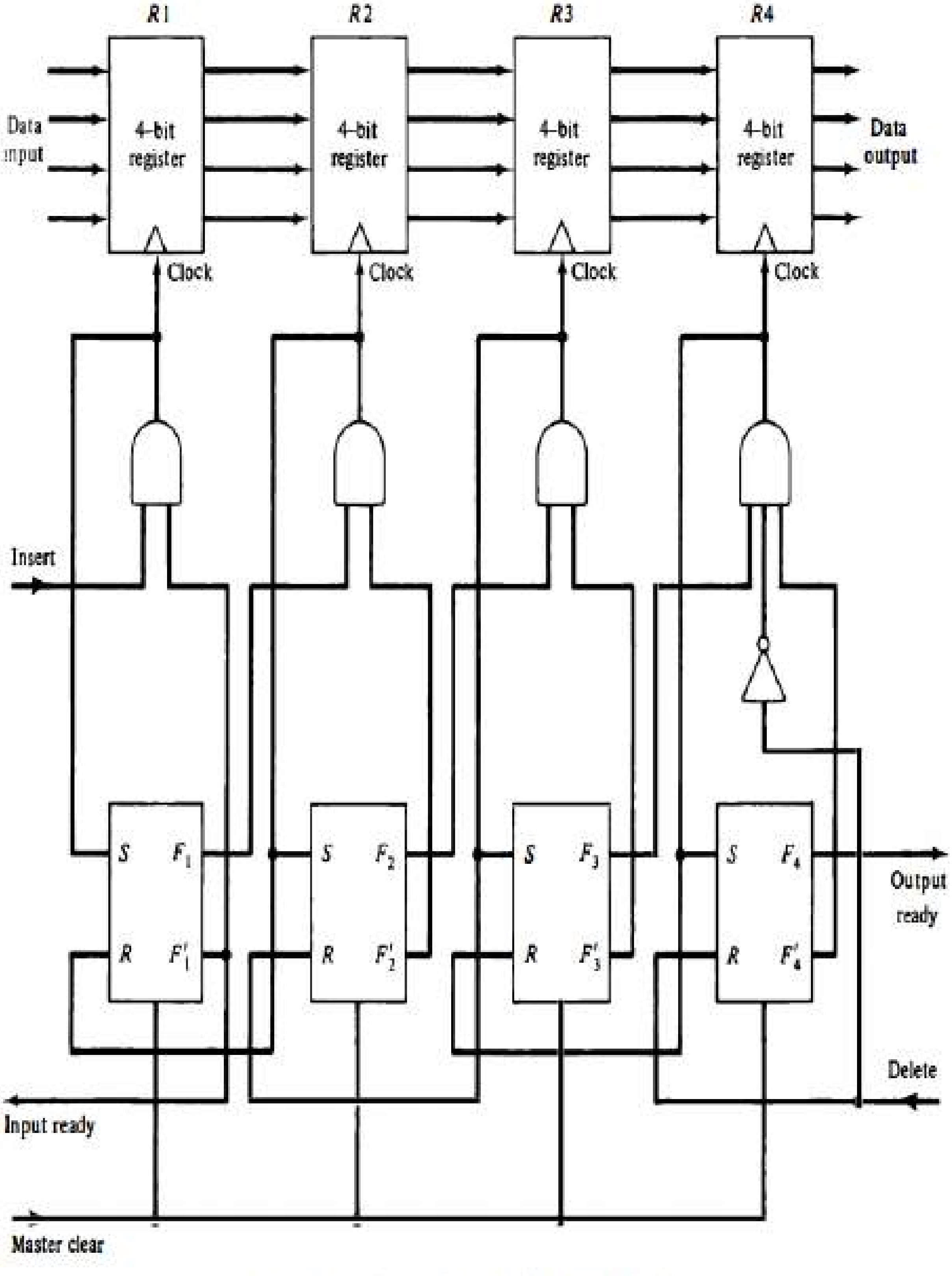


Figure 11-9 Circuit diagram of 4 × 4 FIFO buffer.

→When placed between two units, the FIFO can accept data from the source unit at one rate of transfer and deliver the data to the destination unit at another rate.
→If the source is slower than the destination unit, the buffer can be filled with data at a slow rate and later emptied at the higher rate.
→When placed between two units, the FIFO can accept data from the source unit at one rate of transfer and deliver the data to the destination unit at another rate.
→If the source is faster than the destination, the FIFO is useful for source data arrive in bursts that fills out the buffer. FIFO is useful in some applications when data are transferred asynchronously.

→The logic diagram of a typical 4 x 4 FIFO buffer is shown **Figure. 11-9**. It consists of four 4-bit registers RI, I=1,2,3,4 and a control register with flip-flops F_i , i=1,2,3,4, one for each register.FIFO can store four words of four bits each.
→A flip-flop F_i in the control register that is set to 1 indicates that a 4-bit data word is stored in the corresponding register RI. A 0 in F_i indicates that the corresponding register does not contain valid data. The control register directs the movement of data through the registers.
→Whether the F_i bit of the control register is set (F_i=1) and the bit is reset (F_{i+1}=1), a clock register is generated causing register R(I+1) to accept the data from the register RI. The same clock transition sets F_{i+1} to 1 and resets F_i to 0. This causes the control flag to move one position to the right together with the data.

10. Modes of Transfer:

→The data transfer can be handled by various modes. Some of the modes use CPU as an intermediate path, others transfer the data directly to and from the memory unit and this can be handled by 3 following ways:

A). Programmed I/O
B). Interrupt-initiated I/O
C). Direct memory access (DMA)

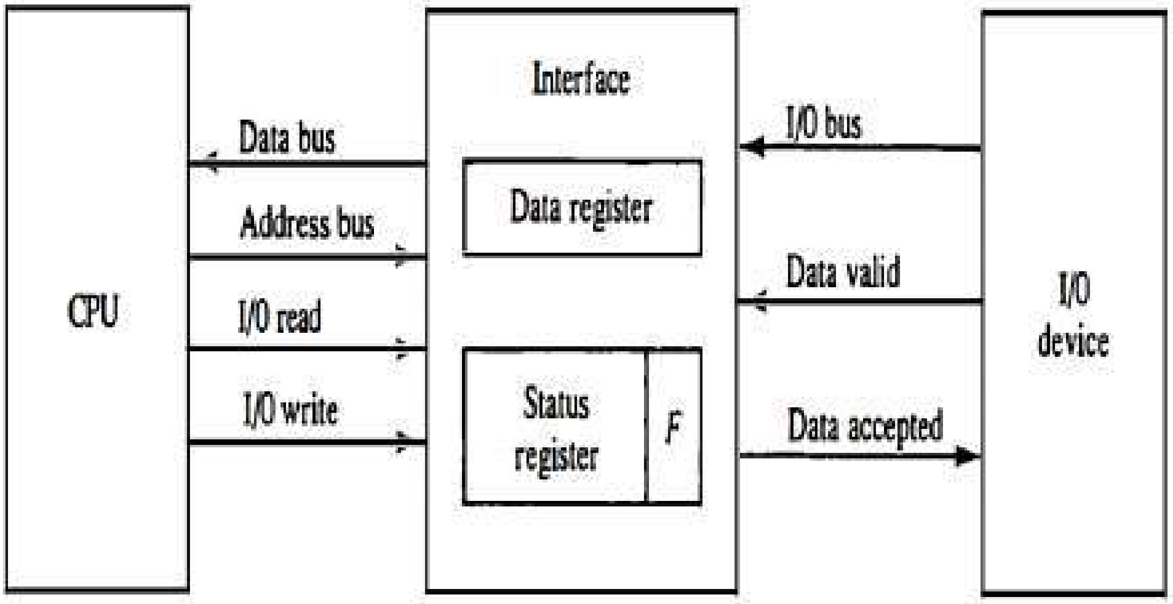
(A).Programmed I/O:

→Programmed 1/0 operations are the result of I/O instructions written in the computer program. Each data item transfer is initiated by an instruction in the program. Usually, the transfer is to and from a CPU register and peripheral. Other instructions are needed to transfer the data to and from CPU and memory. Transferring data under program control requires constant monitoring of the peripheral by the CPU. Once a data transfer is initiated, the CPU is required to monitor the interface to see when a transfer can again be made. It is up to the programmed instructions executed in the CPU to keep close tabs on everything that is taking place in the interface unit and the I/O device.

→In the programmed I/O method, the I/O device does 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. Other instructions may be needed to verify that the data are available from the device and to count the numbers of words transferred.

→An example of data transfer from an I/O device through an interface into the CPU is shown in **Figure 11-10.**

Figure 11-10 Data transfer from I/O device to CPU.



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 bit in the status register that 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. This is according to the handshaking procedure established.

→If the flag is equal to 1, the CPU reads the data from the data register. The flag bit is then cleared to 0 by either the CPU or the interface, depending on how the interface circuits are designed. Once the flag is cleared, the interface disables the data accepted line and the device can then transfer the next data byte.

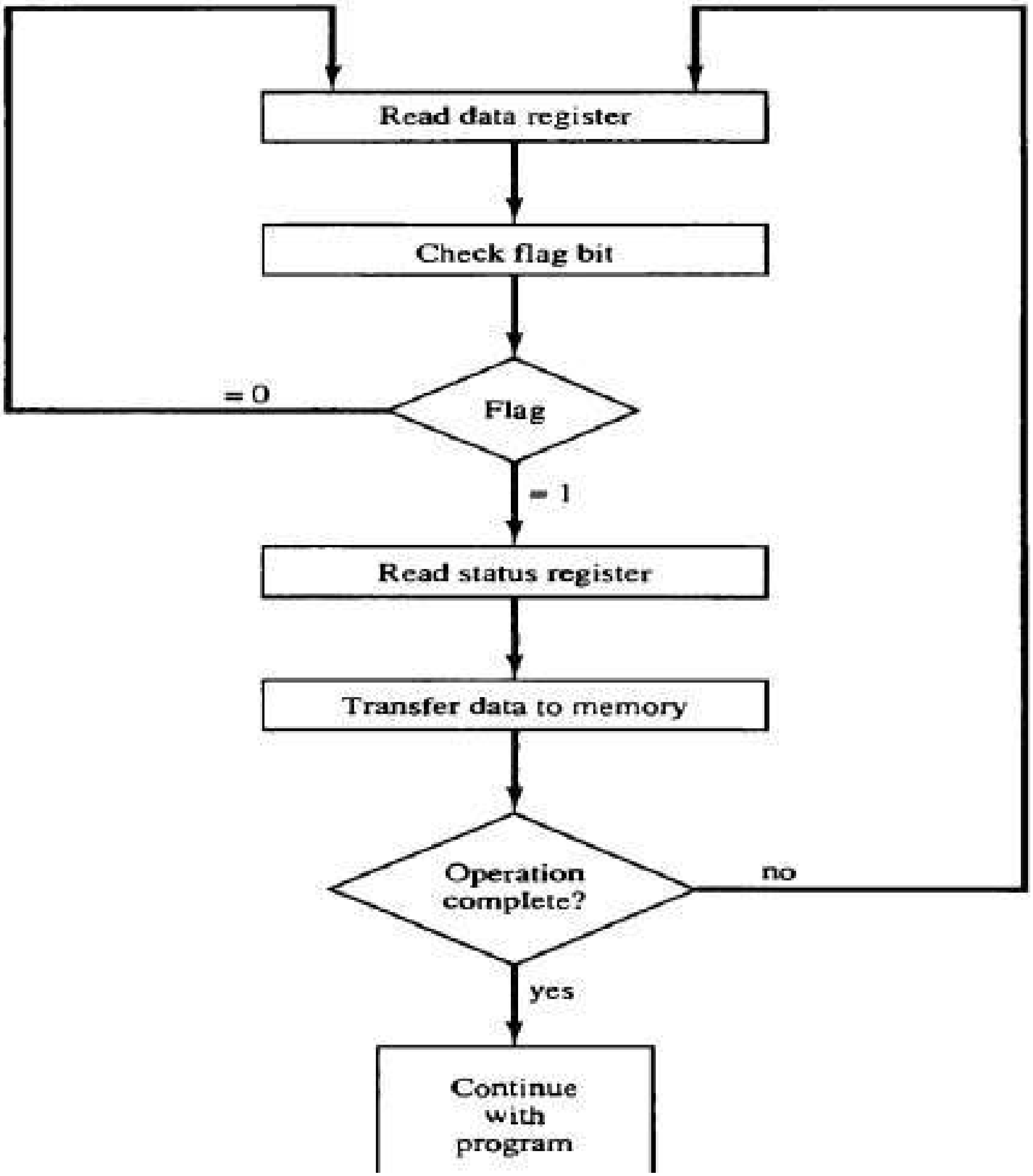


Figure 11-11 Flow chart for CPU program to input data.

→A flow chart of the program that must be written for the CPU is shown in **Figure. 11-11**. 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:

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

- Read the data register

Drawback of the Programmed I/O:

→The main drawback of the Program Initiated I/O was that the CPU has to monitor the units all the times when the program is executing. Thus the CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. This is a time consuming process and the CPU time is wasted a lot in keeping an eye to the executing of program.

→To remove this problem an Interrupt facility and special commands are used.

(B) Interrupt-Initiated I/O:

→In the programmed I/O method, the CPU stays in a program loop until the I/O unit indicates that it is ready for data transfer. This is a time-consuming process since it keeps the processor busy needlessly. It can be avoided by using an **interrupt** facility and special commands to inform the interface to issue an interrupt request signal when the data are available from the device. In the meantime the CPU can proceed to execute another program. The interface meanwhile keeps monitoring the device. 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.

→An alternative to the CPU constantly monitoring the flag is to let the interface inform the computer when it is ready to transfer data. This mode of transfer uses the interrupt facility. While the CPU is running a program it does not check the flag. However, when the flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that the flag has been set. The CPU deviates from what it is doing to take care of the input or output transfer. After the transfer is completed, the computer returns to the previous program to continue what it was doing before the interrupt.

→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.

→For each interrupt there is service routine, service routine address must be known by CPU to branch to it. This is accomplished by two methods:

1. **Vector Interrupt.**
2. **Non-vector Interrupt.**

Non-vectored interrupt:

→In a **non-vectored interrupt**, the branch address is assigned to a fixed location in memory. Vectored interrupt: the source that interrupts supplies the branch information to the computer. This information is called the **interrupt vector.** In some computers the interrupt vector is the first address of the I/O service routine. In other computers the interrupt vector is an address that points to a location in memory where the beginning address of the I/O service routine is stored.

→There is possibility that several sources will request service simultaneously; in this case the system must also decide which device to service first.

11. PRIORITY INTERRUPT:

→A **priority interrupt** is a system that establishes a priority over the various sources to determine which condition is to be serviced first when two or requests arrive simultaneously. The system may also determine which conditions are permitted to interrupt the computer while another interrupt is being serviced.

→Establishing the priority of simultaneous interrupts can be done by **software** or **hardware**.

→Software priority interrupts:

→A polling procedure is used to identify the interrupt source having highest-priority. Only one branch address is used for all interrupts. The priority of each interrupt source determines the order in which it is polled. The source with the highest priority is tested first, and if its interrupt signal is on, control branches to a routine that services that source. Otherwise, the source with the next lower priority is tested, and so on.

→Thus the initial service routine for all interrupts consists of a program that tests the interrupt sources in sequence and branches to one of many possible service routines. The disadvantage of the software 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.

→ Hardware priority interrupts:

→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**.

Daisy-Chaining Priority:

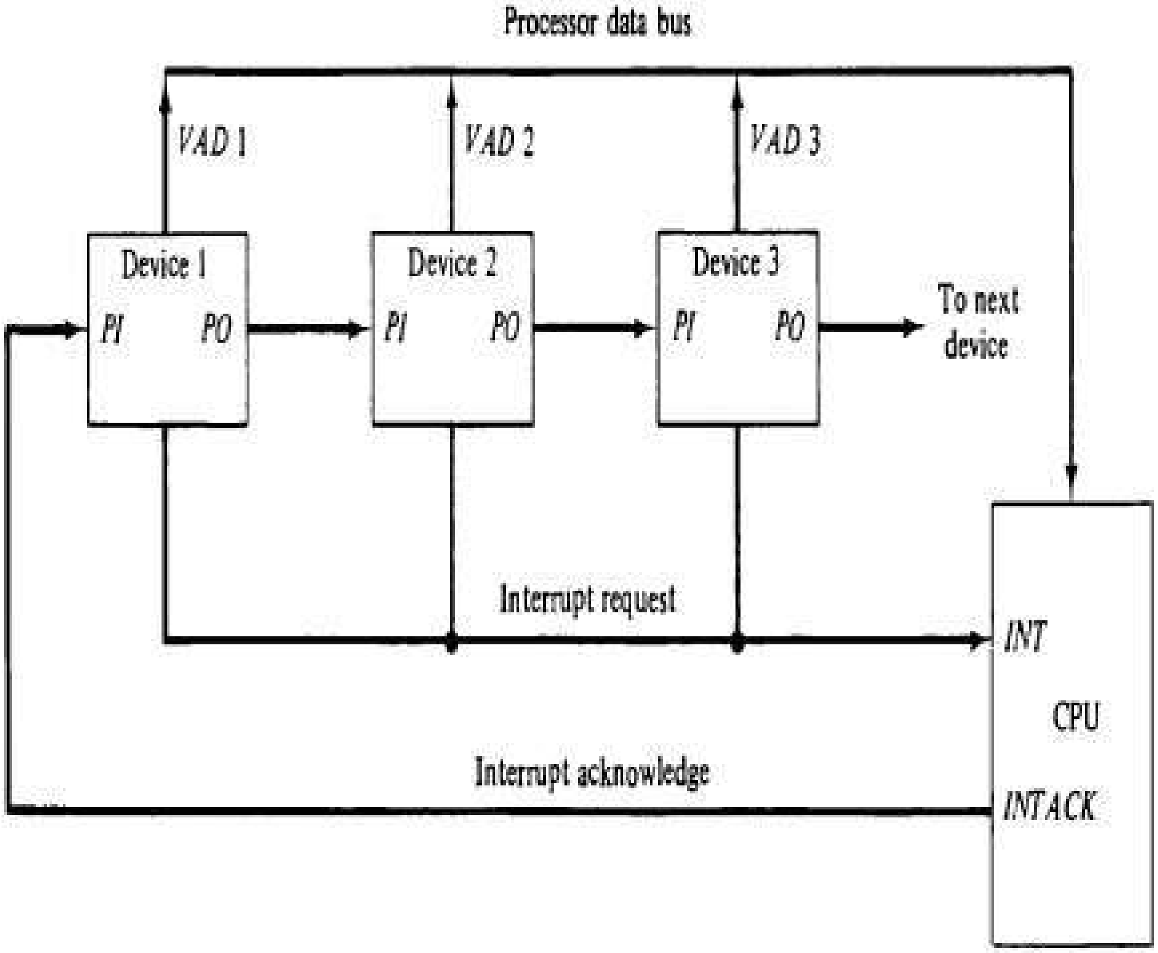


Figure 11-12 Daisy-chain priority interrupt.

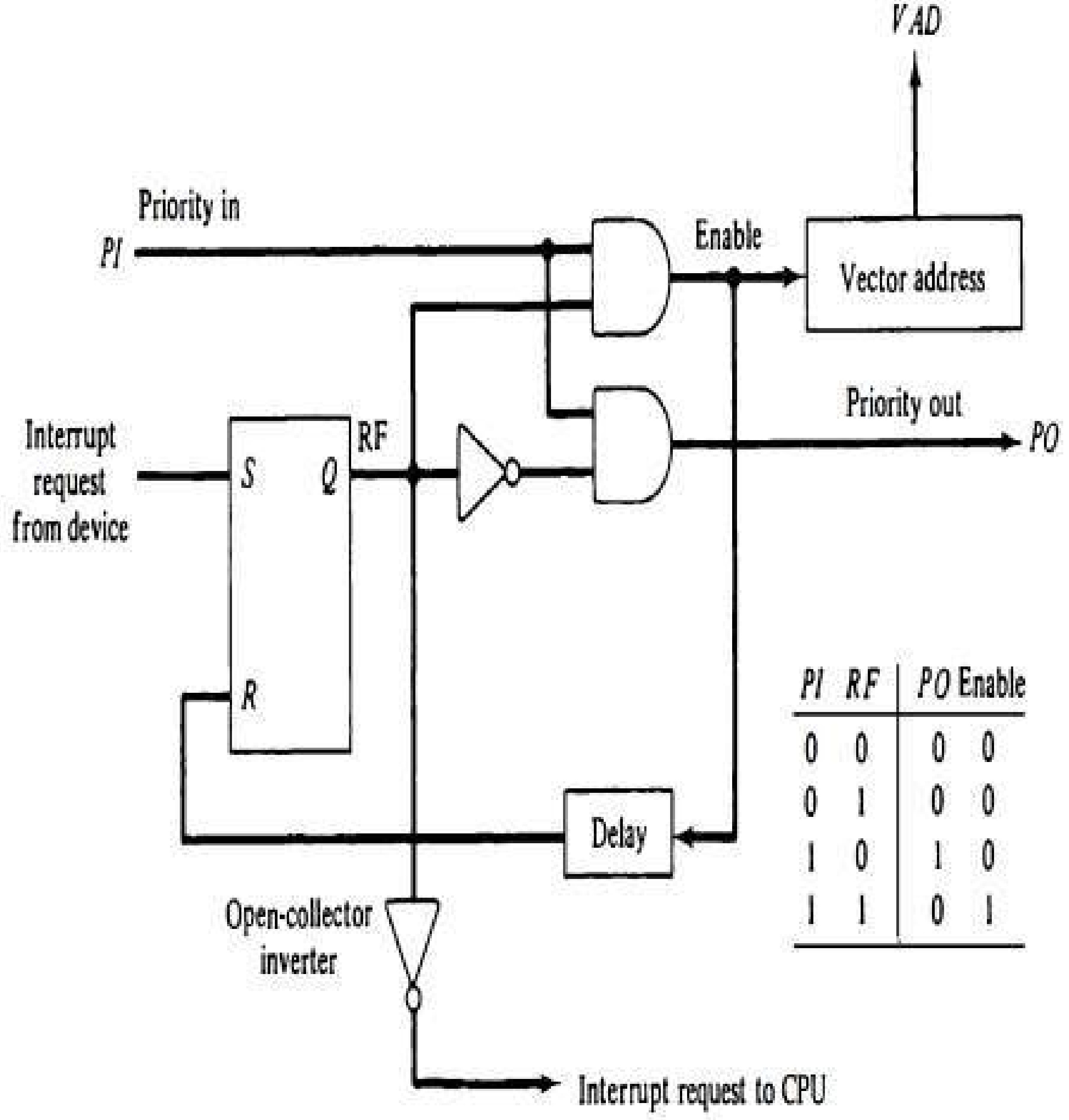
- The daisy-chaining method of establishing priority consists of serial connection of all devices that request an interrupt. The device with the highest priority is placed in the first position, followed by lower-priority devices up to the device with the lowest priority, which is placed last in the chain and closest to the CPU. This method of connection between three devices and the CPU is shown in **Figure. 11-12.**

- 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 in the CPU. When no interrupts are pending, the interrupt line says in the high-level state and no interrupts are recognized by the CPU. This is equivalent to negative logic OR operation.
- The CPU responds to an interrupt request by enabling the interrupt acknowledge line. 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 1 has a pending interrupt, it blocks the acknowledge signal from the next device by placing 0 in the PO output.
- If then proceeds to inserts its own interrupt **vector address (VAD)**into the data bus for the CPU touse during the interrupt cycle.

→A device with a 0 on its PI input generates a 0 on its P0 output to inform the device with the next lower priority that the acknowledge signal has been blocked. A device that is requesting an interrupt and has a 1 on its PI input will intercept the acknowledge signal by placing a 0 on its PO output. If the device does not have pending interrupts, it transmits the acknowledge signal to the next device by placing a 1 on its PO output.

→Thus the device with PI=1 and PO=0 is one with the highest priority that is requesting an interrupt, and the device places its VAD on the data bus. The daisy chain arrangement gives the highest priority to the device that receives that interrupt acknowledge signal from the CPU. The farther the device is from the first position; the lower is its priority.

→**Figure 11-13** shows the internal logic that must be included with each device when connected to the daisy-chain scheme.



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

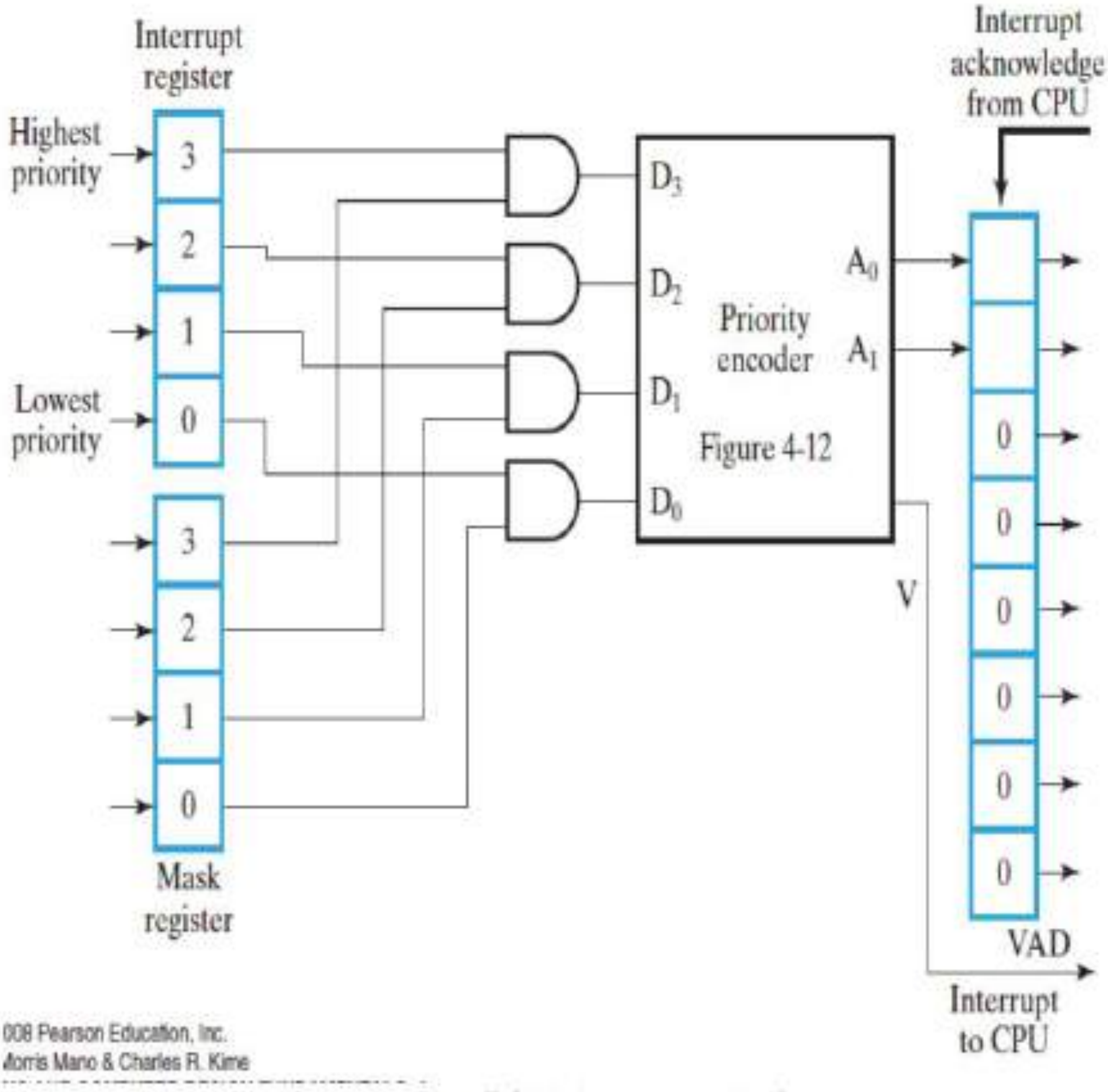
Figure 11-13 One stage of the daisy-chain priority arrangement.

→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.

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 **Figure. 11-14**.



008 Pearson Education, Inc.
Aorris Mano & Charles R. Kime

Figure 12-17 Parallel Priority Interrupt Hardware

→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. Each interrupt bit and its corresponding mask bit are applied to an AND gate to produce the four inputs to a priority encoder. In this way an interrupt is recognized only if its corresponding mask bit is set to 1 by the program. The priority encoder generates two bits of the vector address, which is transferred to the CPU.

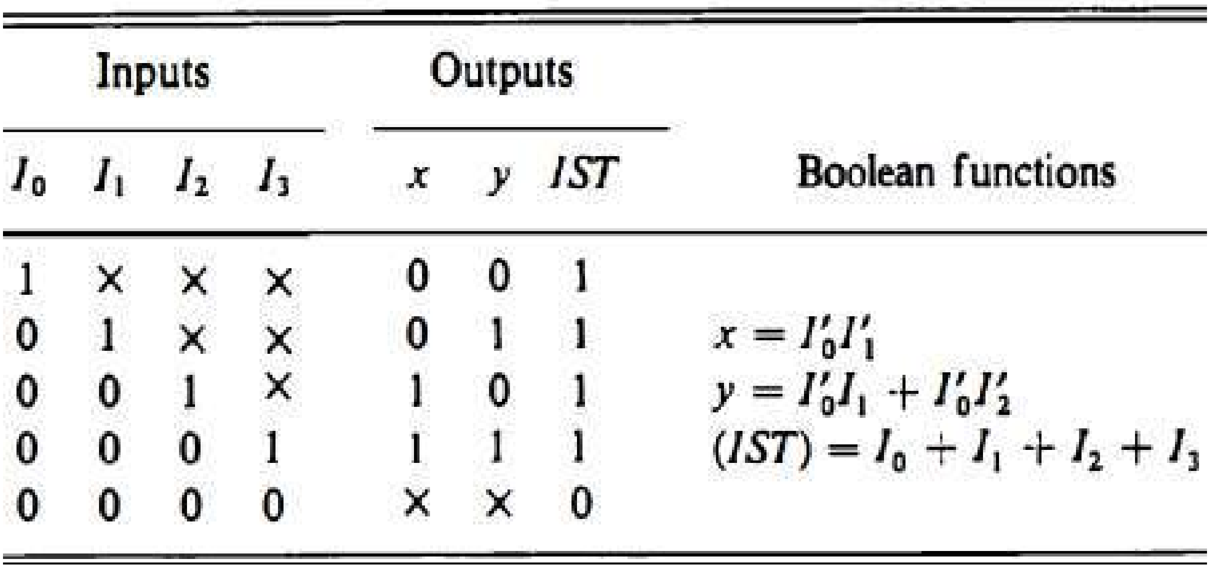
→Another output from the encoder sets an interrupt status flip-flop IST when an interrupt that is not masked occurs. The interrupt enable flip-flop IEN can be set or cleared by the program to provide an overall control over the interrupt system. The outputs of IST ANDed with IEN provide a common interrupt signal for the CPU. The interrupt acknowledge INTACK signal from the CPU enables the bus buffers in the output register and a vector address VAD is placed into the data bus. We will now explain the priority encoder circuit and then discuss the interaction between the priority interrupt controller and the CPU.

Priority Encoder:

→The priority encoder is a circuit that implements the priority function. The logic of the priority encoder is such that if two or more inputs arrive at the same time, the input having the highest priority will take precedence.

→The truth table of a four input priority encoder is given in **Table 11·2**.

TABLE 11-2 Priority Encoder Truth Table



Inputs				Outputs			
I_0	I_1	I_2	I_3	x	y	IST	Boolean functions
1	×	×	×	0	0	1	
0	1	×	×	0	1	1	$x = I_0'I_1'$
0	0	1	×	1	0	1	$y = I_0'I_1 + I_0'I_2'$
0	0	0	1	1	1	1	$(IST) = I_0 + I_1 + I_2 + I_3$
0	0	0	0	×	×	0	

→The x's in the table designate don't-care conditions. Input I_0 has the highest priority; so regardless of the values of other inputs, when this input is 1, the output generates an output xy : 00. I_2 has the next priority level. The output is 01 if I_1 =1 provided that I_0 = 0, regardless of the values of the other two lower-priority inputs. The output for I_2 is generated only if higher-priority inputs are 0, and so on down the priority level.

→The interrupt status IST is set only when one or more inputs are equal to 1. If all inputs are 0, IST is cleared to 0 and the other outputs of the encoder are not used, so they are marked with don't-care conditions. This is because the vector address is not transferred to the CPU when IST = 0.

Interrupt Cycle:

→The interrupt enable flip-flop IEN shown in **Figure. 11-14** can be set or cleared by program instructions. When IEN is cleared, the interrupt request coming from IST is neglected by the CPU. The program-controlled IEN bit allows the programmer to choose whether to use the interrupt facility. An instruction to set IEN indicates that the interrupt facility will be used while the current program is running.

→At the end of each instruction cycle the CPU checks IEN and the interrupt signal from IST. If either is equal to 0, control continues with the next instruction. If both IEN and IST are equal to 1, the CPU goes to an interrupt cycle. During the interrupt cycle the CPU performs the following sequence of micro-operations:

SP ← SP - 1	**Decrement stack pointer**
M [SP] ← PC	**Push PC into stack**
NTACK ← 1	**Enable interrupt acknowledge**
PC ← VAD	**Transfer vector address to PC**
IEN ← 0	**Disable further interrupts**

Go to fetch next instruction

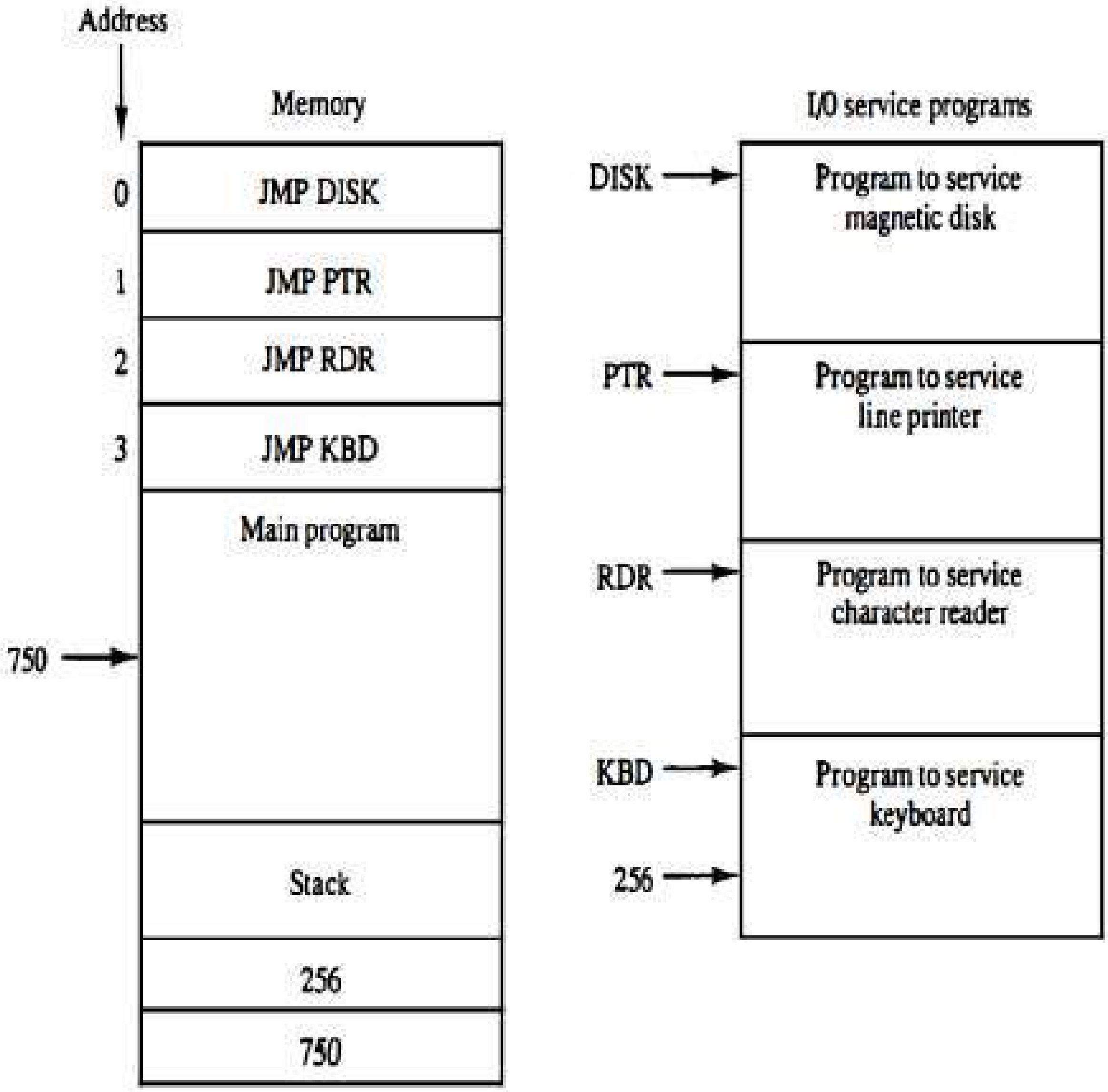
→The CPU pushes the return address from PC into the stack. It then acknowledges the interrupt by enabling the INTACK line. The priority interrupt unit responds by placing a unique interrupt vector into the CPU data bus. The CPU transfers the vector address into PC and clears IEN prior to going to the next fetch phase. The instruction read from memory during the next fetch phase will be the one located at the vector address.

Software Routines:

→A priority interrupt system is a combination of hardware and software techniques. The computer must also have *software routines* for servicing the interrupt requests and for controlling the interrupt hardware registers.

→**Figure 11-15** shows the programs that must reside in memory for handling the interrupt system. Each device has its own service program that can be reached through a jump GMP) instruction stored at the assigned vector address. The symbolic name of each routine represents the starting address of the service program. The stack shown in the diagram is used for storing the return address after each interrupt.

Figure 11-15 Programs stored in memory for servicing interrupts.


Address
Memory
0
JMP DISK
1
JMP PTR
2
JMP RDR
3
JMP KBD
Main program
750
Stack
256
750
I/O service programs
DISK
Program to service magnetic disk
PTR
Program to service line printer
RDR
Program to service character reader
KBD
Program to service keyboard
256

→To illustrate with a specific example assume that the keyboard sets its interrupt bit while the CPU is executing the instruction in location 749 of the main program. At the end of the instruction cycle, the computer goes to an interrupt cycle. It stores the return address 750 in the stack and then accepts the vector address 00000011 from the bus and transfers it to PC. The instruction in location 3 is executed next, resulting in transfer of control to the KBD routine. Now suppose that the disk sets its interrupt bit when the CPU is executing the instruction at address 255 in the KBD program. Address 256 is pushed into the stack and control is transferred to the DISK service program. The last instruction in each routine is a return from interrupt instruction. When the disk service program is completed, the return instruction pops the stack and places 256 into PC. This returns control to the KBD routine to continue servicing the keyboard. At the end of the KBD program, the last instruction pops the stack and returns control to the main program at address 750.

Initial and Final Operations:

→The initial sequence of each interrupt service routine must have instructions to control the interrupt hardware in the following manner:

1. Clear lower-level mask register bits.
2. Clear interrupt status bit IST.
3. Save contents of processor registers.
4. Set interrupt enable bit IEN.
5. Proceed with service routine.

→The final sequence in each interrupt service routine must have instructions to control the interrupt hardware in the following manner:

1. Gear interrupt enable bit IEN.
2. Restore contents of processor registers.
3. Clear the bit in the interrupt register belonging to the source that has been serviced.
4. Set lower-level priority bits in the mask register.
5. Restore return address into PC and set IEN.

12. 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.

.

→**Figure 11-6** shows two control signals in the CPU that facilitate the DMA transfer.

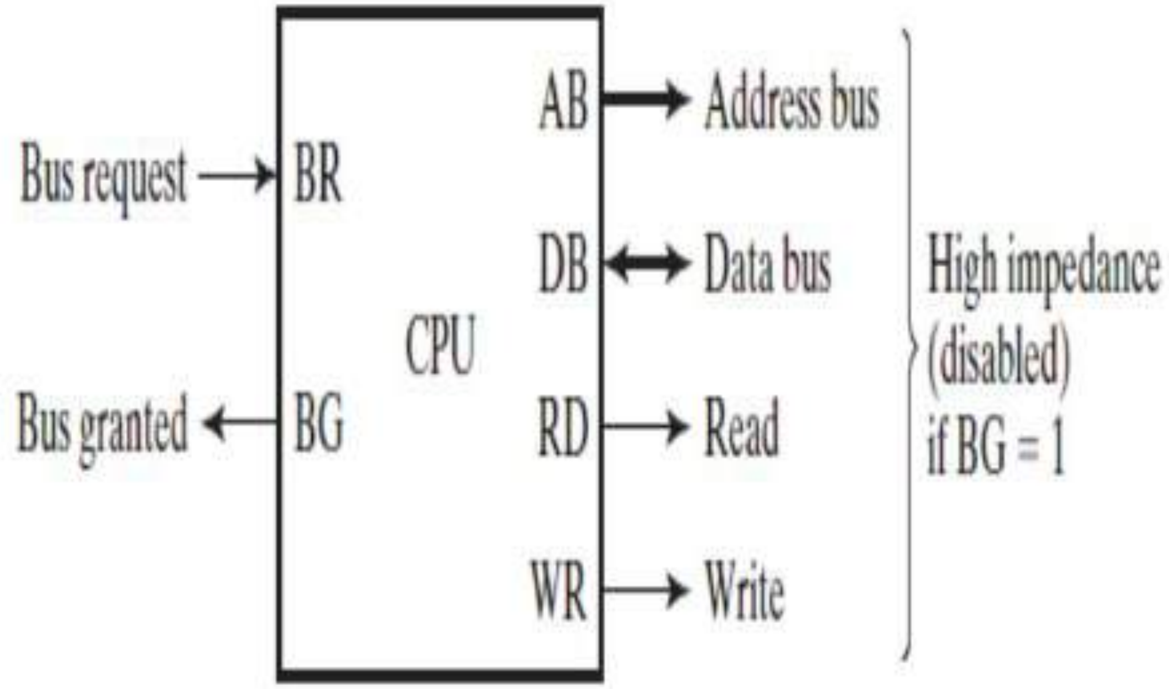


Figure 11-16 CPU bus signals for DMA transfer.

→The **bus request (BR)**inputis used by the DMA controller to request the CPU to relinquish control of the buses. When BR input is asserted, the CPU places the address bus, the data bus, and the read and write-lines into a high-impedance state.

→ Then, the CPU asserts the **bus granted (BG)** output to inform the external DMA that it can take control of the buses. As long as the BG line is asserted, the CPU is unable to proceed with any operations requiring access to the buses.

→When the BR, bus request, signal is reset by the DMA, the CPU returns to its normal operation, resets the BG, bus granted, signal, and resumes control of the buses.

→When the BG line is asserted, the external DMA controller takes control of the bus system in order to communicate directly with memory. The transfer can be made for an entire block of memory words, suspending operation of the CPU until the entire block is transferred, a process referred to as **burst transfer**.

→Alternatively, the transfer can be made one word at a time between executions of CPU instructions, a process called single cycle transfer or **cycle stealing**.

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, a set of address lines.
→**Address Register**: Used to communicate directly with memory Incremented after each word is transferred to memory.
→**Word-Count Register**: Specifies the number of words to transfer Decremented after each word is transferred to memory.
→**Control Register:** Specifies the mode of transfer Modes – read from memory, or write to memory.

→**Figure 11-17** shows the block diagram of a typical DMA controller.

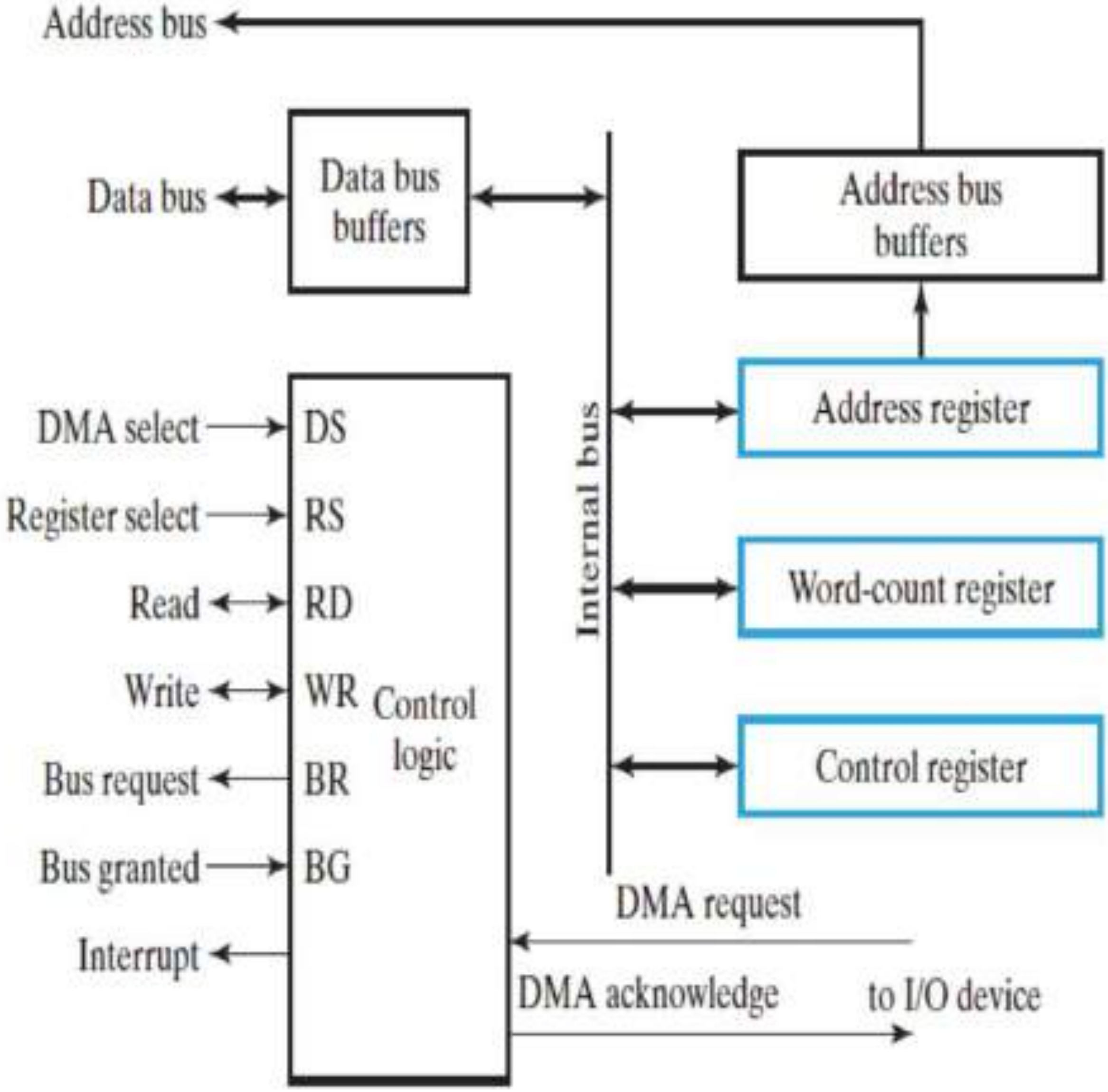


Figure 11-17 Block diagram of DMA controller.

→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 Granted) input is 0, the CPU can communicate with the DMA registers through the data bus to read from or write to those registers.
→When BG=1, the CPU has relinquished the buses, and the DMA can communicate directly with memory by specifying an address on the address bus and activating the RD or WR control.
→The DMA communicates with the external peripheral through the DMA request and DMA acknowledge lines by a prescribed handshaking procedure.

→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 blocks 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.

DMA Transfer:

→The position of the DMA controller among the other components in a computer system is illustrated in **Figure 11-8.**

→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 (DMA Select) and RS (Register Select) lines. The CPU initializes the DMA through the data bus. Once the DMA receives the start control bit, it can begin transferring data between the peripheral device and memory.

→When the peripheral device sends a DMA request, the DMA controller activates the BR (Bus Request) line, informing the CPU that it is to relinquish the buses. • The CPU responds with it BG (Bus Granted) line, informing the DMA that the buses are disabled. • The DMA then puts the current value of its address register onto the address bus, initiates the RD or WR signal, and sends a DMA acknowledge to the peripheral device.

→When the peripheral device receives a DMA acknowledge, it puts a word on the data bus (for writing) or receives a word from the data bus (for reading). • The DMA controls the read or write operation and supplies the address for memory. The peripheral unit can then communicate with memory through the data bus for a direct transfer of data between the two units while the CPU access to the data bus is momentarily disabled.

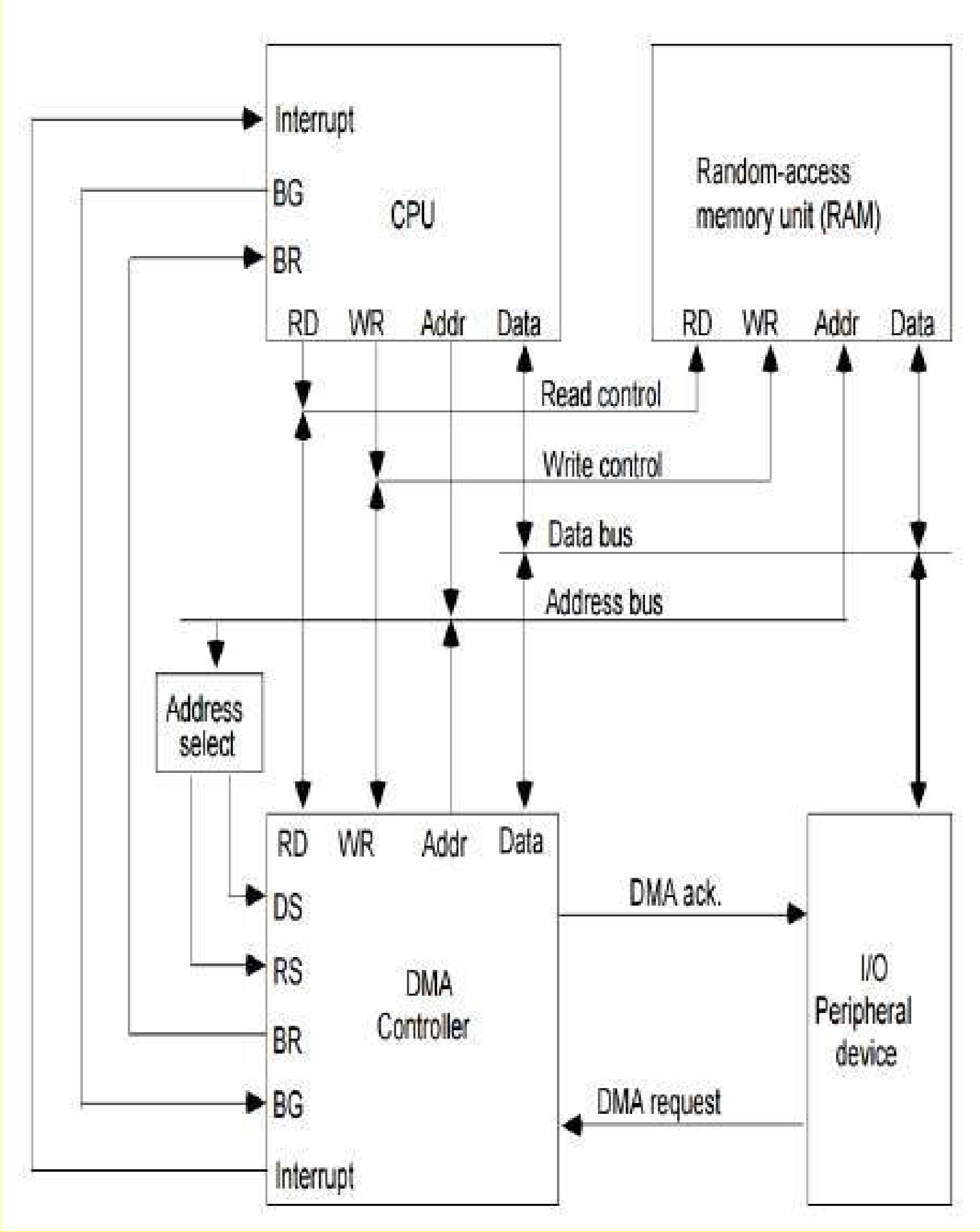


Figure 11-8 DMA transfer in computer system

→For each word that is transferred, the DMA increments its address registers and decrements its word count-register. If the word count does not reach zero, the DMA checks the request line coming from the peripheral. For a high-speed device, the line will be active, as soon as the previous transfer is completed. A second transfer is then initiated, and the process continues until the entire block is transferred. If the peripheral speed is slower, the DMA request line may come somewhat later. In this case the DMA disable the bus request line so that the CPU can continue to execute its program. When the peripheral requests a transfer, the DMA requests the buses again.

→If the word count register reaches zero, the DMA stops any further transfer and removes its bus request. It also informs the CPU of the termination by means of an interrupt. When the CPU responds to the interrupt, it reads the content of the word count-register. The zero value of this register indicates that all the words were transferred successfully.

→DMA transfer is very useful in many applications. It is used for fast transfer of information between magnetic disks and memory.

IMPORTANT QUESTIONS (UNIT-4)

1. Explain in detail how a memory is connected to the CPU? (14M)
2. Describe the block diagram of associative memory?
3. Explain address mapping using pages?
4. Explain different cache memory address mapping techniques?(14M)
5. Explain paging scheme in detail? (14M)
6. Explain associative memory?
7. Explain main memory?
8. Discuss the direct mapping procedure from main memory to cache memory?
9. (a) Describe the concept of cache memory. What are the mapping schemes adopted? Explain.

   (b). Explain the need for auxiliary memory devices. How are they different from main memory and from other peripheral devices?

   (c) What are the virtual memories and how are they implemented? (14M)
10. A computer uses RAM chips or 1024 x I capacity.

    (i)How many chips are needed, and how should their address lines be connected to provide a memory capacity of 1024 bytes?

    (ii)How many chips are needed to provide a memory capacity or 16K bytes? (14M)
11. Explain commonly used memory replacement algorithm in virtual memory system and their implementation?

12. Explain memory management hardware?

13. Distinguish between Isolated I/O and memory mapped I/O? What are the advantages and disadvantages of each?

14. Explain in detail about DMA?

15. Write about Asynchronous Data transfer operation.

16. Explain DMA transfer in a computer system?

17. Describe I/O interface?

18. Explain Priority Interrupt?

19. Describe the data transfer procedure using handshaking?

20. Explain in detail about programmed I/O and Interrupt- Initiated I/O?

21. What is daisy chaining? Explain with neat sketch?

22. Explain I/O versus Memory Bus?

23. Explain about IO Processor?

24. What are the various modes of transfer employed in a computer? Compare their advantages and disadvantages?

25. Explain with a block diagram, how an asynchronous transmitter and receiver communication with each other?

26. What are the advantages of handshaking over strobe control? Outline the steps in source initiated transfer using handshaking?

27. What is an interrupt? Explain parallel priority interrupt and daisy chain interrupt systems with examples?

SHORT QUESTIONS

1. Auxiliary memory.
2. List the different types of memories?
3. What is hit-miss ratio?
4. Draw the block diagram for ROM chip?
5. What is memory hierarchy?
6. Explain cache memory?
7. Draw the cell structure of associative memory?
8. What are pages and frames?
9. What is the purpose and functions of Bootstrap loader?
10. What is priority interrupt?

COMPUTER ORGANIZATION-UNIT5

Multi Processors: Introduction, Characteristics of Multiprocessors, Interconnection Structures, Inter Processor Arbitration.

Pipeline: Parallel Processing, Pipelining, Instruction Pipeline, RISC Pipeline, Array Processor.

---

1. Introduction:

→A multiprocessor system is an interconnection of two or more CPU's with memory and input-output equipment. The term "processor" in multiprocessor can mean either a central processing unit (CPU) or an input-output processor (lOP).

→Multiprocessors are classified as multiple instruction stream, multiple data stream (MIMD) systems. There are some similarities between Multiprocessor and Multicomputer systems since both support concurrent operations. However, there exists important distinction between a system with multiple computers and a system with multiple processors.

→Computers are interconnected with each other means of communication lines to form a *computer network*. The network consists of several autonomous computers that may or may not communicate with each other.

→A *multiprocessor system* is controlled by one operating system that provides interaction between processors and all the components of the system cooperate in the solution of a problem

→*VLSI circuit* technology has reduced the cost of the computers to such a low Level that the concept of applying multiple processors to meet system performance requirements has become an attractive design possibility.

Benefits of Multiprocessing:

1. Multiprocessing increases the reliability of the system so that a failure or error in one part has 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 functions of the disabled one.

2. Improved System performance. System derives high performance from the fact that computations can proceed in parallel in one of the two ways:

(a). Multiple independent jobs can be made to operate in parallel.

(b). A single job can be partitioned into multiple parallel tasks. This can be achieved in two ways:

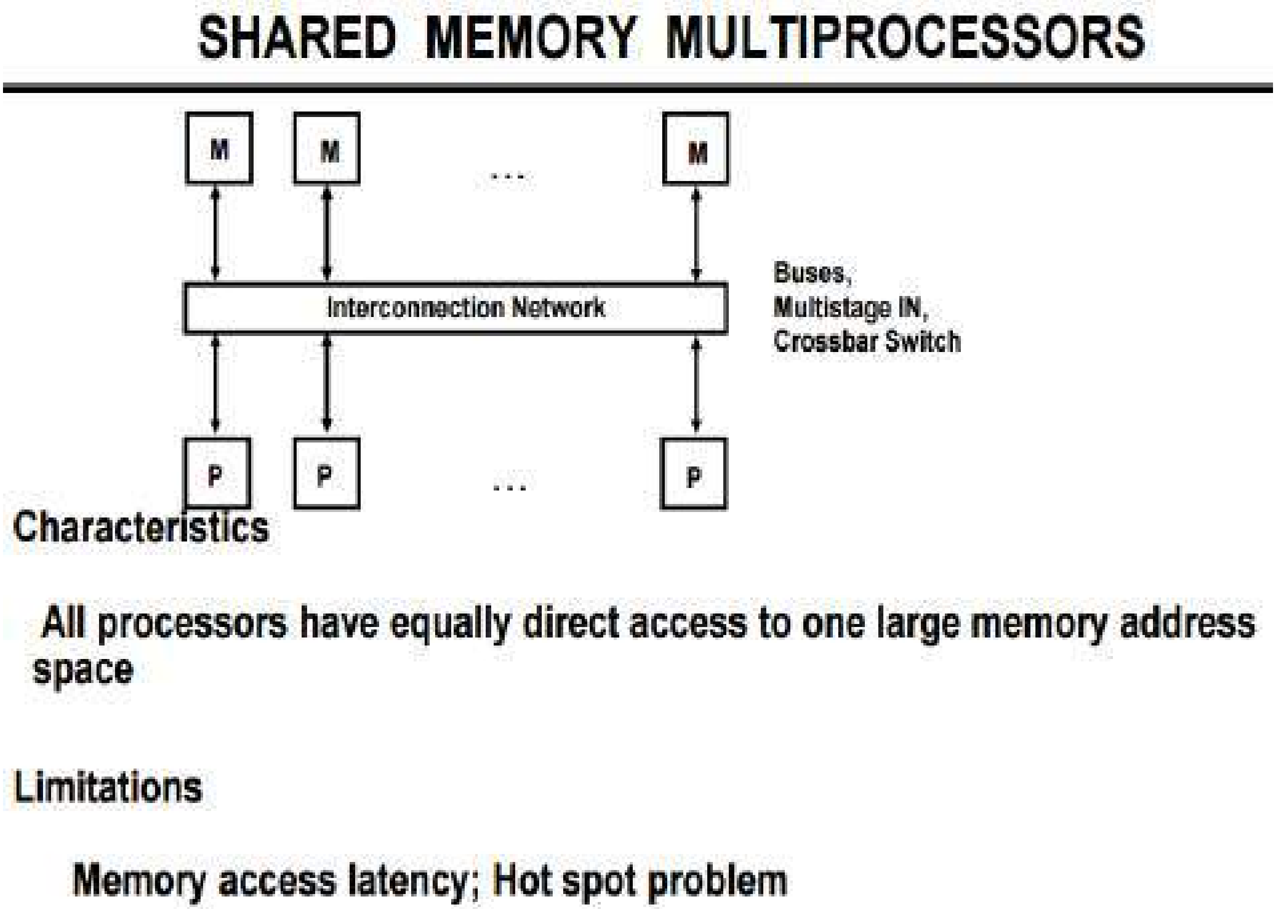
- The user explicitly declares that the tasks of the program be executed in parallel.
- The compiler provided with multiprocessor s/w that can automatically detect parallelism in program. Actually it checks for *Data dependency*.

2. Characteristics of Multiprocessors:

→Multiprocessors are classified by the way their memory is organized.

Tightly coupled micro-processor/Shared memory:

→A multiprocessor system with common shared memory is classified as a *shared memory or tightly coupled multiprocessor*. This does not preclude each processor from having its own local memory. In fact, most commercial tightly coupled multiprocessors provide a cache memory with each CPU. In addition, there is 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.



SHARED MEMORY MULTIPROCESSORS

Characteristics

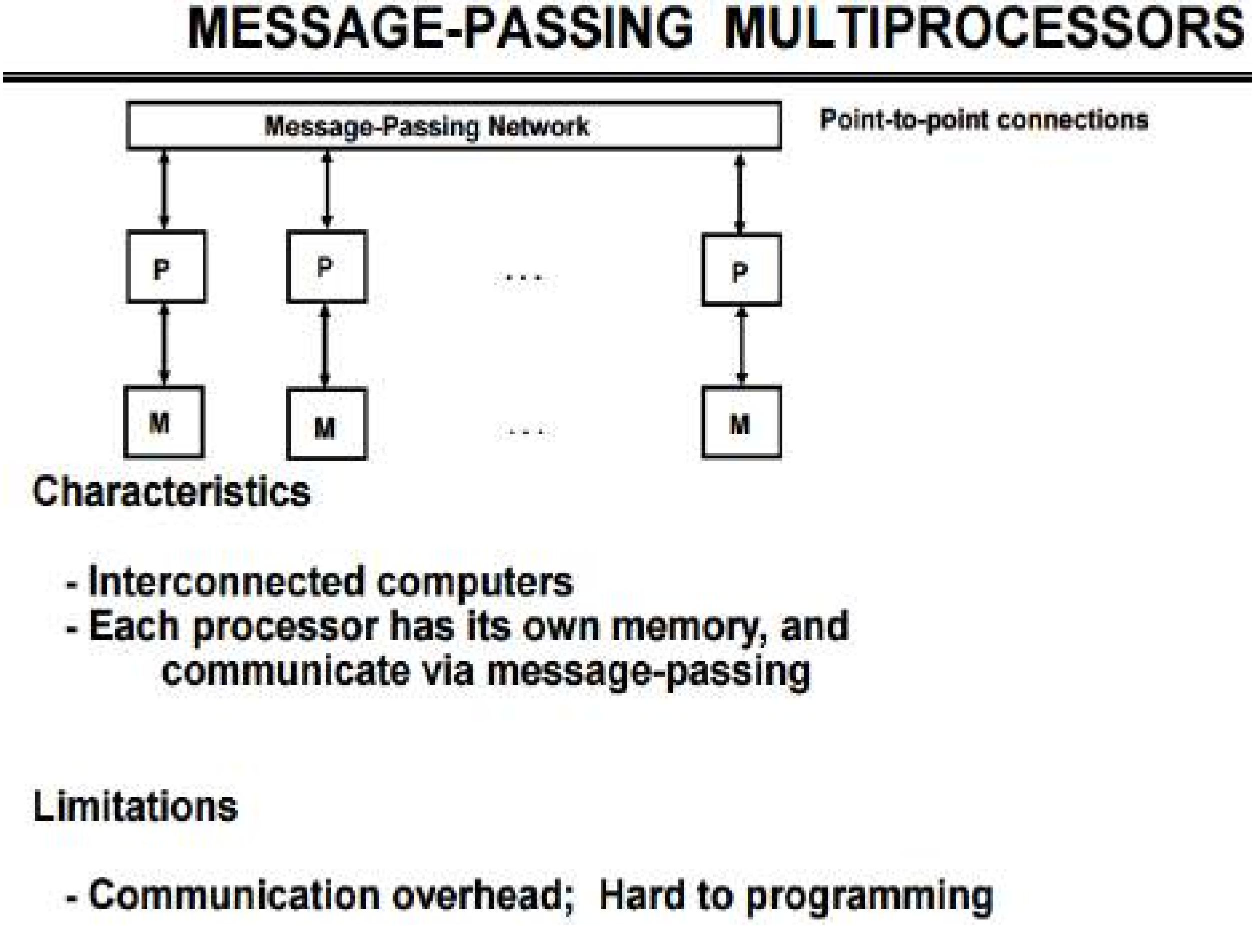
All processors have equally direct access to one large memory address space

Limitations

Memory access latency; Hot spot problem

Loosely coupled micro-processor/Distributed memory:

→An alternative model of microprocessor is the *distributed-memory or loosely coupled system.* Each processor element in a loosely coupled system has its own private local memory. The processors are tied together by a switching scheme designed to route information from one processor to another through a message-passing scheme. The processors relay program and data to other processors in packets. A packet consists of an address, the data content, and some error detection code. The packets are addressed to a specific processor or taken by the first available processor, depending on the communication system used. Loosely coupled systems are most efficient when the interaction between tasks is minimal, whereas tightly coupled systems can tolerate a higher degree of interaction between tasks.



MESSAGE-PASSING MULTIPROCESSORS

Point-to-point connections

Characteristics

- Interconnected computers
- Each processor has its own memory, and communicate via message-passing

Limitations

- Communication overhead; Hard to programming

→The principal characteristic of a multiprocessor is its ability to share a set of main memory and some I/O devices. This sharing is possible through some physical connections between them called the interconnection structures

3. Interconnection Structures:

→There are several physical forms available for establishing an inter connection network.

- **Time-shared common bus**
- **Multiport memory**
- **Crossbar switch**
- **Multistage switching network**
- **Hypercube system**

COMPUTER ORGANIZATION-UNIT5

Time-shared common bus:

→A common-bus multiprocessor system consists of a number of processorsconnected through a common path to a memory unit. A time-shared common bus for five processors is shown in **Fig. 13-1.**

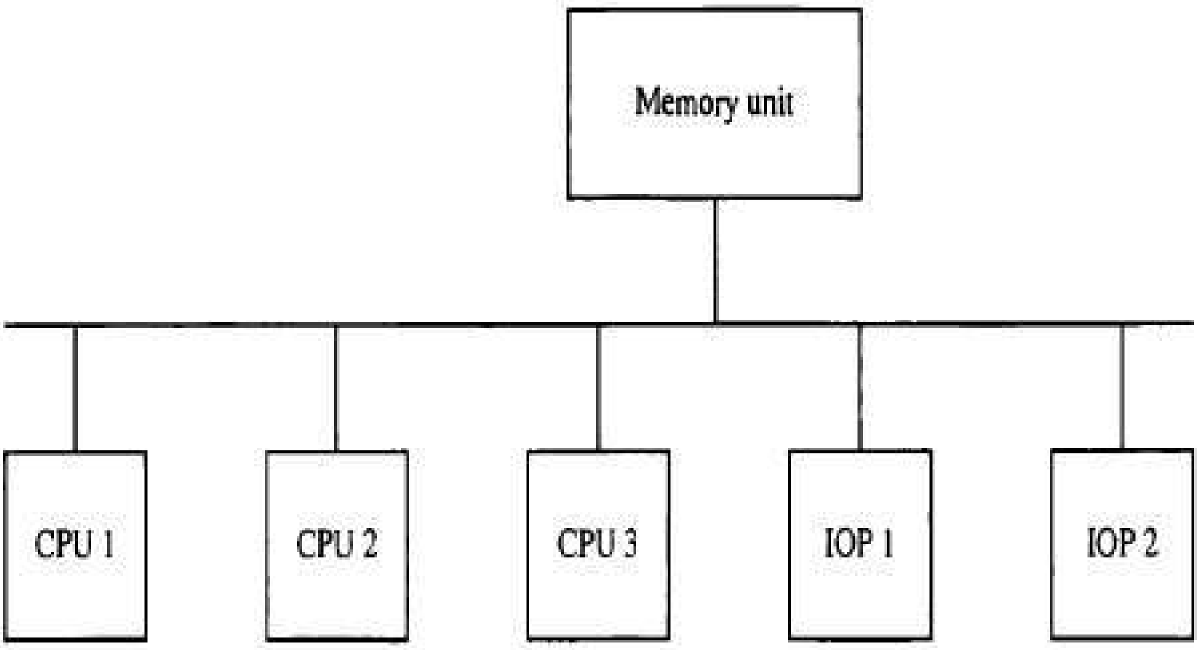


Figure 13-1 Time-shared common bus organization.

→Only one processor can communicate with the memory or another processor at any given time.

→A single common-bus system is restricted to one transfer at a time. This means that when one processor is communicating with the memory, all other processors are either busy with internal operations or must be idle waiting for the bus. As a consequence, the total overall transfer rate within the system is limited by the speed of the single path. The processors in the system can be kept busy more often through the implementation of two or more independent buses to permit multiple simultaneous bus transfers. However, this increases the system cost and complexity.

→A more economical implementation of a dual bus structure is depicted in **Fig. 13-2**.

→Here we have a number of local buses each connected to its own local memory and to one or more processors. Each local bus may be connected to a CPU, an IOP, or any combination of processors. A system bus controller links each local bus to a common system bus. The I/O devices connected to

the local IOP, as well as the local memory are available to the local processor. The memory connected to the common system bus is shared by all processors. If an IOP is connected directly to the system bus, the I/O devices attached to it may be made available to all 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.

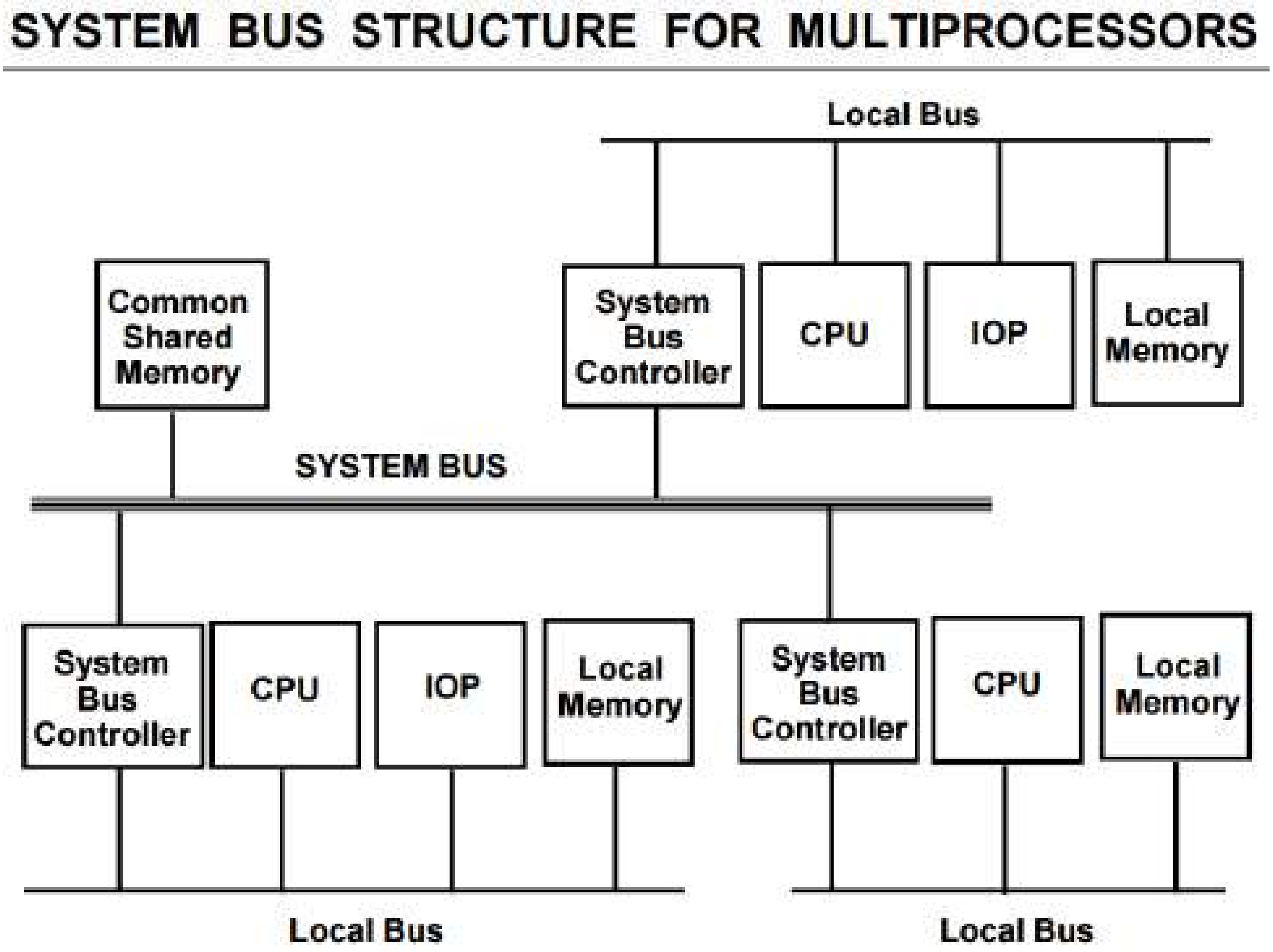


Figure 13-2 System Bus Structure for multiprocessors.

Multiport memory:

→A multiport memory system employs separate buses between each memory module and each CPU.

→This is shown in **Fig. 13-3** for four CPUs and four memory modules (MMs). Each processor bus is connected to each memory module. A processor bus consists of the address, data, and control lines required communicating with memory. The memory module is said to have four ports and each port accommodates one of the buses.

→The priority for memory access associated with each processor may be established by the physical port position that its bus occupies in each module. Thus CPU 1 will have priority over CPU 2, CPU 2will have priority over CPU 3, and CPU 4 will have the lowest priority.

→The advantage of the multiport memory organization is the high transfer rate that can be achieved because of the multiple paths between processors and memory.

→The disadvantage is that it requires expensive memory control logic and a large number of cables and connectors. As a consequence, this interconnection structure is usually appropriate for systems with a small number of processors.

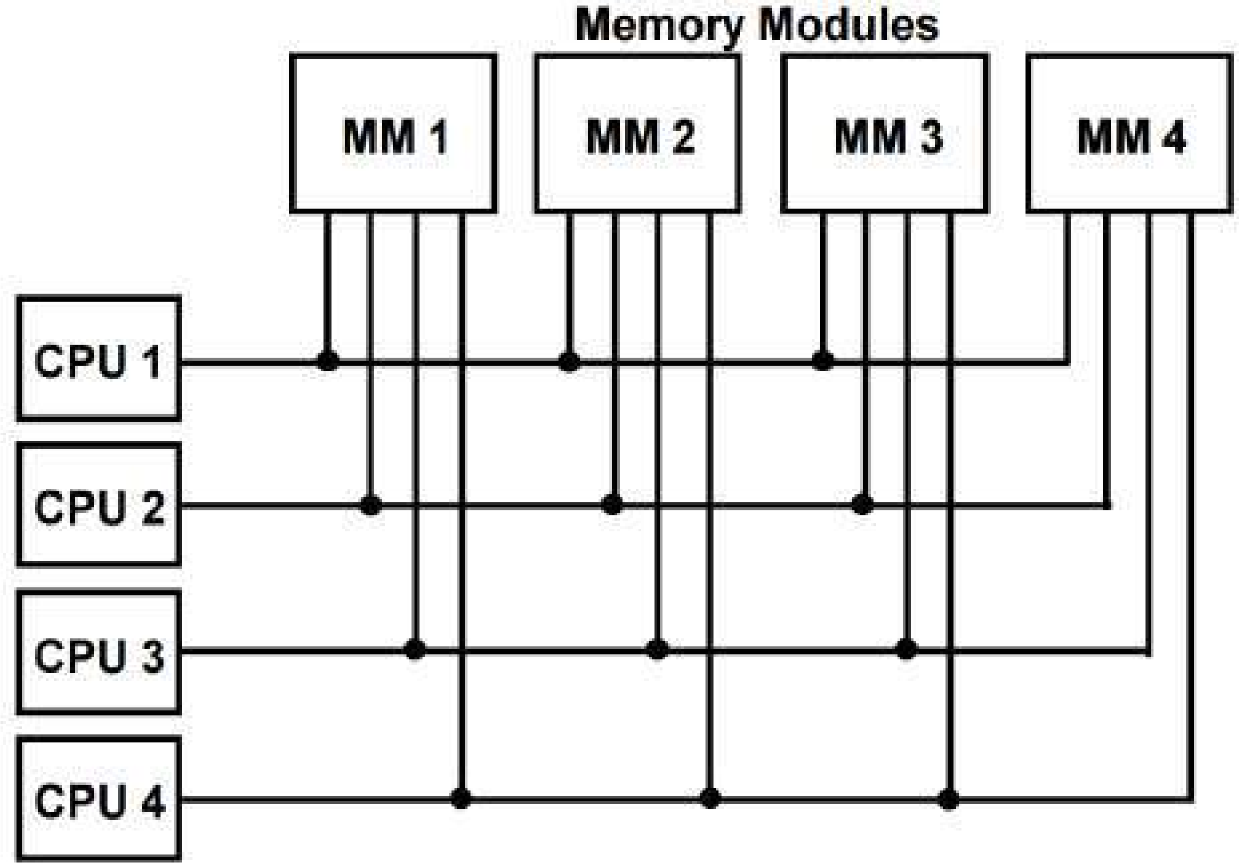


Figure 13-3 Multi-port memory organization.

Crossbar switch:

→The crossbar switch organization consists of a number of cross points that are placed at intersections between processor buses and memory module paths.

→**Figure 13-4** shows a crossbar switch interconnection between four CPUs and four memory modules.

→The small square in each cross point is a switch that determines the path from a processor to a memory module. Each switch point has control logic to set up the transfer path between a processor and memory. It examines the address that is placed in the bus to determine whether its particular module is being addressed. It also resolves multiple requests for access to the same memory module on a predetermined priority basis.

→**Figure 13-5** shows the functional design of a crossbar switch connected to one memory module. The circuit consists of multiplexers that select the data, address, and control from one CPU for communication with the memory module. Priority levels are established by the arbitration logic to select one CPU when two or more CPUs attempt to access the same memory. The multiplexers are controlled with the binary code that is generated by a priority encoder within the arbitration logic.

→A crossbar switch organization supports simultaneous transfers from memory modules because there is a separate path associated with each me module. However, the hardware required to implement the switch can become quite large and complex.

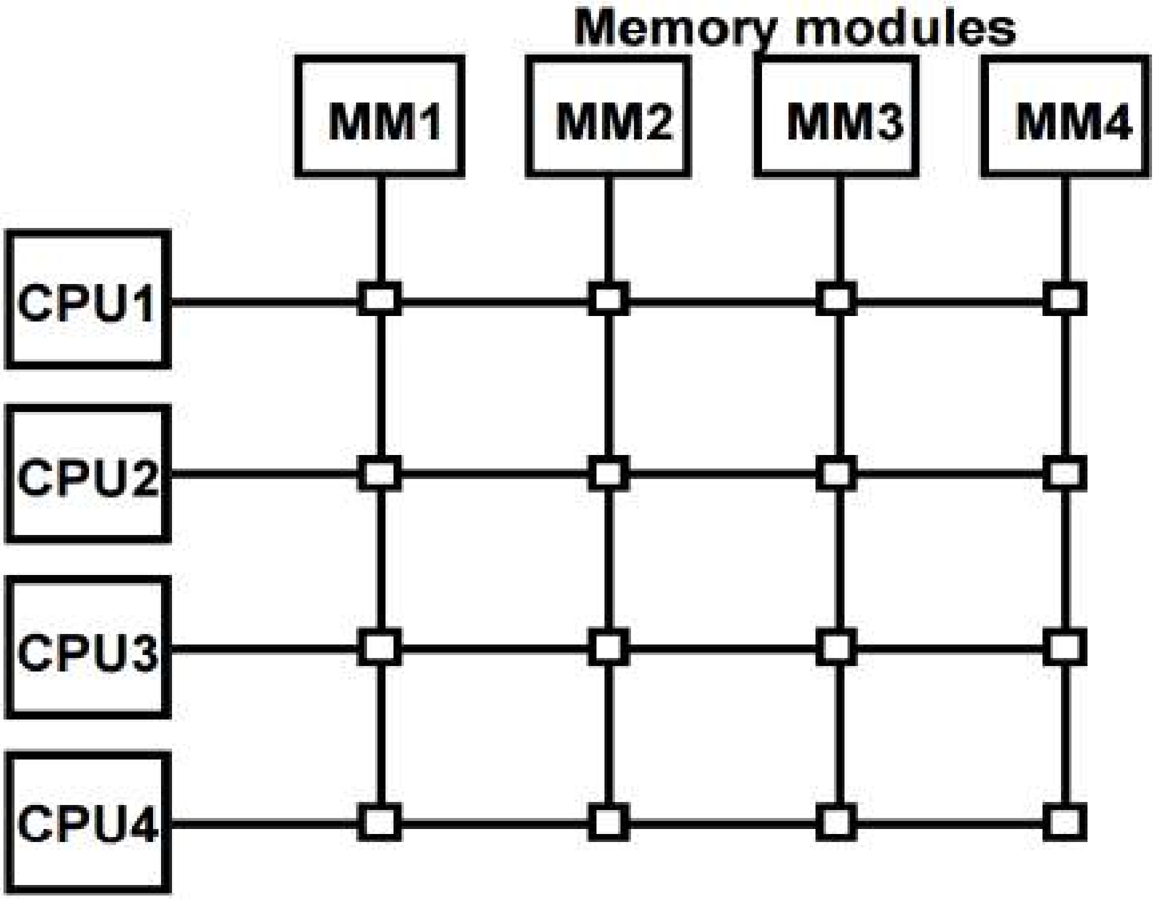


Figure 13-4 Crossbar switch.

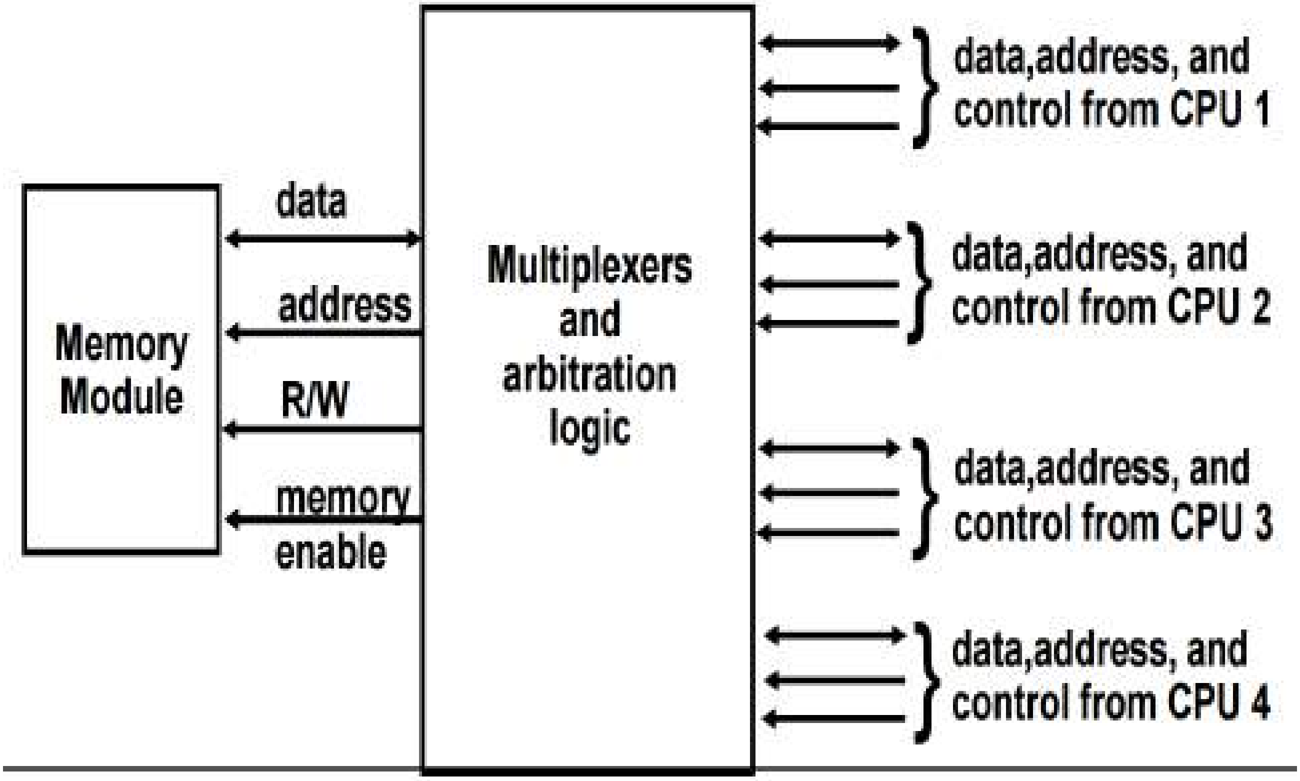

Memory Module
data
address
R/W
memory enable
Multiplexers and arbitration logic
data,address, and control from CPU 1
data,address, and control from CPU 2
data,address, and control from CPU 3
data,address, and control from CPU 4

Figure 13-15 Block diagram of crossbar switch.

Multistage switching network:

→The basic component of a multistage network is a two-input, two-output interchange switch. As shown in **Fig. 13-6**, the 2X2 switch has two input labeled A and B, and two outputs, labeled 0 and 1. There are control sign (not shown) associated with the switch that establish the interconnection between the input and output terminals. The switch has the capability connecting input A to either of the outputs. Terminal B of the switch behaves in a similar fashion. The switch also has the capability to arbitrate between conflicting requests. If inputs A and B both request the same output terminal only one of them will be connected; the other will be blocked.

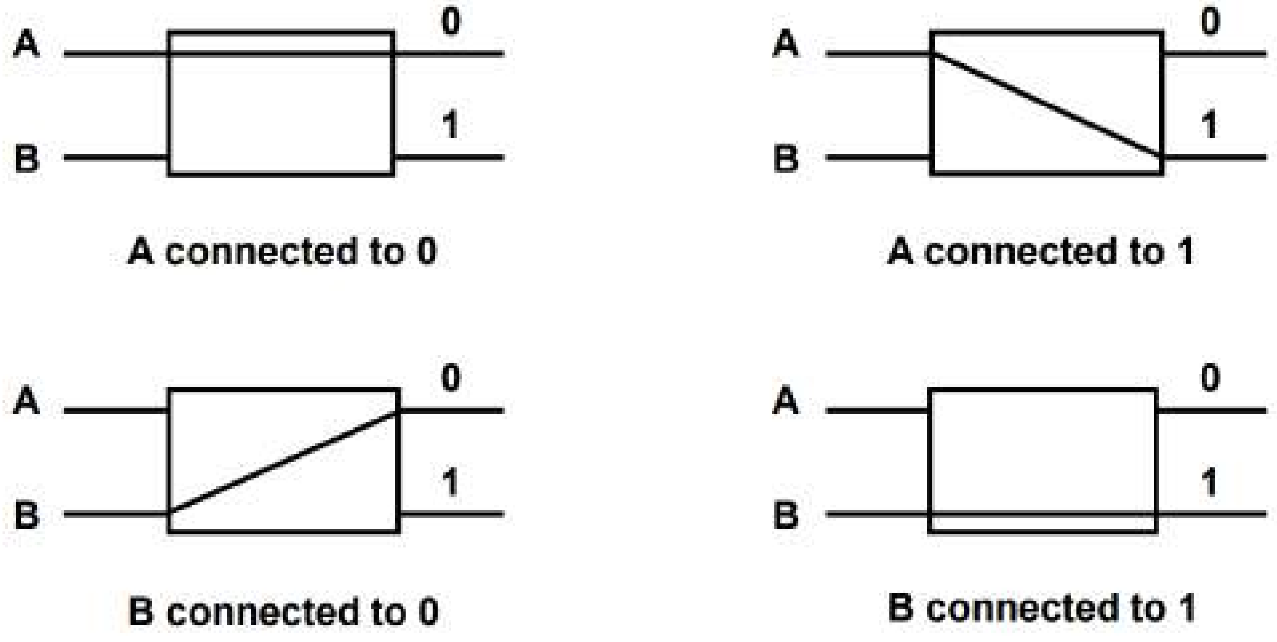


Figure 13-6 Operation of a 2x2 interchange switch.

→Using the 2x2 switch as a building block, it is possible to build multistage network to control the communication between a number of sources and destinations.

→Consider the binary tree shown **Fig. 13-7**. The two processors P1 and P2 are connected through switches to eight memory modules marked in binary from 000 through 111. The path from source to a destination is determined from the binary bits of the destination number. The first bit of the destination number determines the switch output in the first level. The second bit specifies the output of the switch in the second level, and the third bit specifies the output of the switch in the third level.

→For example, to connect P1 to memory 101, it is necessary to form a path from P1to output 1 in the first-level switch, output 0 in the second-level switch, and output 1 in the third-level switch. It is clear that either P1 or P2 can be connected to any one of the eight memories. Certain request patterns, however, cannot be satisfied simultaneously. For example, if P1 is connected to one of the destinations 000 through 011, P2 can be connected to only one of the destinations100 through 111.

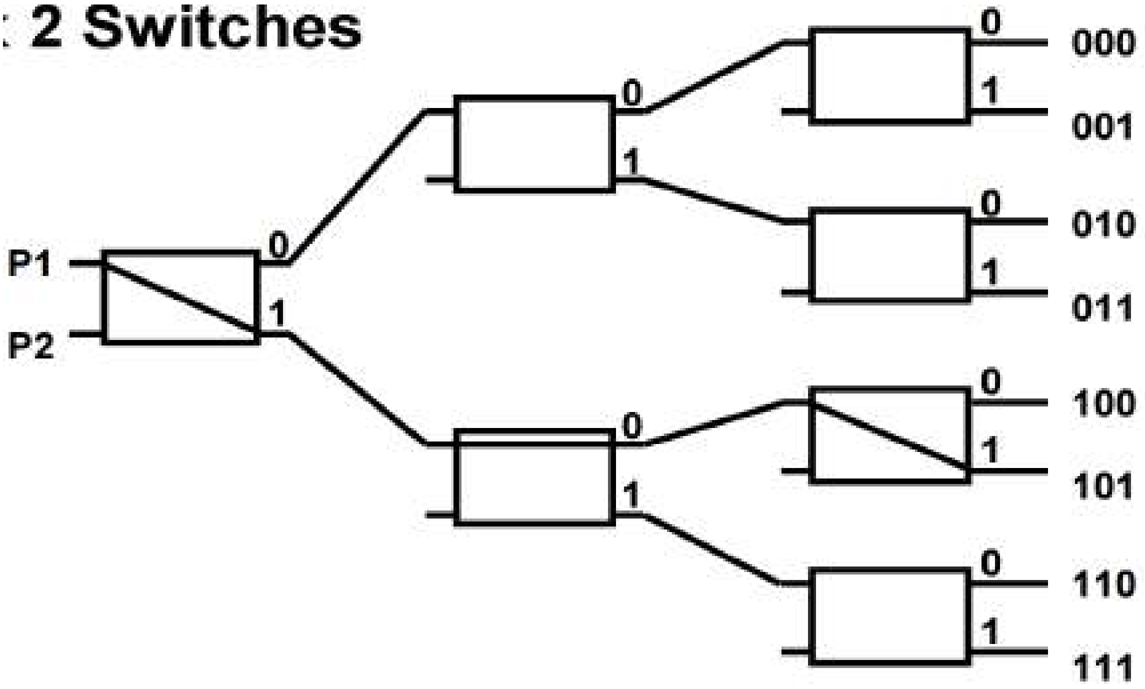


Figure 13-17 Binary trees with 2x2 switches.

→Many different topologies have been proposed for multistage switching networks to control processor-memory communication in a tightly coupled multiprocessor system or to control the communication between the processing elements in a loosely coupled system.

→One such topology is the omega switching network shown in **Fig. 13-8**. In this configuration, there is exactly one path from each source to any particular destination. Some request patterns, however, cannot be connected simultaneously.

→For example, any two sources cannot be connected simultaneously to destinations 000 and 001.

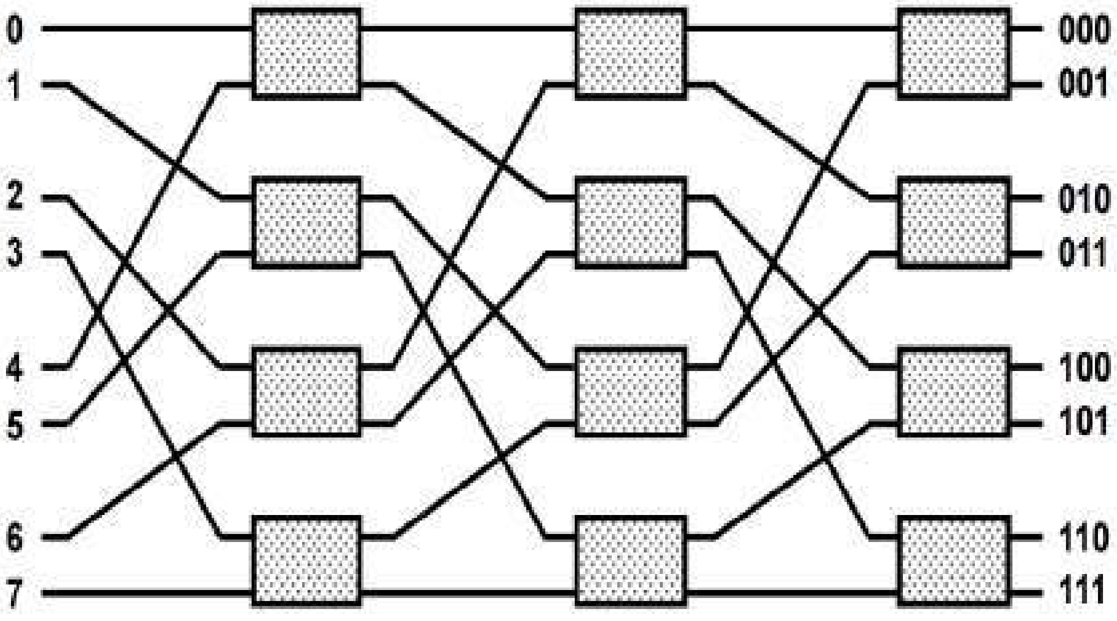


Figure 13-8 8x8 omega switching network.

Hypercube Interconnection:

→The hypercube or binary n-cube multiprocessor structure is a loosely coupled system composed of N = 2^n processors interconnected in an n - dimensional binary cube. Each processor forms a node of the cube.

→There are 2^n distinct n-bit binary addresses that can be assigned to the processors. Each processor address differs from that of each of its n neighbors by exactly one bit position.

→**Figure 13-9** shows the hypercube structure for n = 1, 2, and 3. A one-cube structure has n = 1 and 2^n = 2. It contains two processors interconnected bya single path. A two-cube structure has n = 2 and 2^n = 4. It contains four no disinter connected as a square. A three-cube structure has eight nodes inter connected as a cube.

→An n -cube structure has 2^n nodes with a processor residing in each node. Each node is assigned a binary address in such a way that the addresses of two neighbors differ in exactly one bit position.

→For example, the three neighbors of the node with address 100 in a three-cube structure are 000, 110, and 101. Each of these binary numbers differs from address 100 by one bit value.

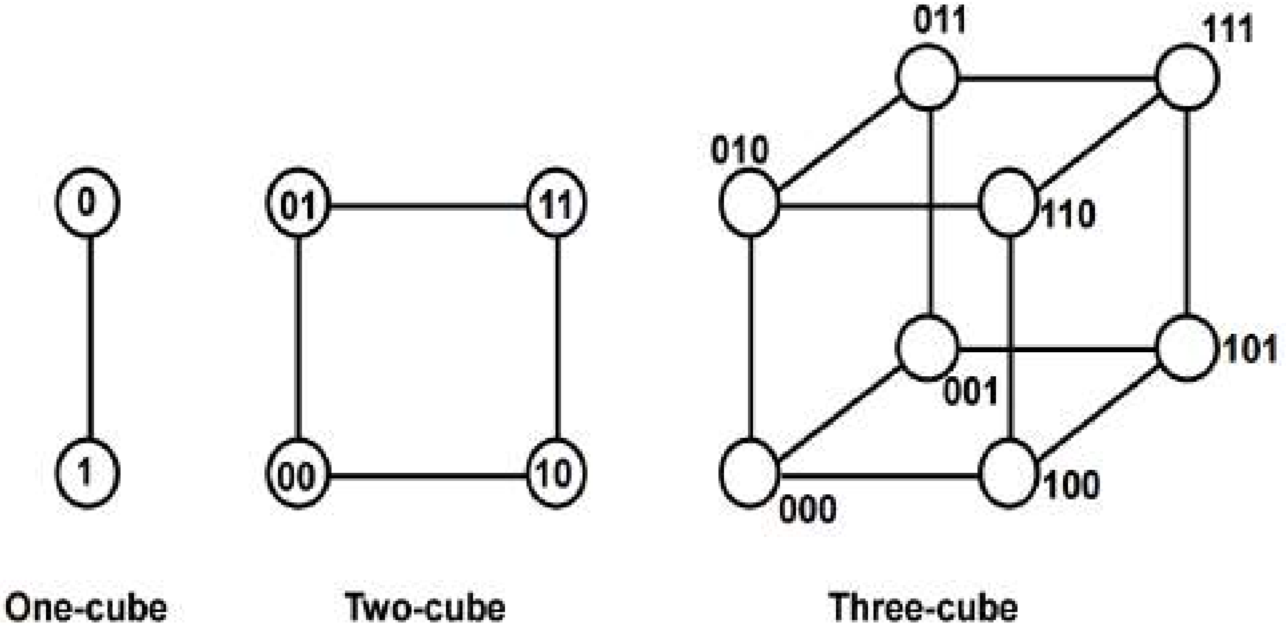


Figure 13-19 Hypercube structures for n=1, 2, 3.

→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 (from 000 to 001 to 011 or from

000to 010 to 011). It is necessary to go through at least three links to communicate from node 000 to node 111.

→A routing procedure can be developed by computing the exclusive-OR of the source node address with the destination node address. The resulting binary value will have 1 bits corresponding to the axes on which the two nodes differ. The message is then sent along any one of the axes.

→For example, in a three-cube structure, a message at 010 going to 001produces an exclusive-OR of the two addresses equal to 01 1 . The message can be sent along the second axis to 000 and then through the third axis to 001.

4. Interprocessor Arbitration:

→Computer systems contain a number of buses at various levels to facilitate the transfer of information between components.

→The CPU contains a number of internal buses for transferring information between processor registers and ALU.

→A memory bus consists of lines for transferring data, address, and read/write information.

→An I/O bus is used to transfer information to and from input and output devices. A bus that connects major components in a multiprocessor system, such as CPUs, IOPs, and memory, is called a system bus.

- Only one of CPU, IOP, and Memory can be granted to use the bus at a time.
- Arbitration mechanism is needed to handle multiple requests to the shared resources to resolve multiple contentions.

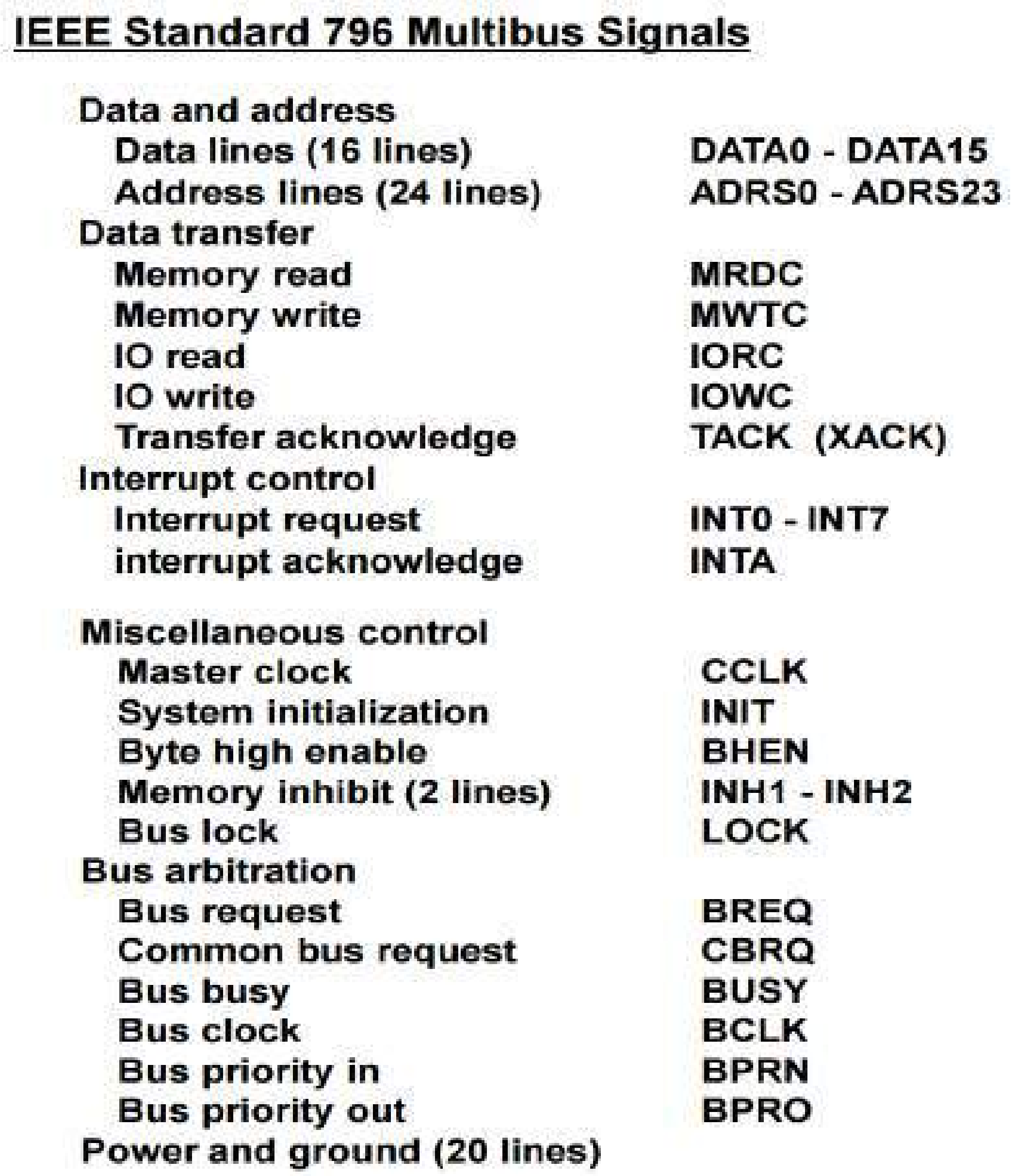
System Bus:

→A bus that connects the major components such as CPU’s, IOP’s and memory.

→A typical system bus consists of approximately 100 signal lines. These lines are divided into three functional groups: data, address, and control. In addition, there are power distribution lines that supply power to the components.

→For example, the IEEE standard 796 multi bus system has 16 data lines, 24 address lines, 26 control lines, and 20 power lines, for a total of 86 lines.

→**Table 13-1** lists the 86 lines that are available in the IEEE standard 796multibus. It includes 16 data lines and 24 address lines. All signals in the multi bus are active or enabled in the low-level state. The data transfer control signals include memory read and write as well as 110 read and write. Consequently, the address lines can be used to address separate memory and 110spaces.



IEEE Standard 796 Multibus Signals

Data and address	
Data lines (16 lines)	DATA0 - DATA15
Address lines (24 lines)	ADRS0 - ADRS23
Data transfer	
Memory read	MRDC
Memory write	MWTC
IO read	IORC
IO write	IOWC
Transfer acknowledge	TACK (XACK)
Interrupt control	
Interrupt request	INT0 - INT7
interrupt acknowledge	INTA
Miscellaneous control	
Master clock	CCLK
System initialization	INIT
Byte high enable	BHEN
Memory inhibit (2 lines)	INH1 - INH2
Bus lock	LOCK
Bus arbitration	
Bus request	BREQ
Common bus request	CBRQ
Bus busy	BUSY
Bus clock	BCLK
Bus priority in	BPRN
Bus priority out	BPRO
Power and ground (20 lines)	

Serial Arbitration Procedure:

→**Figure 13-10** shows the daisy-chain connection of four arbiters. It is assumed that each processor has its own bus arbiter logic with priority-in and priority-out lines.

→The priority out (PO) of each arbiter is connected to the priority in (PI) of the next-lower-priority arbiter.

→The PI of the highest-priority unit is maintained at logic 1 value. The highest-priority unit in the system will always receive access to the system bus when it requests it. The PO output for a particular arbiter is equal to 1 if its PI input is equal to 1 and the processor associated with the arbiter logic is not requesting control of the bus. This is the way that priority is passed to the next unit in the chain. If the processor requests control of the bus and the corresponding arbiter finds its PI input equal to 1, it sets its PO output to 0. Lower-priority arbiters receive a 0 in PI and generate a 0 in PO. Thus the processor whose arbiter has a PI = 1 and PO = 0 is the one that is given control of the system bus.

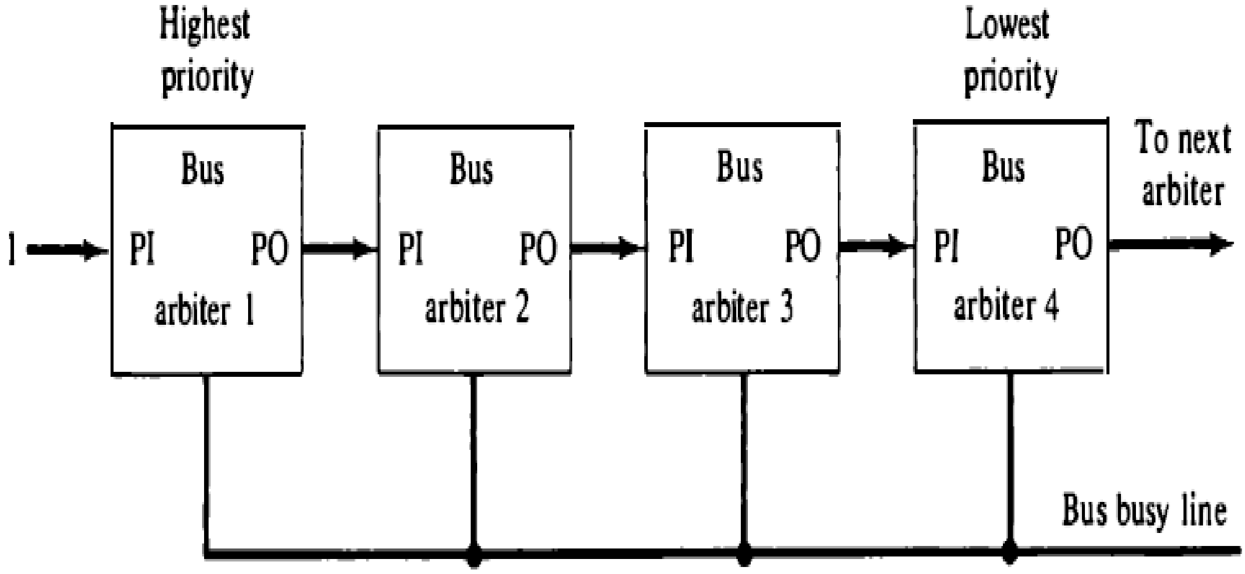


Figure 13-10 Serial (daisy-chain) arbitration.

→ The bus busy line shown in Fig. 13-10 provides a mechanism for an orderly transfer of control. The busy line comes from open-collector circuits in each unit and provides a wired-OR logic connection.

→If the arbiter finds the busy line active, it means that another processor is currently using the bus. The arbiter keeps examining the busy line while the lower-priority processor that lost control of the bus completes its operation. When the bus busy line returns to its inactive state, the higher-priority arbiter enables the busy line, and its corresponding processor can then conduct the required bus transfers.

Parallel Arbitration Logic:

→The parallel bus arbitration technique uses an external priority encoder and a decoder as shown in **Fig. 13-11**. Each bus arbiter in the parallel scheme has a bus request output line and a bus acknowledge input line. Each arbiter enables the request line when its processor is requesting access to the system bus. The processor takes control of the bus if its acknowledge input line is enabled. The bus busy line provides an orderly transfer of control, as in the daisy-chaining case.

→**Figure 13-11** shows the request lines from four arbiters going into a 4 x 2priority encoder. The output of the encoder generates a 2-bit code which represents the highest-priority unit among those requesting the bus. The truth table of the priority encoder can be found in Table 11-2 (Sec. 11-5). The 2-bitcode from the encoder output drives a 2 x 4 decoder which enables the proper acknowledge line to grant bus access to the highest-priority unit.

→The bus priority-in BPRN and bus priority-out BPRO are used for a daisy-chain connection of bus arbitration circuits. The bus busy signal BUSY is an open-collector output used to instruct all arbiters when the bus is busy conducting a transfer. The common bus request CBRQ is also an open-collector output that serves to instruct the arbiter if there are any other arbiters of lower-priority requesting use of the system bus. The signals used to construct a parallel arbitration procedure are bus request BREQ and priority-in BPRN, corresponding to the request and acknowledge signals in Fig. 13-11. The busclock BCLK is used to synchronize all bus transactions.

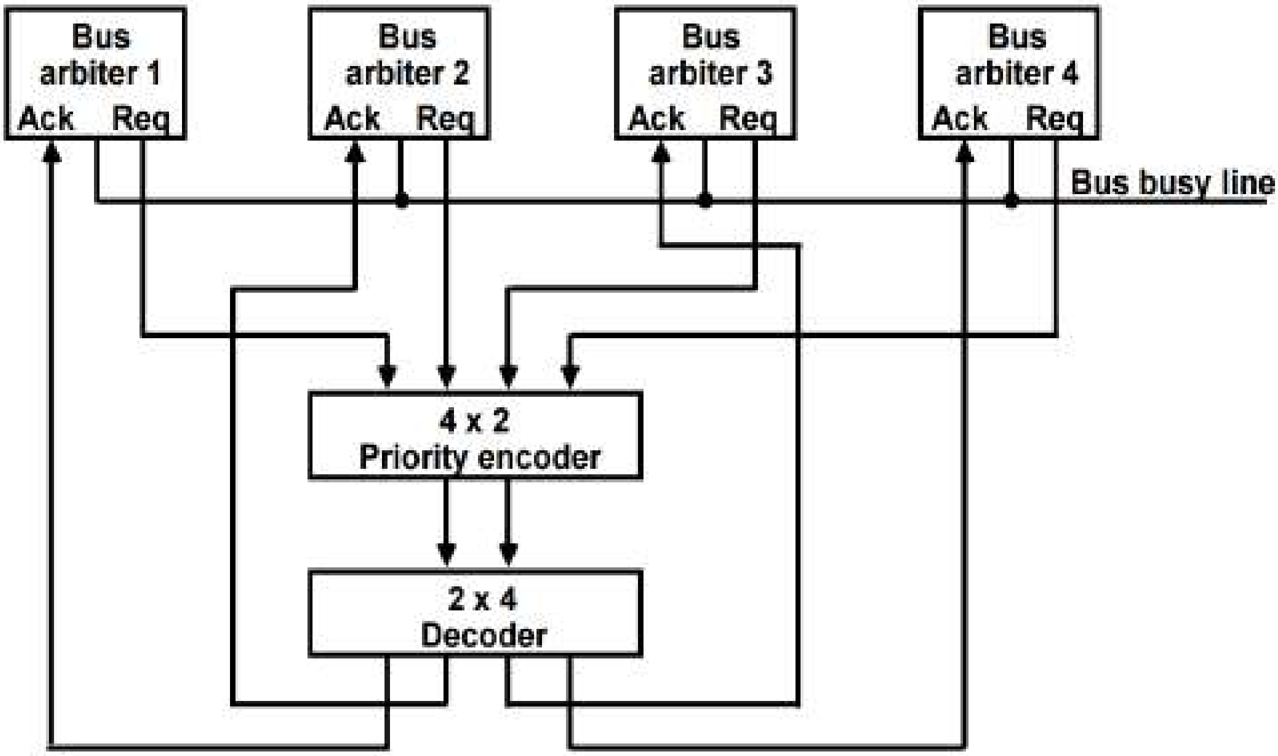


Figure 13-11 Parallel arbitration.

Dynamic Arbitration Algorithms:

→The *time slice algorithm* allocates a fixed-length time slice of bus time that is offered sequentially to each processor, in round-robin fashion. The service given to each system component with this scheme is independent of its location along the bus. No preference is given to any particular device since each is allotted the same amount of time to communicate with the bus.

→In a bus system that uses *polling*, the bus grant signal is replaced by a set of lines called poll lines which are connected to all units. These lines are used by the bus controller to define an address for each device connected to the bus. The bus controller sequences through the addresses in a prescribed manner. When a processor that requires access recognizes its address, it activates the bus busy line and then accesses the bus. After a number of bus cycles, the polling process continues by choosing a different processor. The polling sequence is normally programmable, and as a result, the selection priority canbe altered under program control.

→*The least recently used(LRU) algorithm* gives the highest priority to there questing device that has not used the bus for the longest interval. The priorities are adjusted after a number of bus cycles according to the LRU algorithm. 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.

→In *the first-come, first-serve scheme*, requests are served in the order received. To implement this algorithm, the bus controller establishes a queue arranged according to the time that the bus requests arrive. Each processor must wait for its turn to use the bus on a first-in, first-out (FIFO) basis.

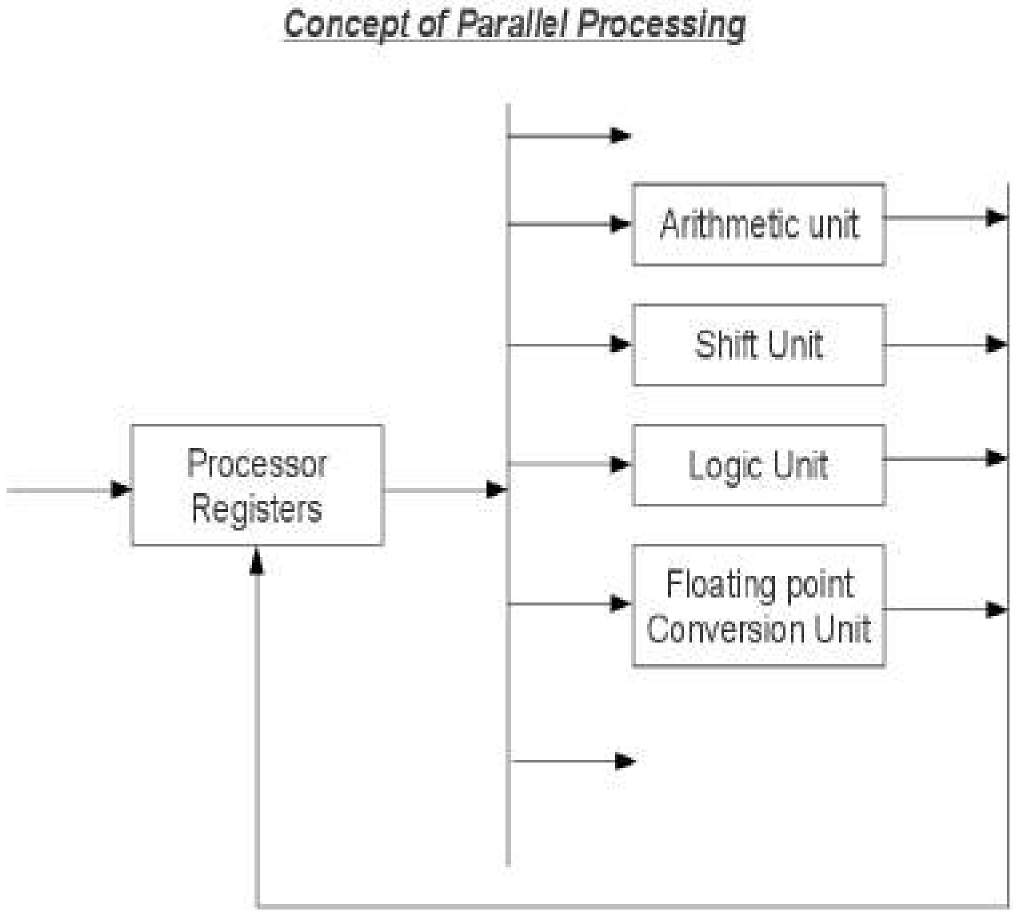
→The *rotating daisy-chain procedure* is a dynamic extension of the daisychain algorithm. In this scheme there is no central bus controller, and the priority line is connected from the priority-out of the last device back to the priority-in of the first device in a closed loop.

5. Parallel Processing:

→The term parallel processing indicates that the system is able to perform several operations in a single time. Now we will elaborate the scenario, in a CPU we will be having only one Accumulator which will be storing the results obtained from the current operation. Now if we are giving only one command such that "a+b" then the CPU performs the operation and stores the result in the accumulator. Now we are talking about parallel processing, therefore we will be issuing two instructions "a+b" and "c-d" in the same time, now if the result of "a+b" operation is stored in the accumulator, then "c-d" result cannot be stored in the accumulator in the same time. Therefore the term parallel processing in not only based on the Arithmetic, logic or shift operations. The above problem can be solved in the following manner. Consider the registers R1 and R2 which will be storing the operands before operation and R3 is the register which will be storing the results after the operations. Now the above two instructions "a+b" and "c-d" will be done in parallel as follows.

- Values of "a" and "b" are fetched in to the registers R1 andR2
- The values of R1 and R2 will be sent into the ALU unit to perform the addition
- The result will be stored in the Accumulator
- When the ALU unit is performing the calculation, the next data "c" and "d" are brought into R1 and R2.
- Finally the value of Accumulator obtained from "a+b" will be transferred into theR3
- Next the values of C and D from R1 and R2 will be brought into the ALU to perform the "c-d" operation.
- Since the accumulator value of the previous operation is present in R3, the result of "c-d" can be safely stored in the Accumulator.

This is the process of parallel processing of only one CPU. Consider several such CPU performing the calculations separately. This is the concept of parallel processing.


Concept of Parallel Processing
Processor Registers
Arithmetic unit
Shift Unit
Logic Unit
Floating point Conversion Unit

→In the above figure we can see that the data stored in the processor registers is being sent to separate devices basing on the operation needed on the data. If the data inside the processor registers is requesting for an arithmetic operation, then the data will be sent to the arithmetic unit and if in the same time another data is requested in the logic unit, then the data will be sent to logic unit for logical operations. Now in the same time both arithmetic operations and logical operations are executing in parallel. This is called as parallel processing.

Instruction Stream: The sequence of instructions read from the memory is called as an Instruction Stream

Data Stream: The operations performed on the data in the processor is called as a Data Stream.
The computers are classified into 4 types based on the Instruction Stream and Data Stream. They are called as the Flynn's Classification of computers.

COMPUTER ORGANIZATION-UNIT5

Flynn's Classification of Computers:

- Single Instruction Stream and Single Data Stream(SISD)
- Single Instruction Stream and Multiple Data Stream(SIMD)
- Multiple Instruction Stream and Single Data Stream(MISD)
- Multiple Instruction Stream and Multiple Data Stream(MIMD)

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 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.

MISD structure is only of theoretical interest since no practical system has been constructed using this organization because multiple instruction streams means more no of instructions, therefore we have to perform multiple instructions on same data at a time. This is practically impossible.

MIMD structure refers to a computer system capable of processing several programs at the same time operating on different data.

6. 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. We can consider the pipelining concept as a collection of several segments of data processing programs which will be processing the data and sending the results to the next segment until the end of the processing is reached. We can visualize the concept of pipelining in the example below.

Consider the following operation: Result=(A+B)*C

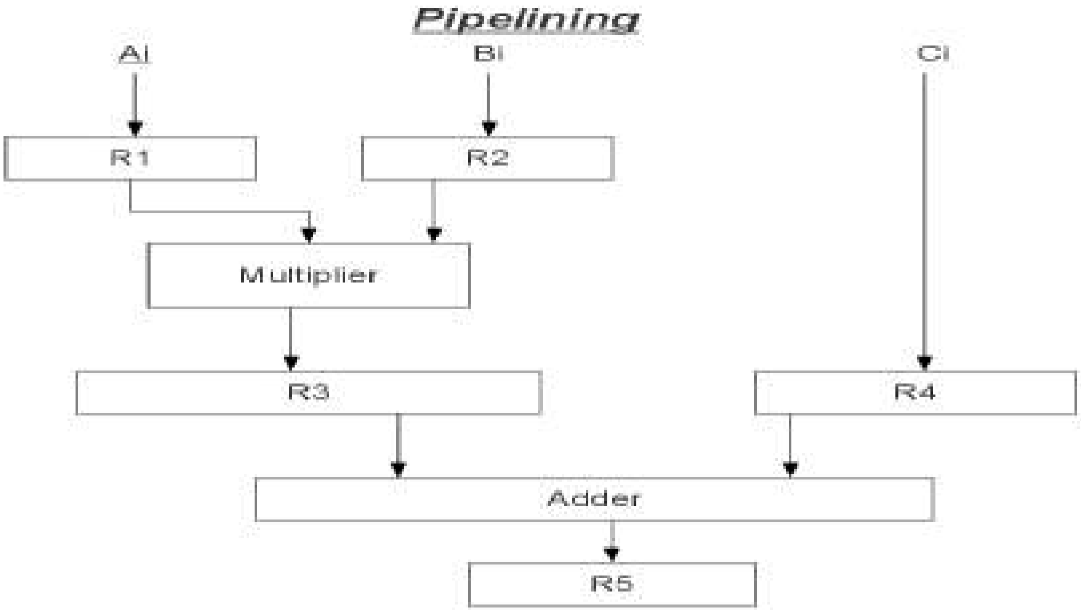
- First the A and B values are Fetched which is nothing but a “Fetch Operation”.
- The result of the Fetch operations is given as input to the Addition operation, which is an Arithmetic operation.
- The result of the Arithmetic operation is again given to the Data operand C which is fetched from the memory and using another arithmetic operation which is

Multiplication in this scenario is executed.

- Finally the Result is again stored in the “Result” variable.

→In this process we are using up-to 5 pipelines which are the

→ Fetch Operation (A)| Fetch Operation (B)|Addition of (A & B) | Fetch Operation(C) | Multiplication of ((A+B), C) | Load ((A+B)*C),Result);



The contents of the Registers in the above pipeline concept are given below. We are considering the implementation of A[7] array with B[7] array

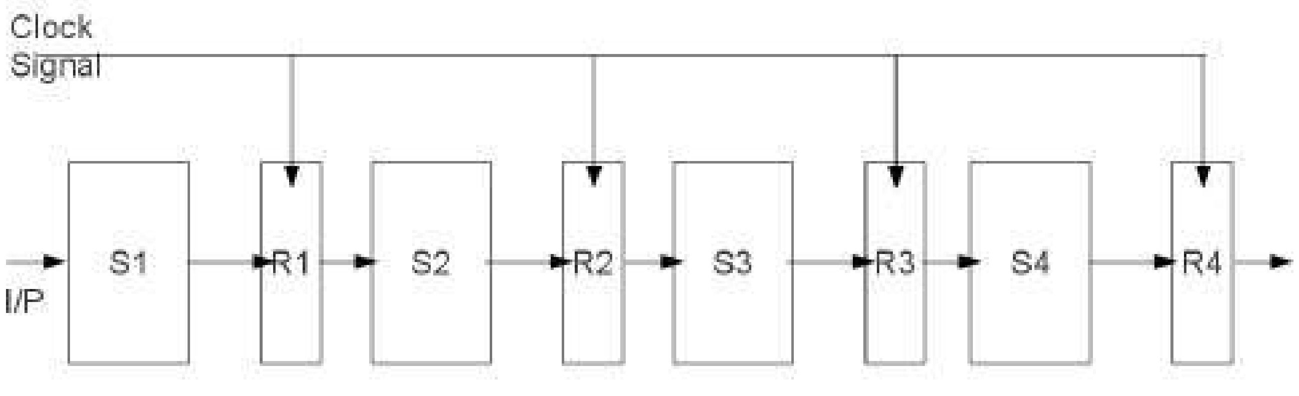
Clock Pulse Number	Segment1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A1	B1	-	-	-
2	A2	B2	A1*B1	C1	-
3	A3	B3	A2*B2	C2	A1*B1+C1
4	A4	B4	A3*B3	C3	A2*B2+C2
5	A5	B5	A4*B4	C4	A3*B3+C3
6	A6	B6	A5*B5	C5	A4*B4+C4
7	A7	B7	A6*B6	C6	A5*B5+C5

8	A7*B7	C7	A6*B6+C6
9			A7*B7+C7

→If the above concept is executed without the pipelining, then each data operation will be taking 5 cycles, totally they are 35 cycles of CPU are needed to perform the operation. But if are using the concept of pipeline, we will be cutting off many cycles. Like given in the table below when the values of A1 and B1 are coming into the registers R1 and R2, the registers R3, R4 and R5 are empty. Now in the second cycle the multiplication of A1 and B1 is transferred to register R3, now in this point the contents of the register R1 and R2 are empty. Therefore the next two values A2 and B2 can be brought into the registers. Again in the third cycle after fetching the C1 value the operation (A1*B1) + C1 will be performed. So in this way we can achieve the total concept in only 9 cycles. Here we are assuming that the clock cycle timing is fixed. This is the concept of pipelining.

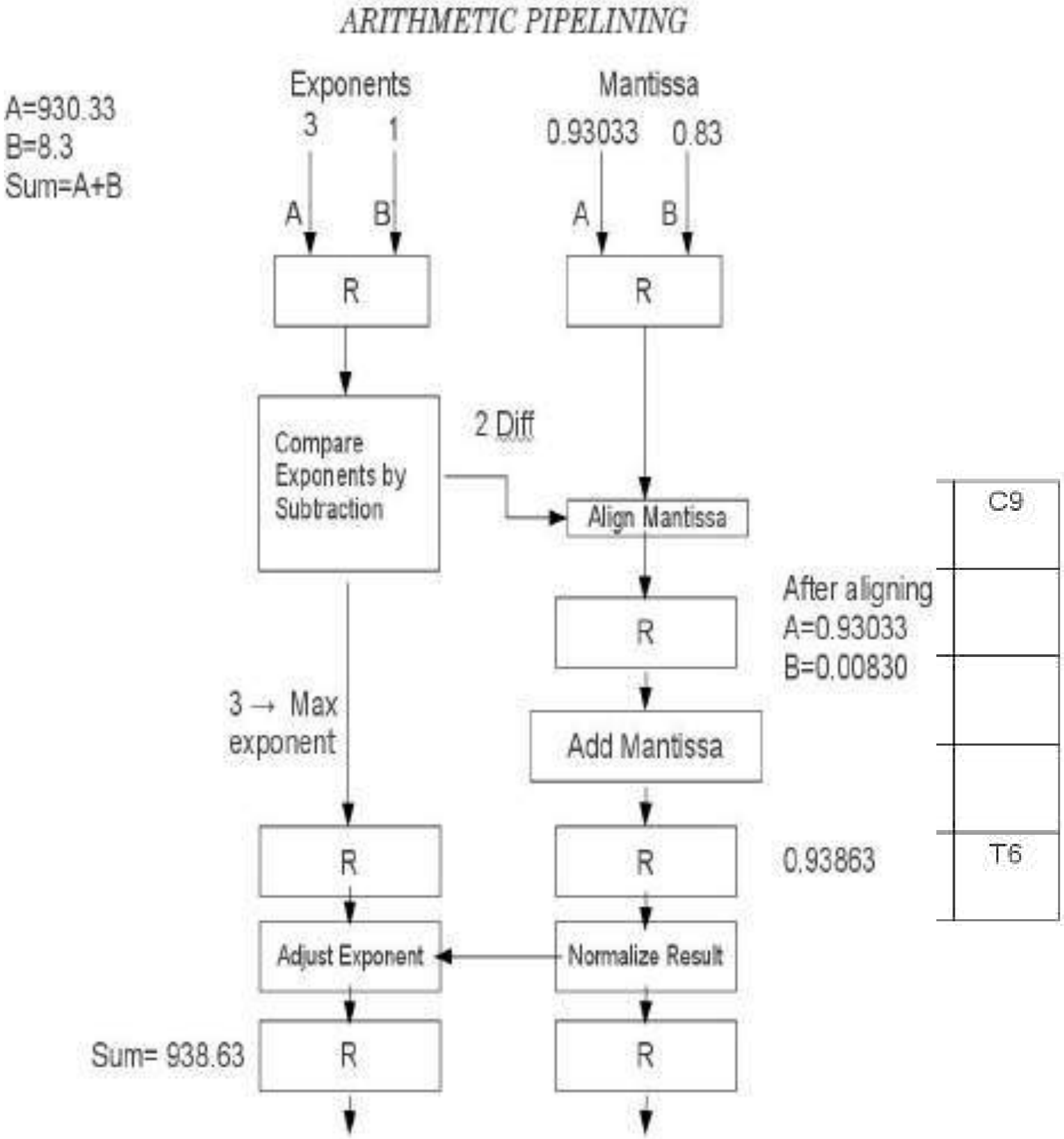
Below is the diagram of 4 segment pipeline.

Segment Representation



4 Segment pipeline

The below table is the space time diagram for the execution of 6 tasks in the 4 segment pipeline.



→The above diagram represents the implementation of arithmetic pipeline in the area of floating point arithmetic operations. In the diagram, we can see that two numbers A and B are added together. Now the values of A and B are not normalized, therefore we must normalize them before start to do any operations. The first thing is we have to fetch the values of A and B into the registers. Here R denote a set of registers. After that the values of A and B are normalized, therefore the values of the exponents will be compared in the comparator. After that the alignment of mantissa will be taking place. Finally, we will be performing addition, since an addition is happening in the adder circuit. The source registers will be free and the second set of values can be brought. Like wise when the normalizing of the result is taking place, addition of the new values will be added in the adder circuit and when addition is going on, the new data values will be brought into the registers in the start of the implementation. We can see how the addition is being performed in the diagram.

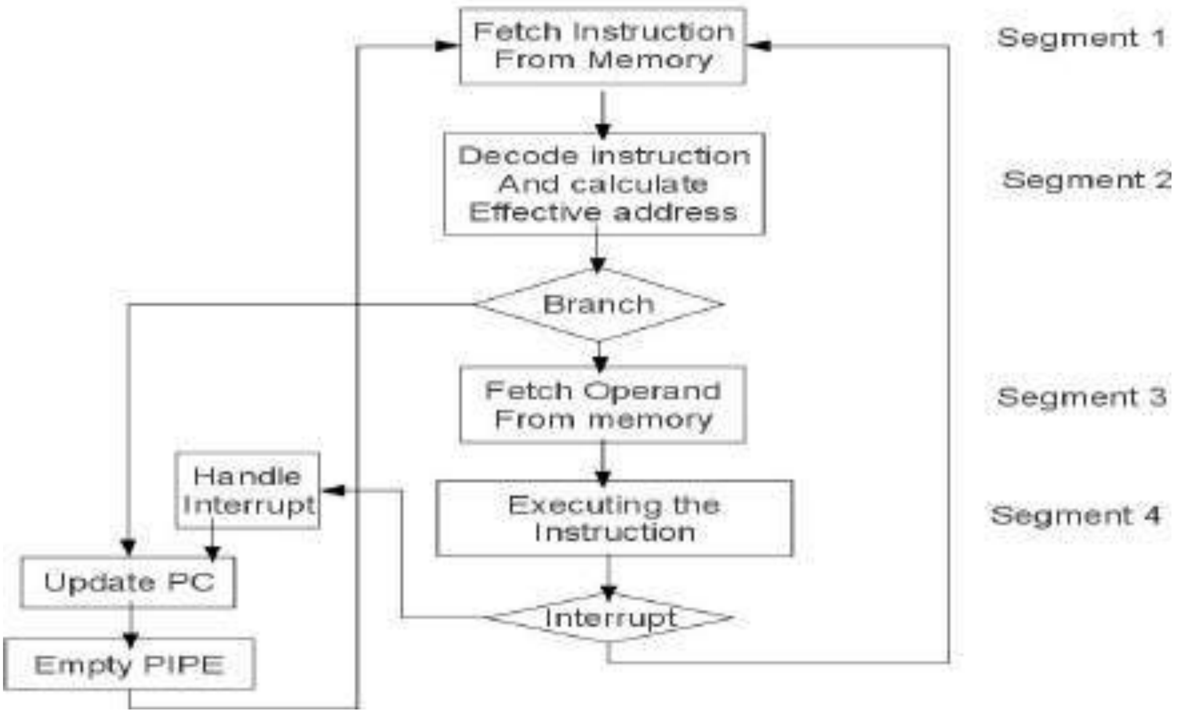
7. Instruction Pipeline:

→Pipelining concept is not only limited to the data stream, but can also be applied on the instruction stream. The instruction pipeline execution will be like the queue execution. In the queue the data that is entered first, will be the data first retrieved. Therefore when an instruction is first coming, the instruction will be placed in the queue and will be executed in the system. Finally the results will be passing on to the next instruction in the queue. This scenario is called as Instruction pipelining. The instruction cycle is given below

- Fetch the instruction from the memory
- Decode the instruction
- calculate the effective address
- Fetch the operands from the memory
- Execute the instruction
- Store the result in the proper place.

In a computer system each and every instruction need not necessary to execute all the above

Instruction pipelining

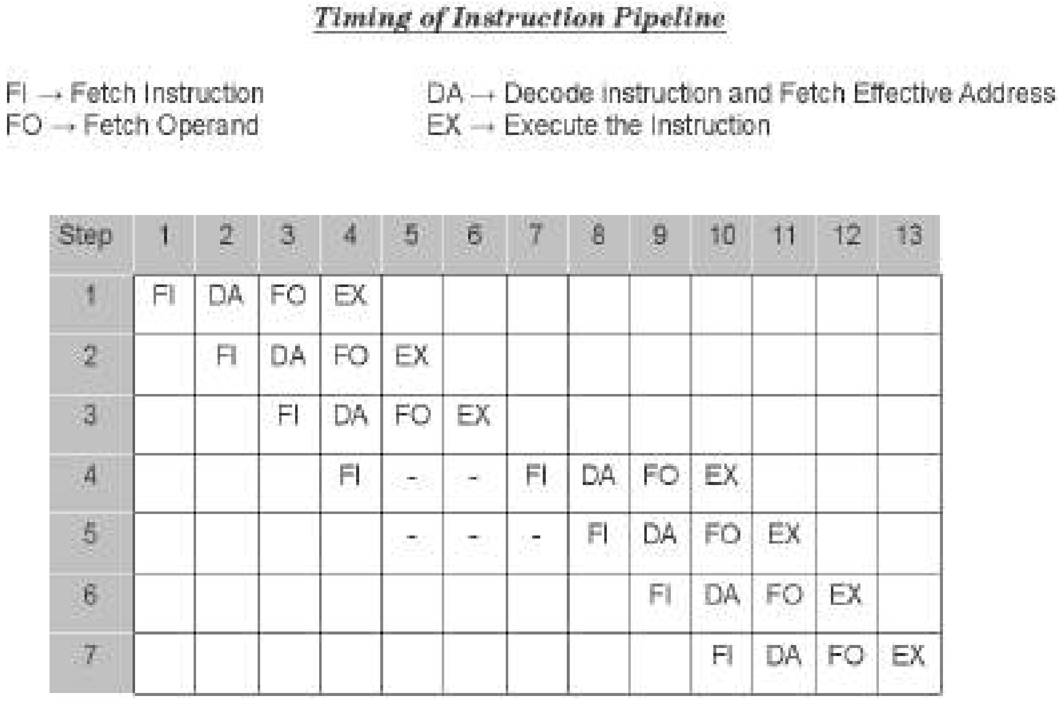

Fetch Instruction From Memory
Segment 1
Decode instruction And calculate Effective address
Segment 2
Branch
Fetch Operand From memory
Segment 3
Handle Interrupt
Executing the Instruction
Segment 4
Update PC
Interrupt
Empty PIPE

phases. In a Register addressing mode, there is no need of the effective address calculation. Below is the example of the four segment instruction pipeline.

In the above diagram we can see that the instruction which is first executing has to be fetched from the memory, there after we are decoding the instruction and we are calculating the effective address. Now we have two ways to execute the instruction. Suppose we are using a normal instruction like ADD, then the operands for that instruction will be fetched and the instruction will be executed. Suppose we are executing an instruction such as Fetch command. The fetch command itself has internally three more commands which are like ACTDR, ARTDR etc.., therefore we have to jump to that particular location to execute the command, so we are using the branch operation. So in a branch operation, again other instructions will be executed. That means we will be updating the PC value such that the instruction can be executed. Suppose we are fetching the operands to perform the original operation such as ADD, we need to fetch the data. The data can be fetched in two ways, either from the main memory or else from an input output devices. Therefore in order to use the input output devices, the devices must generate the interrupts which should be handled by the CPU. Therefore the handling of interrupts is also a kind of program execution. Therefore we again have to start from the starting of the program and execute the interrupt cycle.

The different instruction cycles are given below:

- FI → FI is a segment that fetches an instruction
- DA → DA is a segment that decodes the instruction and identifies the effective address.
- FO → FO is a segment that fetches the operand.
- EX → EX is a segment that executes the instruction with the operand.



Timing of Instruction Pipeline

FI → Fetch Instruction
FO → Fetch Operand
DA → Decode instruction and Fetch Effective Address
EX → Execute the Instruction

Step	1	2	3	4	5	6	7	8	9	10	11	12	13
1	FI	DA	FO	EX									
2		FI	DA	FO	EX								
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

COMPUTER ORGANIZATION-UNIT5

Pipelining Conflicts: There are different conflicts that are caused by using the pipeline concept. They are

- Resource Conflicts: These are caused by access to memory by two or more segments at the same time. Most of these conflicts can be resolved by using separate instruction and data memories
- Data Dependency: These conflicts arise when an instruction depends on the result of a previous instruction, but this result is not yet available.
- Branch difficulties: These difficulties arise from branch and other instructions that change the value of PC.

Data Dependency Conflict: The data dependency conflict can be solved by using the following methods.

- Hardware Interlocks: The most straight forward method is to insert hardware interlocks. An interlock is a circuit that detects instructions whose source operands are destination 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. This approach maintains the program sequence by using hardware to insert the required delay.
- Operand Forwarding: Another technique called operand forwarding uses special hardware to detect a conflict and avoid the conflict path by using a special path to forward the values between the pipeline segments.
- Delayed Load: The delayed load operation is nothing but when executing an instruction in the pipeline, simply delay the execution starting of the instruction such that all the data that is needed for the instruction can be successfully updated before execution.

Branch Conflicts:

The following are the solutions for solving the branch conflicts that are obtained in the pipelining concept.

- Pre-fetch Target Instruction: In this the branch instructions which are to be executed are pre-fetched to detect if any errors are present in the branch before execution.

- Branch Target Buffer: BTB is the associative memory implementation of the branch conditions.
- Loop buffer: The loop buffer is a very high speed memory device. Whenever a loop is to be executed in the computer. The complete loop will be transferred in to the loop buffer memory and will be executed as in the cache memory.
- Branch Prediction: The use of branch prediction is such that, before a branch is to be executed, the instructions along with the error checking conditions are checked. Therefore we will not be going into any unnecessary branch loops.
- Delayed Branch: The delayed branch concept is same as the delayed load process in which we are delaying the execution of a branch process, before all the data is fetched by the system for beginning the CPU.

8. RISC Pipeline:

→The ability to use the instruction pipelining concept in the RISC architecture is very efficient. The simplicity of the instruction set can be utilized to implement an instruction pipeline using a small number of sub operations, with each being executed in one clock cycle. Due to fixed length instruction format, the decoding of the operation can occur at the same time as the register selection. Since the arithmetic, logic and shift operations are done on register basis, there is no need for extra fetching or effective address decoding steps to perform the operation. So pipelining concept can be effectively used in this scenario. Therefore the total operations can be categorized as one segment will be fetching the instruction from program memory, the other segment executes the instruction in the ALU and the third segment may be used to store the result of the ALU operation in a destination register. The data transfer instructions in RISC are limited to only Load and Store instructions. To prevent conflicts in data transfer, we will be using two separate buses one for storing the instructions and other for storing the data.

→Example of three segment instruction pipeline:

We want to perform a operation in which there is some arithmetic, logic or shift operations. Therefore as per the instruction cycle, we will be having the following steps:

- I: Instruction Fetch
- A: ALU Operation
- E: Execute Instruction.

The 1segment will be fetching the instruction from program memory. The instruction is decoded and an ALU operation is performed in the A segment. In the A segment the ALU operation instruction will be fetched and the effective address will be retrieved and finally in the E segment the instruction will be executed.

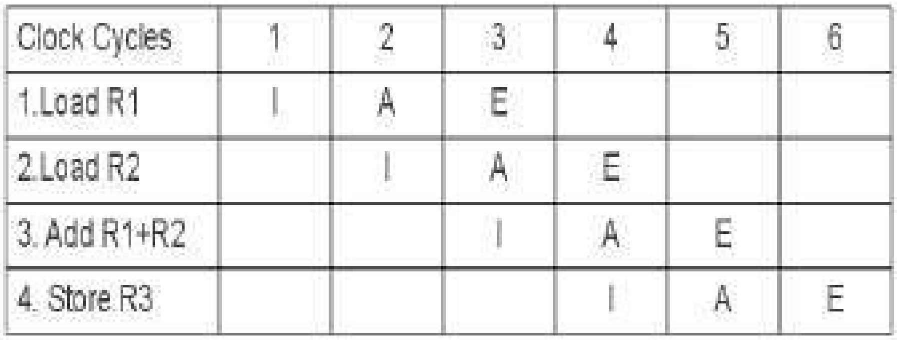
Delayed Load:

Consider the following instructions:

1. LOAD: R1 ← M[address1]
2. LOAD: R2 ← M[address2]
3. ADD: R3 ← R1 +R2
4. STORE: M[address 3] ←R3

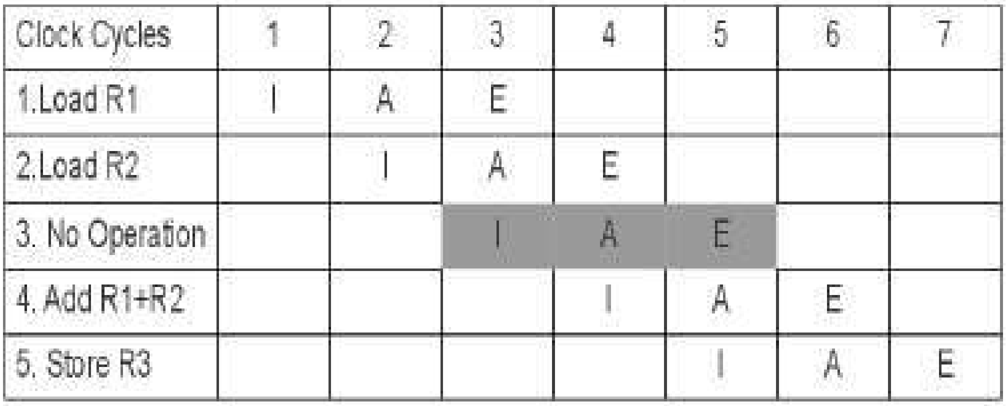
The below tables will be showing the pipelining concept with the data conflict and without conflict.

Pipeline timing with data conflict



Clock Cycles	1	2	3	4	5	6
1.Load R1	I	A	E			
2.Load R2		I	A	E		
3. Add R1+R2			I	A	E	
4. Store R3				I	A	E

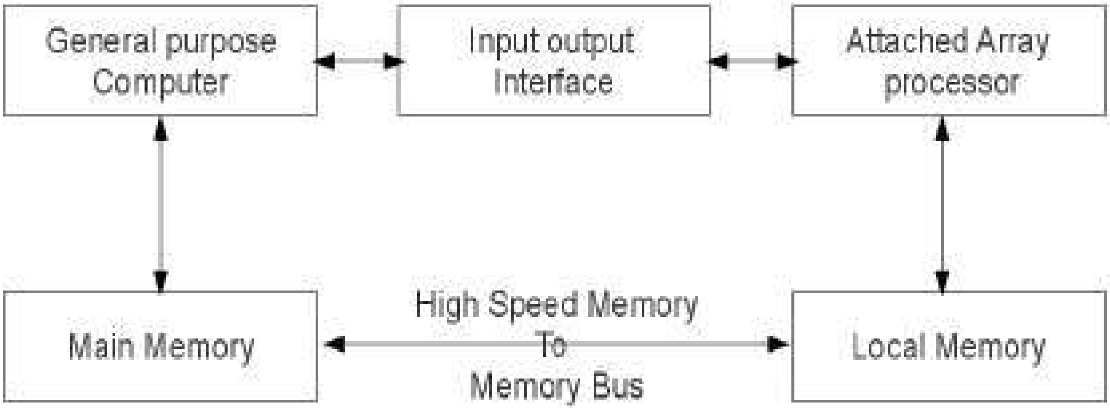
Pipeline timing with delayed load



Clock Cycles	1	2	3	4	5	6	7
1.Load R1	I	A	E				
2.Load R2		I	A	E			
3. No Operation			I	A	E		
4. Add R1+R2				I	A	E	
5. Store R3					I	A	E

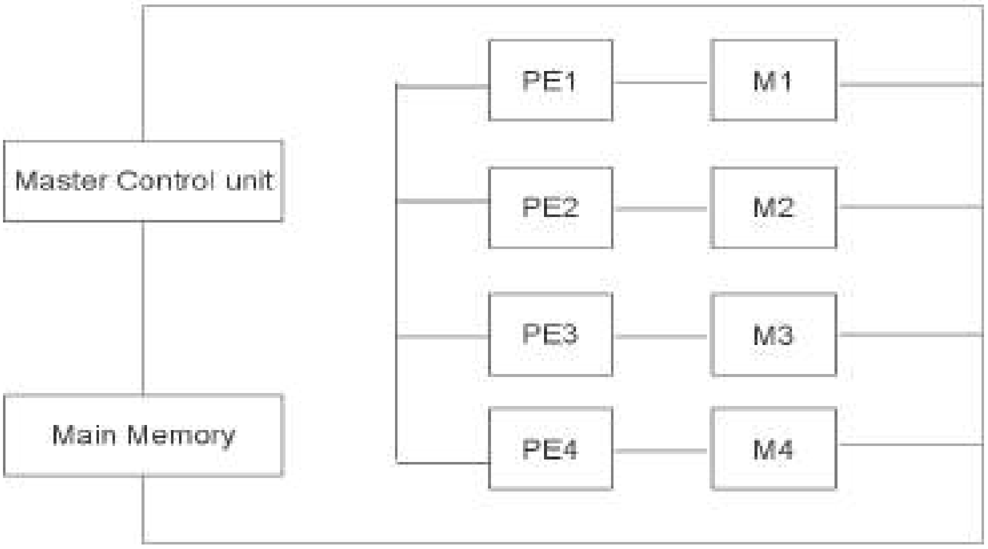
9. Array Processors:

In a distributed computing we will be having several computers working on the same task such that their processing power will be shared among all the systems so that they can perform the task fast. But the disadvantage of the distributed computing is that we have to give separate resources for each system and every system need to be controlled by a task initiating system or can be called as a central control unit. The management of this kind of systems is very hard. In order to perform a specific operation involving a large processing there is no need of distributed computing. The alternate for this kind of scenarios is array processors or attached array processors. The simplest is the SIMD Attached array processor.



Attached Array processor

→The above diagram shows that the system is attached a separate processor which will be used for operation specific purpose. If the array processor is designed for solving floating point arithmetic, then it will only perform that operation. The detailed figure of the attached array processor is given in the diagram below. This will be having the SIMD architecture. In this we will be having a master control unit which will be coordinating all the process in the array processor. Each processing unit in the array processor is having a local memory unit as in the memory interleaving concept on which it performs the operations. Finally we will be having a main memory in which the original source data and the results that are obtained from the array processor will be stored. This is the working principle of the SIMD array processor technology.



SIMD Array Processor Technology

COMPUTER ORGANIZATION-UNIT5

IMPORTANT QUESTIONS (UNIT-5)

1. Explain the Flynn’s classification to accomplish parallel processing.

2) a) Define pipelining? Explain the structure of pipelining with an example.

b) Explaining the implementation of four stage pipelining.

3) List out several of characteristics of multi processors.

4) Explain Instruction Pipeline?

5) What is meant by interconnection structure? Mention the various types of interconnection structures.

6) Explain Inter Process Arbitration

7) What is array processor? Explain the two categories of array processor

8) What is multi processor?

9) What is pipelining?

10) List out the limitations of instruction pipeline.

11) List out the advantages of RISC and CISC