# COMPUTER ORGANIZATION

**Lesson Writers :**

**Smt. T. Anuradha,** M.Sc., M.S.
Associate Professor
Department of Computer Science,
J.K.C. College,
Guntur

**Sri C.V.P.R. Prasad,** M.C.A.
Lecturer
Department of Computer Science,
J.K.C. College,
Guntur

*Editor & Academic Advisor for the Course :*

**Prof. I. Ramesh Babu,** M.E., Ph.D.,
Dept. of Computer Science,
Acharya Nagarjuna University,
Nagarjuna Nagar – 522 510.

***Director***
**Prof. V. Chandrasekhara Rao,** M.Com., Ph.D.,

**CENTRE FOR DISTANCE EDUCATION**
**Acharya Nagarjuna University**
**Nagarjuna Nagar - 522 510**

Ph: 0863-2293299, 2293356, 08645-211023, Cell : 98482 85518
08645 - 211024 ( Study Material )
Website : www.anucde.com,
e-mail : anucde@yahoo.com

**B.Sc. IT Computer Organization**

First Edition : 2010

No. of Copies : 1000

(C) **Acharya Nagarjuna University**

This book is exclusively prepared for the use of students of B.Sc. ITCentre for Distance Education, Acharya Nagarjuna University and this book is meant for limited circulation only.

Published by :
**Prof. V. Chandrasekhara Rao,**
*Director*
Centre for Distance Education,
Acharya Nagarjuna University

*Printed at :*
**Don Bosco Technical School Press**
**Ring Road, Guntur - 7**

# FOREWORD

Since its establishment in 1976, Acharya Nagarjuna University has been forging ahead in the path of progress and dynamism, offering a variety of courses and research contributions. I am extremely happy that by gaining a B++ (80-85) grade from the NAAC in the year 2003, the University has achieved recognition as one of the front rank universities in the country. At present Acharya Nagarjuna University is offering educational opportunities at the UG, PG levels apart from research degrees to students from about 447 affiliated colleges spread over the three districts of Guntur, Krishna and Prakasam.

The University has also started the Centre for Distance Education with the aim to bring higher education within reach of all. The Centre will be a great help to those who cannot join in colleges, those who cannot afford the exorbitant fees as regular students, and even housewives desirous of pursuing higher studies. With the goal of bringing education to the doorstep of all such people, Acharya Nagarjuna University has started offering B.A. and B.Com courses at the Degree level and M.A., M.Com., M.Sc, M.B.A. and LL.M. courses at the PG level from the academic year 2003-2004 onwards.

To facilitate easier understanding by students studying through the distance mode, these self-instruction materials have been prepared by eminent and experienced teachers. The lessons have been drafted with great care and expertise within the stipulated time by these teachers. Constructive ideas and scholarly suggestions are welcome from students and teachers involved respectively. Such ideas will be incorporated for the greater efficacy of this distance mode of education. For clarification of doubts and feedback, weekly classes and contact classes will be arranged at the UG and PG levels respectively.

It is my aim that students getting higher education through the Centre for Distance Education should improve their qualification, have better employment opportunities and in turn facilitate the country's progress. It is my fond desire that in the years to come, the Centre for Distance Education will grow from strength to strength in the form of new courses and by catering to larger number of people. My congratulations to all the Directors, Academic coordinators, Editors and Lesson - writers of the Centre who have helped in these endeavours.

**Prof. Y. R. Haragopal Reddy**
Vice - Chancellor
Acharya Nagarjuna University

M.Sc. IT First Year

## COMPUTER ORGANIZATION

# SYLLABUS

**Unit – I :** Introduction, Computer Evolution and Performance

**Unit – II :** A Top - level view of Computer Function and Interconnection

**Unit – III :** Externel Memory

**Unit – IV :** Computer Arithmetic

**Unit - V :** CPU Structure and Function

**Prescribed Book :**

Stallings William ; Computer organization & Architectures (PHI).
Chapters 1,2,3,6,9,12

**REFERENCE BOOKS :**

Hyes J.P., Computer Architecture & Organization.

M.C.A. First Year

## COMPUTER ORGANIZATION

# SYLLABUS

**Unit – I :** Introduction, Computer Evolution and Performance

**Unit – II :** A Top - level view of Computer Function and Interconnection

**Unit – III :** Externel Memory

**Unit – IV :** Computer Arithmetic

**Unit - V :** CPU Structure and Function

**Prescribed Book :**

Stallings William ; Computer organization & Architectures (PHI).
Chapters 1,2,3,6,9,12

**REFERENCE BOOKS :**

Hyes J.P., Computer Architecture & Organization.

**P.G. Diploma in Information Technology (PGDIT) Syllabus (Course Code: 96)**

# DIT 05 : COMPUTER ORGANIZATION

# SYLLABUS

**Unit – I :** Introduction, Computer Evolution and Performance

**Unit – II :** A Top - level view of Computer Function and Interconnection

**Unit – III :** Externel Memory

**Unit – IV :** Computer Arithmetic

**Unit - V :** CPU Structure and Function

**Prescribed Book :**

Stallings William ; Computer organization & Architectures (PHI).
Chapters 1,2,3,6,9,12

**REFERENCE BOOKS :**

Hyes J.P., Computer Architecture & Organization.

B.Sc. IT First Year

# COMPUTER ORGANIZATION

## SYLLABUS

**Unit – I :**

**Digital Logic Circuits :** Logic gates, Boolean Algebra, Map Simplications, Combinational Circuits, Flip-Flops, Sequential Circuits.

Integrated Circuits and Digital Functions : Digital intergrated Circuits, Decoders & Multiplexes, Binary counters, Shift registers.

**Unit – II :**

**Data Representation :** Data types, Fixed point Representation, Floating point representation, Binary codes, error codes.

Register Transfer and Micro-operations : Register transfer language, Micro Operations.

1. Arthemetic
2. Logic
3. Strings

**Basic Computer Organisation :** Computer Instructions, Timing and Control, Execution of instruction, Input-output and Interupt.

**Unit – III :**

**Central Processing Organisation :** Processor buy organization, ALU, Stock organization, Instruction formats, Addressing modes, Data transfer and Manipulation, Program Control, Parallel processing.

**Arthmetic Processor Design & Arithmetic Algorithms :** Comparision and algorithms - Addition, Multiplication, Division, Subtraction, Multiplication and division, Floating point arithmetic operations, Decimal arithmetic unit.

**Unit – IV :**

**Input-output Organisation :** Peripheral Devices, I/O interface, Data transfer, DMA, Interrupts, I/O Processor, Multiprocessor System Organisation.

**Memory Organisation :** Anxiliary Memory, Memory Hierarchy, Associate memory, Cache Memory, Virtual Memory.

**Prescribed Book :**

Computer Organisation by Morris mano

M.C.A., First Year Degree Examination, May 2007

**Paper – III : COMPUTER ORGANIZATION**

**Time : 3 Hrs** **Max. Marks : 75 Marks**

**SECTION- A ( 3 X 15 = 45 Marks)**
**Answer any three Questions**

1. Draw expanded structure of the IAS computer. Explain partial flowchart of IAS operation.
2. Explain program flow control with interrupts and without interrupts and various cycles fetch, execute, instruction cycles.
3. Explain different levels of RAID scheme.
4. (a) How the negative members are represented memory?
   (b) Explain Booths algorithm for multiplication of two binary numbers with suitable example.
5. Write about instruction pipelining.

**SECTION-B (5 x 5 = 25 Marks)**

**Answer any FIVe Questions**

6. Discuss about IBM system/360.
7. Explain various interconnection structures ?
8. Explain Data formatting and organization on magnetic disk.
9. Explain Division of unsigned binary integers.
10. Explain Data transfer, Arithmetic and logical instructions.
11. Explain different types of control and status registers.
12. Explain about ALU
13. Explain about RISC pipelining.

**SECTION -C (5 X 1 = 5 Marks)**

**Answer all Questions**

14. What is multiprocessors and vector processing?
15. What are the contents of MBR?
16. Define WORM
17. Define normalized and denormalized numbers
18. Explain equal and overflow flag.

M.Sc, Information Technology (Previous) Degree Examination, May 2007

**Paper – III : COMPUTER ORGANIZATION**

**Time : 3 Hrs** **Max. Marks : 75 Marks**

**SECTION- A ( 3 X 15 = 45 Marks)**
**Answer any three Questions**

1. Explain the extended structures of IAS computer with a diagram.
2. Explain instruction fetch and execution with a state diagram.
3. Explain various Bus interconnection schemes.
4. Explain Booths algorithm for multiplication of signed binary numbers.
5. Explain instruction pipelining and pipelining performance measures.

**SECTION-B (5 x 5 = 25 Marks)**

**Answer any FIVe Questions**

6. Explain the key distinguish features of a micro processor.
7. Briefly explain the two approaches to dealing with multiple interrupts.
8. Explain the terms sleek time and rotational delay.
9. Explain the serpentine reading.
10. What are the common characteristics are shared by all RAID levels?
11. Explain IEEE standard for floating point representation
12. What general roles are performed by CPU registers?
13. What categories of data are generally supported by user-variable registers?

**SECTION -C (5 X 1 = 5 Marks)**

**Answer all Questions**

14. List classes of Interrupts.
15. Define access time
16. What is the typical disk sector size?
17. What is based representation?
18. What is direction flag (DF)?

P.G. Diploma Examination, May 2007

Information Technology

**Paper – III : COMPUTER ORGANIZATION**

**Time : 3 Hrs** **Max. Marks : 75 Marks**

**Answer any FIVE Questions**
**All Questions carry equal marks.**

1. Explain the various features of computer generations.
2. Explain performance improving methods.
3. Explain instruction fetch and execution with a state diagram.
4. Explain PCI bus structures.
5. Explain magnetic tape reading and writing methods.
6. Briefly explain seven RAID levels.
7. Explain various methods to represent binary numbers with examples and discuss advantages of each.
8. Explain floating point addition and subtraction with a flow chart.
9. Explain the organization of registers in CPU.
10. Briefly explain instruction pipeling and pipeline performance measures.

## P. G. Diploma Examination, May 2007

(Examination at the end of First Year)
Part II - Information Technology

## Paper – II : COMPUTER ORGANIZATION

**Time : 3 Hrs** **Max. Marks :100 Marks**

**Answer any FIVE Questions**
**All Questions carry equal marks.**

1. a) Simplify the following Boolean function in Sum-of-product form by means of four variable map. Draw the logic diagram with
   i) OR- And gates ii) Nor gates $F(A,B,C,D) = \sum (1,5,7,9,12,13,14,15)$
   b) Explain combinational circuits.
2. a) Discuss error detection codes, explain with circuit diagram.
   b) Explain Gray code and Excess-3 code with examples.
3. a) Draw the block diagram of a dual 8-to-1 line multiplexers and explain its operation by means of a function table.
   b) Illustrate the working of shift Registers.
4. a) What are binary counters? How binary counters differ with binary counter with parallel load.
   b) Explain stack organisation.
5. a) Describe micro programming with illustrations.
   b) Draw a neat flow chart for Booths multiplication.
6. a) Explain Hardwired control organization with examples.
   b) Discuss different addressing modes with examples.
7. a) Explain various instruction formats with examples.
   b) Briefly explain Bus and Memory Transfers.
8. a) Explain interrupt cycle by using flow chart. b) Write about DMA.
9. a) Explain the asynchronous data transfer using hand shaking signals
   b) What is stored program organization.
10. Briefly Explain :
    a) Floating point representation b) Cache memory.
    c) Virtual memory d) Reserve polish notation.

# CONTENT

| | | |
|---|---|---|
| Lesson - I | Digital Logic Circuits | 1.1 - 1.27 |
| Lesson - 2 | Digital Components | 2.1 - 2.12 |
| Lesson - 3 | Data Representation | 3.1 - 3.22 |
| Lesson - 4 | Register Transfer and Micro Operations | 4.1 - 4.14 |
| Lesson - 5 | Basic Computer Organization | 5.1 - 5.12 |
| Lesson - 6 | Central Processing Organization - I | 6.1 - 6.14 |
| Lesson - 7 | Central Processing Organization - II | 7.1 - 7.17 |
| Lesson - 8 | Arithenetic Processor Design and Arthmetic Algorithums - I | 8.1 - 8.18 |
| Lesson - 9 | Arithmetic Processor Design and Arthmetic Algorithums - II | 9.1 - 9.14 |
| Lesson - 10 | Input - Out put Organization | 10.1 - 10.17 |
| Lesson - 11 | Memory Organization | 11.1 - 11.23 |

Lesson -1

# Digital Logic Circuits

## 1.0 OBJECTIVE

In this lesson we introduce the fundamental knowledge needed for designing of digital systems constructed with individual gates and flip-flops. It covers Boolean algebra, Combinational Circuits and Sequential Circuits. This chapter provides necessary background for understanding the digital logic circuits.

**STRUCTURE OF THE LESSON**


- **1.1. Digital Computers**
- **1.2. Logic Gates**
- **1.3. Boolean Algebra**
- **1.4. Map Simplifications**
- **1.5. Combinational Circuits**
- **1.6. Flip-Flops**
- **1.7. Sequential Circuits**
- **1.8. Summary**
- **1.9. Model Questions**
- **1.10. References**


## 1.1. DIGITAL COMPUTERS

The **Digital Computer** is a digital system that performs various computational tasks. The word digital implies that the information in the computer is represented by variables that take a limited number of discrete values. These values are processed internally by components that can maintain a limited number of discrete states. The decimal digits 0,1,2...9. Digital Computers use the **binary number system**, which has two digits: **0 and 1**. A binary digit is called a **bit**. Information is represented in digital computers in-group of bits. A Computer system is sometimes subdivided into two functional entities **hardware** and **software**. The **hardware** of the computer consists of electronic components and electromechanical devices that comprise the physical entity of the device. Computer **software** consists of the instructions and data that the computer manipulates to perform various data-processing task. A sequence of instructions for the computer is called a **program.** The hardware of the computer is usually divided into three major parts.

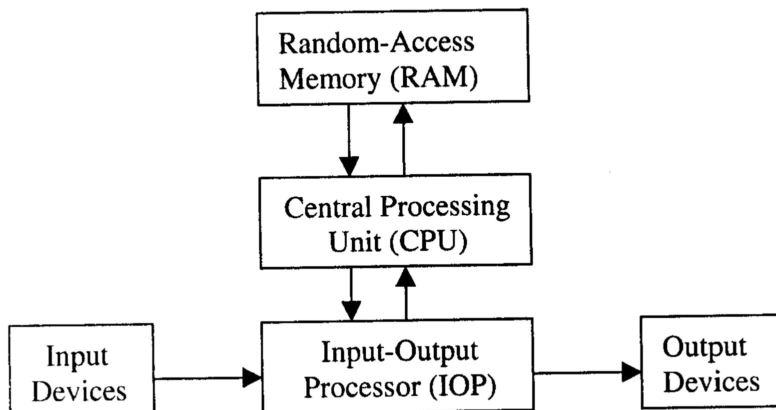



**Figure 1.1 Block diagram of a digital computer**

The central processing unit (**CPU**) contains an arithmetic and logic unit for manipulating data, a number of registers for storing data, and control circuits for fetching and executing instructions. The memory of a computer contains storage for instructions and data. It is called random-access memory (**RAM**) because the CPU can access any location in memory at random and retrieve the binary information within a fixed interval of time. The input and output processor (**IOP**) contains electronic circuits for communicating and controlling the transfer of information between the computer and the outside world. The input and output devices connected to the computer include keyboards, printers, terminals, magnetic disk drives, and other communication devices.

## 1.2. Logic Gates

A gate is an electronic circuit with one or more input signals and a single output. As the input or output is either 0 or 1. These gates are called Two State Circuits. Gates are basic building blocks of a digital system. These gates are used in digital system designing and interfacing circuits for Microprocessor based systems.

Gates are classified into Basic Gates, Universal Gates and Realized Gates.

### Basic Gates

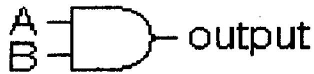
Basic Gates are classified into AND gate, OR gate and NOT gate.

### AND GATE

An AND gate consists of two or more inputs and only one output. The output of the AND gate is equal to the product of the inputs. Output is mathematically represented as Y = A.B.

**Symbolic Representation:**

Two input AND Gate




A AND B = A•B = AB

**Figure 1.2a Two input AND Gate**

Truth Table:

| Input | | Output |
|---|---|---|
| **A** | **B** | **Y** |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Table1.1 Two Input AND Gate**

The output of an AND gate is high if all the inputs are high.

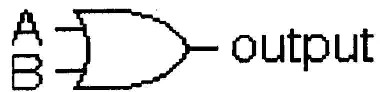
USES: The AND gate is useful for clearing, or masking, specified bit positions. AND gates are used in Interfacing Circuitry, Microcomputer designing, and Decoders.

**OR GATE**

An OR gate consists of two or more inputs and only one output. The output of an OR gate is equal to the logical OR of the inputs. It is mathematically represented as Y = A + B and read as Y is equal to A OR B.

Symbolic Representation:

Two input OR gate




A OR B = A + B

**Figure 1.2b Two input OR Gate**

Truth Table:

| Input | | Output |
|---|---|---|
| A | B | Y |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**Table1.2 Two input OR Gate**

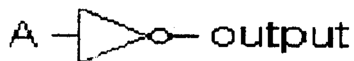
The output of an OR gate is high if the either of the inputs is high else low.

USES: OR gates are used in Decision Making Circuits, Multiplexers and Encoders. The OR gate is useful for setting specified bits.

**NOT GATE**

It consists of one input and a single output. The output of a NOT gate is the complement or reverse or inverse of given input. A NOT gate is also called as Inverter.

**Symbolic Representation**




NOT A = $\overline{A}$ = A'

**Figure 1.2c NOT Gate**

Truth Table:

| Input | Output |
|---|---|
| A | Y |
| 0 | 1 |
| 1 | 0 |

**Table1.3 NOT Gate**

**USES:** It is used to invert logic levels whenever it is required. Negative logic is widely used to design of keyboards, common Anode Display.

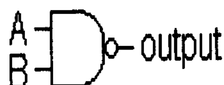
**UNIVERSAL GATES**

Universal Gates are classified into NAND and NOR gates.

## NAND GATE

A NAND gate consists of two or more inputs and only one output. The output of a NAND gate is the complement of the product of the given inputs. A NAND gate is a combination of AND and NOT gates. It is mathematically represented as y= $(A . B)'$.

**Symbolic Representation**
Two inputs NAND gate




$A \text{ NAND } B = \text{NOT } A \text{ AND } B = \overline{(A \cdot B)}$

**Figure 1.2d Two input NAND Gate**

Truth Table:

| Input | | Output |
|---|---|---|
| A | B | Y |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Table1.4 Two input NAND Gate**

The output of a NAND gate is high if either of the inputs is low. The output of the NAND gate is low if all the inputs are high.

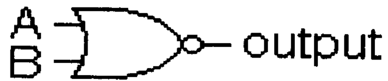
**USES:** NAND gates are used to replace other gates, as they are very cheap. NAND gates are used in the construction of Demultiplexers.

## NOR GATE

A NOR gate consists of two or more inputs and a single output. The output of a NOR gate is the complement of the OR of the inputs. A NOR gate is a combination of OR and NOT gates. Mathematical representation of NOR gate is
Y= $(A + B)'$.

**Symbolic Representation**

Two input NOR gate




$$A \text{ NOR } B = \overline{(A + B)}$$

**Figure 1.2e Two input NOR Gate**

Truth Table:

| Input | | Output |
|---|---|---|
| **A** | **B** | **Y** |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

**Table1.5 Two input NOR Gate**

The output of a NOR gate is low if at least one of the input is high else high.

USES: NOR gates are used to replace other gates, as they are very cheap.

**REALISED GATES**

60

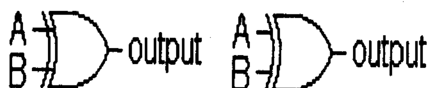
Realised Gates are classified in to X-OR and X-NOR gates.

**EXCLUSIVE OR (XOR) Gate**

It consists of two or more inputs and only one output. It is mathematically represented as Y = A' B + A B'.

Symbolic Representation

Two input X-OR Gate




$$A \text{ XOR } B = \overline{A} \cdot B + A \cdot \overline{B} = (A + B) \cdot (\overline{A} + \overline{B}) = A \oplus B$$

**Figure 1.2f Two input X-OR Gate**

Truth Table:

| Input | | Output |
|---|---|---|
| **A** | **B** | **Y** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Table1.6 Two input X-OR Gate**

The EXCLUSIVE OR gate is useful for complementing specified bits. The output of an EXCLUSIVE OR gate is low if all the inputs are same.

**USES**: X-OR gate is used in the design of Adders, Subtractors, Parity Checkers, Binary-Gray and Gray-Binary Converters. The output of X-OR gate is equal to the sum of inputs.

**EXCLUSIVE NOR (XNOR) GATE**

It consists of two or more inputs and a single output. The output of the X-NOR gate is the complement of the X-OR gate. Mathematically representation is Y=(A' B + A B')' = A' B' + AB. EXCLUSIVE NOR gate is the combination of EXCLUSIVE OR and NOT gate.

**Symbolic Representation**

Two inputs X-NOR gate

$A \text{ XNOR } B = A \cdot B + \bar{A} \cdot \bar{B} = \overline{A \oplus B}$

**Figure 1.2g Two input X-NOR Gate**

Truth Table:

| Input | | Output |
|---|---|---|
| **A** | **B** | **Y** |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

**Table1.7 Two input X-NOR Gate**

The output of an X-NOR gate is high if all the inputs are same. The output of the X-NOR gate is high when the input bits are even parity.

## 1.3. BOOLEAN ALGEBRA

The binary 0 and 1 states are naturally related to the true and false logic variables. We can define

**OR Set**

0+A=A'
1+A= 1
A+A =A
A+A'=1

**AND Set**

0.A=0
1.A=A
A.A=A
A.A'=0

**NOT Set**

(A')'=A
1'=0
0'=1

**COMMUNICATION Set**

$A + B = B + A$
$A \cdot B = B \cdot A$

**ABSORPTION Set**

$A + A \cdot B = A$
$A \cdot (A + B) = A$

**ASSOCIATION Set**

$A + (B + C) = (A + B) + C$
$A \cdot (B \cdot C) = (A \cdot B) \cdot C$

**DISTRIBUTION Set**

$A + B \cdot C = (A + B) \cdot (A + C)$
$A \cdot (B + C) = A \cdot B + A \cdot C$

Other rules are

A+A'B=A+B
A'+AB=A'+B

## DEMORGAN'S THEOREM

Demorgan's first theorem states that the output of a NOR gate is equivalent to that of a bubbled AND gate i.e.,

(A+B)'=A'. B'

(A+B+C)'=A'. B'. C'

The complement of the OR of the inputs is same as the product of the complements of the input.

LHS RHS
(A+B)' = A' . B'

Truth Table for LHS

| Input | | Output |
|---|---|---|
| A | B | (A+B)' |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

Truth Table for RHS

| Input | | | | Output |
|---|---|---|---|---|
| A | A' | B | B' | A'. B' |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |

Symbolic Representation

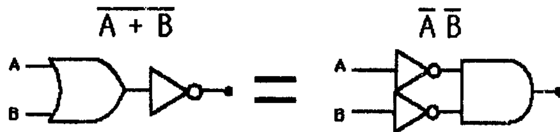



**Figure 1.3a Graphic symbol for NOR Gate**

Demorgan's second theorem states that the output of a NAND gate is equivalent to that of a bubbled OR gate i.e.

(A.B)' = A' + B'

(A.B.C)' = A' + B' + C'

The complement of the product of input is same as the sum of lthe complements of the input.

LHS RHS

(A.B)' = A' + B'

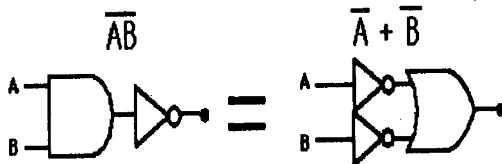
Truth Table for LHS

| Input | | Output |
|---|---|---|
| A | B | (A.B)' |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Truth Table for RHS

| Input | | | | Output |
|---|---|---|---|---|
| A | A' | B | B' | A'+ B' |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 |

Symbolic Representation




A NAND gate is equivalent to an inversion followed by an OR

**Figure 1.3b Graphic symbol for NAND Gate**

**Examples of Boolean algebra:**

**1. Y=ABC' + A'B'CD + ACD' + AC + AB**

Y=AB (C'+1) +A'B'CD+AC (D'+1)
Y=AB+A'B'CD+AC (from 1+A = 1)
Y=AB+C (A'B'D+A)
Y=AB+C (A+B'D) (from A+A'B=A+B)

**2. A+A'B+AB' = A+B**
LHS
A+A'B+AB'
A (1+B') + A'B
A+A'B (from 1+A = 1)
A+B = RHS (from A+A'B = A+B).

## DUALITY THEOREM

Positive and Negative logic give rise to a basic duality in all the identities when changing from one logic system to another. 0 becomes 1 and 1 becomes 0. Further more AND gates become OR gates and OR gates become AND gates. This means that given any Boolean identity we can produce a dual identity by changing (+) sign to (.) sign vice versa and complementing all zeros to ones.

Ex::

1. A+0=A
   A.1=A
2. A+A'=1
   A.A'=0
3. A+A=A
   A.A=A
4. A+AB=A
   A. (A+B)=A

# 1.4. MAP SIMPLIFICATION

**SUM OF PRODUCTS AND PRODUCT OF SUMS:**

**FUNDAMENTAL PRODUCTS**

There are four possible ways to AND two input signals that are in complemented or un-complemented form. The four products are A'B', A'B, AB', AB. The products are called as Fundamental Products. Similarly there are eight ways of combining three input signals with an AND gate A'B'C, A'B'C, A'BC', AB'C', A'BC, AB'C, ABC' and ABC.

Method: Whenever the input variable is zero the same input is complemented else the variable is considered as it is in the fundamental product.

ABCD = 0110, The Fundamental Product is A'BCD'.

**SUM OF PRODUCTS**

In Sum of Products method the inputs are ANDed, then ORed.
The steps for sum of products:

1. The fundamental product corresponding to each 1 output in the truth table.
2. ORing the fundamental products.
3. Write down the equation.

Example: Given a truth table you can find the Boolean equation for the output by ORing the fundamental products that on outputs.

| A | B | C | FP | Y |
|---|---|---|---|---|
| 0 | 0 | 0 | A'B'C' | 0 |
| 0 | 0 | 1 | A'B'C | 0 |
| 0 | 1 | 0 | A'BC' | 0 |
| 0 | 1 | 1 | A'BC | 1 |
| 1 | 0 | 0 | AB'C' | 0 |
| 1 | 0 | 1 | AB'C | 1 |
| 1 | 1 | 0 | ABC' | 1 |
| 1 | 1 | 1 | ABC | 1 |

The fundamental products that result in 1 output are listed. They are A'BC, AB'C, ABC' and ABC. By ORing these products, we get the Boolean equation of output as Y= A'BC + AB'C + ABC' + ABC. This equation results in a one output only for the input conditions 011, 101, 110 and 111.

The above three steps always results in a Boolean expression that is a logical sum of the logical products. The method of getting Boolean equation is called as sum of products.

## PRODUCT OF SUMS

A sum of products solution always results in a group of AND gates driving a final OR gate. This type of network is known as NAND-NAND network. There is another alternative known as Product of Sums solution, and results in a group of OR gates working into a final AND gate. Because of De Morgan's theorems, this is equivalent to NOR gates driving a NOR gate.

The steps for Product of Sums:

1. Given the truth table for Y. Get Sum of Products equation for Y'.
2. Apply De Morgan's theorems to get the product of sums equation.

Example: From the above example of Sum of Products equation is
Y'= A'BC + AB'C + ABC' + ABC
Complement both sides
(Y')'= (A'BC + AB'C + ABC' + ABC)'
Apply De Morgan's first theorem (i.e., (A + B)' = A'. B')
Y= (A'BC)' (AB'C)' (ABC')' (ABC)'
Apply De Morgan's second theorem (i.e. (AB)'= A' + B')
Y = ((A')' + B' + C') (A' + (B')' +C') (A' + B' + (C')') (A'+ B' + C')
Y= (A + B' + C') (A' + B + C') (A' + B' + C) (A' + B' +C')

## KARNAUGH MAPS

Boolean Algebra is algebra of binary variables 0 and 1. Mathematical operations in Boolean algebra are AND, OR, NOT. The main aim of using Boolean algebra is to reduce the number of basic circuits to perform a digital operation. The minimum number of circuits reduces the cost and increases the speed, efficiency and reliability of the system

.

**K-MAP** is a graphical representation of fundamental products in a truth table. Instead of using Boolean algebraic theorems for simplification we can use K-maps. The Karnaugh map can also be described as a special arrangement of a truth table.

### Construction of a Two Variable K-Map

The values inside the squares are copied from the output column of the truth table, therefore there is one square

| A | B | F |
|---|---|---|
| 0 | 0 | a |
| 0 | 1 | b |
| 1 | 0 | c |
| 1 | 1 | d |

Truth Table.

| B \ A | 0 | 1 |
|---|---|---|
| 0 | a | b |
| 1 | c | d |

F.

in the map for every row in the truth table. Around the edge of the Karnaugh map are the values of the two input variable. A is along the top and B is down the left hand side. The diagram below explains this:

| A | B | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 ← |
| 1 | 1 | 1 |

Truth Table.

| B \ A | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 ← |
| 1 | 1 | 1 |

F.

The values around the edge of the map can be thought of as coordinates. So as an example, the square on the top right hand corner of the map in the above diagram has coordinates A=1 and B=0. This square corresponds to the row in the truth table where A=1, B=0 and F=1. Note that the value in the F column represents a particular function to which the Karnaugh map corresponds

## Construction of a Three Variable K-Map

| A | B | C | F |
|---|---|---|---|
| 0 | 0 | 0 | a |
| 0 | 0 | 1 | b |
| 0 | 1 | 0 | c |
| 0 | 1 | 1 | d |
| 1 | 0 | 0 | e |
| 1 | 0 | 1 | f |
| 1 | 1 | 0 | g |
| 1 | 1 | 1 | h |

| X\YZ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | a | b | d | c |
| 1 | e | f | h | g |

Example:

| A | B | C | F |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

| X\YZ | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 |

## Construction of a Four Variable K-Map

| A | B | C | D | F |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | a |
| 0 | 0 | 0 | 1 | b |
| 0 | 0 | 1 | 0 | c |
| 0 | 0 | 1 | 1 | d |
| 0 | 1 | 0 | 0 | e |
| 0 | 1 | 0 | 1 | f |
| 0 | 1 | 1 | 0 | g |
| 0 | 1 | 1 | 1 | h |
| 1 | 0 | 0 | 0 | i |
| 1 | 0 | 0 | 1 | j |
| 1 | 0 | 1 | 0 | k |
| 1 | 0 | 1 | 1 | l |
| 1 | 1 | 0 | 0 | m |
| 1 | 1 | 0 | 1 | n |
| 1 | 1 | 1 | 0 | o |
| 1 | 1 | 1 | 1 | p |

| CD\AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | a | b | d | c |
| 01 | e | f | h | g |
| 11 | m | n | p | o |
| 10 | i | j | l | k |

Example:

| A | B | C | D | F |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

| CD\AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 1 | 1 | 1 |
| 01 | 0 | 1 | 1 | 0 |
| 11 | 1 | 1 | 1 | 0 |
| 10 | 0 | 0 | 1 | 0 |

## ENCIRCLING CONCEPT

### PAIRS

A pair means two 1's that are horizontally or vertically adjacent. Diagonally adjacent ones are worthless. They lead to no simplification. Two vertical 1's is called a Vertical Pair and two horizontal 1's is called a Horizontal pair. A pair always eliminates one variable ($2^1$).

### QUAD

A quad is a group of four 1's that are end to end or group of four 1's in the form of a square. A group of four 1's horizontally is called a Horizontal Quad.

| 1 | 1 | 1 | 1 |
|---|---|---|---|

A group of four 1's vertically is called as a Vertical quad.

| 1 |
|---|
| 1 |
| 1 |
| 1 |

A group of four ones in the form of a square is called a Square quad.

| | |
|---|---|
| 1 | 1 |
| 1 | 1 |

**OCTET**

1 1

1 1

An Octet is a group of eight 1's may be in the form of two adjacent rows or adjacent columns. An octet eliminates three variables and their complements. ($2^3$)

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |

| | |
|---|---|
| 1 | 1 |
| 1 | 1 |
| 1 | 1 |
| 1 | 1 |

1 1

1 1

1 1

1 1

Note: Encircle the Octets first, Quads second, Pairs third and then lastly individual circles in the case of three variables.

**HEX**

Group of 16 ones in a K-map is known as Hex. This eliminates all four variables and their complements ($2^4$).

| | | | |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 |

Reduction of Boolean Equations

1. Enter 1's in the K-map for each fundamental product that produces a one output in the truth table. Enter 0's in the remaining places.
2. Encircle the HEX, OCTET, QUAD and PAIRS. Remember to overlap and role to get the largest possible groups.
3. If any isolated 1 remains encircle each.
4. Review the groups and eliminates any redundant groups.
5. Write the Boolean equation by ORing the products corresponding to the encircling groups.

**DON'T CARE CONDITIONS**

1. Don't Cares are represented by 'X'. Enter 1's, 0's on the K-map that produces output 1's, 0's in the truth table and enter X's for the forbidden inputs i.e., those that cannot occur during normal operation.
2. Encircle the actual 1's on the K-map in the largest groups you can find by treating the don't cares as 1's.
3. After the actual 1's have been included in the groups discard the remaining don't cares by visualizing them as zeros.

**Example:**

**f (ABCD) = Σ(0, 1, 4, 5, 6, 7, 10, 12, 14, 15)**

| CD \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 1 | 0 | 0 |
| 01 | 1 | 1 | 1 | 1 |
| 11 | 1 | 0 | 0 | 1 |
| 10 | 0 | 0 | 0 | 1 |

| CD \ AB | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 1 | 0 | 0 |
| 01 | 1 | 1 | 1 | 1 |
| 11 | 1 | 0 | 0 | 1 |
| 10 | 0 | 0 | 0 | 1 |

**The simplified equation is AC + CD+ BD+ABC**

**The logic diagram for the above equation is**

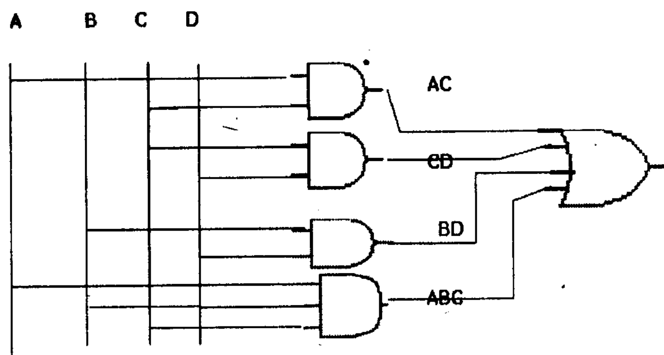



**Figure 1.4 logic diagram for above equation**

## 1.5. COMBINATIONAL CIRCUITS

A combinational circuit is a connected arrangement of logic gates with a set of inputs and outputs. Any combinational circuit output entirely depends upon input present at that moment. It consists of logic gates whose output at any time is determined directly from the present combination of inputs without regard to past input. A combinational circuit performs information logically by a set of Boolean functions. Combinational circuits are employed in digital

computers for generating binary control decisions and for providing digital components required for data processing.

Ex: - Adders, subtractor, Multiplexer, Decoders etc.,

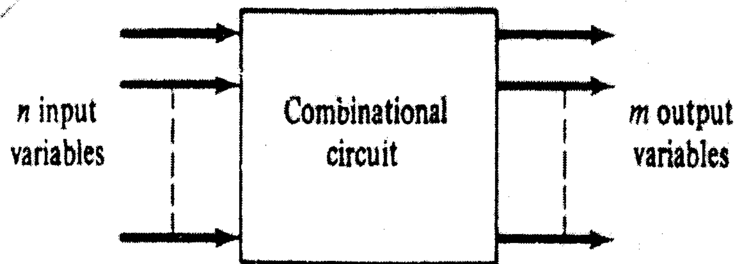



**Figure 1.5 Block diagram of a combinational circuit**

Steps involved in the design of combinational circuits:

1. The problem is stated.
2. The number of available input variables and required output variables is defined.
3. The input and output variables assigned letter symbols.
4. The truth table, which gives the required relationship between input and output variables, is derived.
5. Simplified Boolean expression for each output is obtained by K-maps.
6. The logic diagram is designed.

Before going to design a combinational circuit, consider the following points.

1. The minimum number of gates.
2. Use minimum number of inputs to a gate.
3. Use minimum propagation time of the signal through the circuit.
4. The minimum numbers of inter connections.

**Half-Adder**

The most digital arithmetic circuit is the addition of two binary **digits. A** Half-Adder is a combinational circuit that adds two binary bits. It consists of two inputs **assigned** to variables **A** and **B** and two outputs **sum** and **carry.**

Truth Table

| Input | | Output | |
|---|---|---|---|
| **A** | **B** | **Sum** | **Carry** |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

The simplified Boolean functions for the two outputs can be obtained from K- map. The simplified sums of products equations are

Sum = A'B + AB' =A XOR B = $A \oplus B$

Carry = AB.

The logic diagram for sum and carry is as follows:

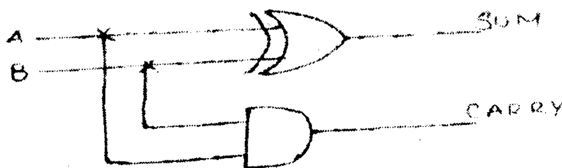



**Figure 1.6 logic diagram for Half Adder**

**Full Adder**

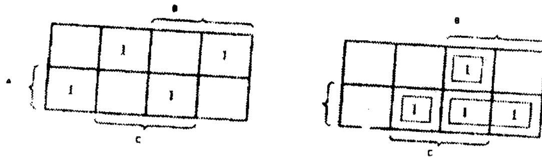
A full adder is a combinational circuit that adds three binary bits. It consists of three inputs assigned to the variables A, B, C and gives two outputs Sum and Carry. A full adder is a combination of two half adders.

Truth Table

| Input | | | Output | |
|---|---|---|---|---|
| A | B | C | Sum | Carry |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

The values of the output variables are determined from the arithmetic sum of the input bits. When input bits are 0, the output is 0. The Sum output is equal to 1 when only one input is equal to 1 or when all three inputs are equal to 1. The Carry output has a carry of 1 if two or three inputs are equal to 1.

**K-map for sum and carry**



The maps of a three variable K-map are used to find algebraic expressions for the two output variables. The 1's in the squares for the maps of sum and carry are determined directly from the minterms in the truth table. The squares with 1's for the sum output do not combine in groups of adjacent squares. The simplified equation for sum is

Sum = A'B'C + A'BC' + ABC + AB'C'

The squares with 1's for the carry output may be combined in a variety of ways. The possible expression for carry is

Carry = BC + AB + AC.

The logic diagram for full adder

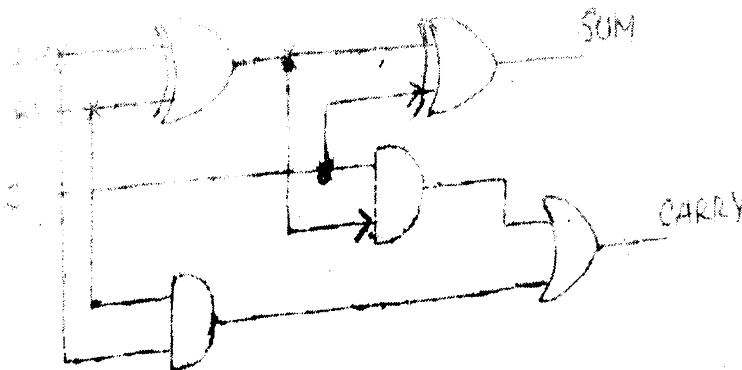



**Figure 1.7 logic diagram for full adder**

## 1.6. FLIP FLOPS

Every digital system is likely to have a combinational circuit. Most systems encountered in practice also include storage elements. The output of a sequential circuit depends upon the present input and the past output present at that particular moment. The most common type of sequential circuit is the synchronous type. Synchronization is achieved by clock generator, which generates clock pulses. These pulses are generated through out the system in such a way that memory elements are activated with the pulse.

A Flip-Flop is a basic binary storage device. It is s basic memory element of a computer memory organization. It can store binary bit either 0 Or 1. It has the property to remain in one state indefinitely until it is directed by an input signal to switch over to the other state. So, it is called as Bi-stable Multi-vibrator.

**SR Flip-Flop**

SR flip-flop consists of two NOR gates G1 and G2. The cross-coupled connection from the output of one output the input of another gate gives a feed back path. Each flip-flop circuit has two output values Q and Q' and two inputs set S and reset R. This type of flip-flop is called a Latch Circuit. Apply 1 to S and 0 to R. If one of the inputs of an NOR gate is high the output is low. So, as one of the inputs to gate G2 is high, the output of G2 is 0. This is a fed back to gate G1. Now both inputs to G1 are low, so the output is high. Now the flip-flop is said to be in SET state.

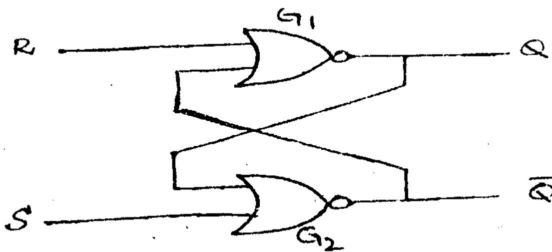
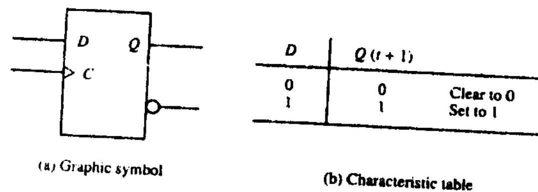



**Figure 1.8 SR flip-flop**

Apply 0,0 to R and.S. The previous output of the Flip-flop is high. Due to the feed back path the previous output Q=1 is fed back to G2. So the output of G2 is low. By taking this as input G1 output is high. It is same as previous output. So when S=R=0 the state of flip-flop is unchanged i.e., same as previous output. Neglect the previous output now and give the inputs S=0 and R=1. One of the inputs to G1 is high. So the output of G1 is low. By taking this as input the output of G2 is 1. Then the flip-flop is said to be in Reset state as the output Q=0. Now apply S=R=1. In this state G2 has one high input so the output is zero. Taking this as input the output of G1 is 0. Now the output of both Q and Q' are zero, which is absurd. This condition is said to be racing condition. In this condition the flip-flop may switch on to 1 or 0 depending upon the faster gate.

**D Flip-flop**

A D flip-flop is a slight modification of the SR flip-flop. An SR flip-flop is converted to a D flip-flop by inserting an inverter between S and R and assigning the symbol D to the single input. The D input is sampled during the occurrence of a clock transition from 0 to 1. If D = 1, the output of the flip-flop goes to the 1 state. If D = 0 the output of the flip-flop goes to the 0 state. The graphic symbol and characteristic table of the D flip flop are shown in the diagram.




| D | Q (t + 1) | |
|---|---|---|
| 0 | 0 | Clear to 0 |
| 1 | 1 | Set to 1 |

(b) Characteristic table

**Figure 1.9 D flip-flop**

From the characteristic table we note that the next state Q (t+1) is determined from the D input. The relationship can be expressed by characteristic equation Q (t+1) = D. This means that the Q output of the flip-flop receives it value from the D inputs every time that the clock signal goes through a transition from 0 to 1.

**JK Flip-flop**

In JK flip-flop inputs J and K behave like inputs of S and R in SR flip-flop. In SR flip-flop if S=R=1the flip-flop output is unpredictable or indeterminate or racing. But in the case of JK flip-flop if J=K=1 the state is permissible. In this case the output state is complement of previous state available. In the previous state Q=0, it becomes 1 and vice versa i.e., it can be written as $Q_{t+1}= Q_t$. An SR flip-flop is converted to JK flip-flop by making

J=Q'. S. Clk
K=Q. R. Clk

JK flip-flop functions similar to SR flip-flop except when both the inputs are high. Here instead of racing the complement of past output is displayed.

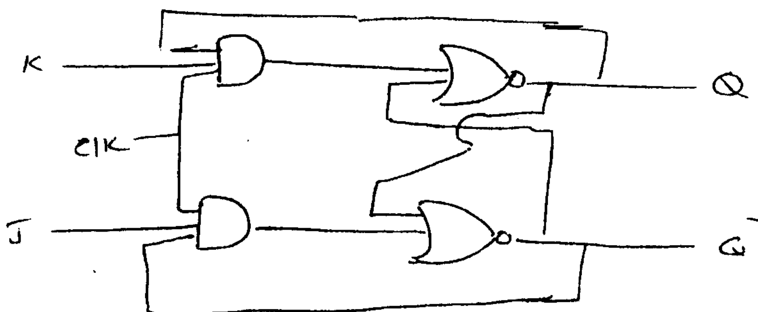



**Figure 1.10 JK Flip Flop**

**Edge-Triggered Flip-flops**

The most common type of flip-flop used to synchronize the state change during a clock pulse transition is the edge-triggered flip-flop. In this output transitions occur at a specific level of the clock pulse. When the pulse input level exceeds this threshold level, the inputs are locked out so that the flip-flop is unresponsive to further changes in inputs until the clock pulse returns to 0 and another pulse occurs. Some edge-triggered flip-flops cause a transition on the rising edge of the clock signal (positive-edge transition), and others cause a transition on the falling edge (negative-edge transition).

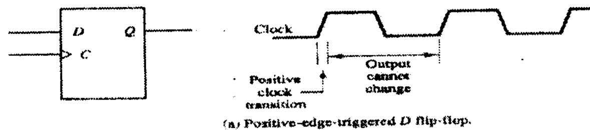



**Figure 1.11a Positive edge-triggered D flip-flop**

The above diagram shows the clock pulse signal in a positive-edge triggered D flip-flop. The value in the D input is transferred to the Q output when the clock makes a positive transition. The output cannot change when the clock is in the 1 level, in the 0 level, or in a transition form the 1 level to the 0 level. The effective positive clock transition includes a minimum time called the setup time in which the D input must remain at a constant value before the transition and a definite time called the hold time in which the D input must not change after the positive transition. The effective positive transition is usually a very small fraction of the total period of the clock pulse.

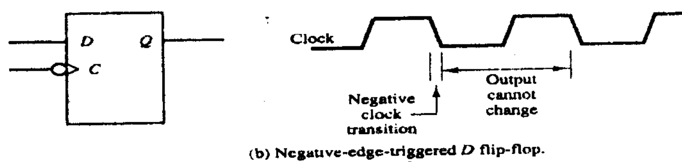



**Figure 1.11b Negative edge-triggered D flip-flop**

The above diagram shows the graphic symbol and timing for a negative-edge triggered D flip-flop. The graphic symbol includes a negation small circle in front of the dynamic indicator at the C input. This denotes a negative-edge triggered behavior. In this case the flip-flop responds to a transition from the 1 level to the 0 level of the clock signal.

**Excitation Tables**

The excitation table tells the designer the minimum inputs that are necessary to generate a particular next state when the current state is known.

Because the "input" is the current state = 1 bit, and the output is the "next state" = 1 bit, excitation tables can be derived from 4-square Karnaugh maps. But it's often easier just to look at the possible solutions.

**Flip-Flop Tables**

| D Flip-Flop | | | | | | |
|---|---|---|---|---|---|---|
| D | | $Q^+$ | | Q $Q^+$ | | D |
| 0 | | 0 | | 0 0 | | 0 |
| 1 | | 1 | | 0 1 | | 1 |
| | | | | 1 0 | | 0 |
| | | | | 1 1 | | 1 |

| T Flip-Flop | | | | | | |
|---|---|---|---|---|---|---|
| T | | $Q^+$ | | Q $Q^+$ | | T |
| 0 | | Q | | 0 0 | | 0 |
| 1 | | Q' | | 0 1 | | 1 |
| | | | | 1 0 | | 1 |
| | | | | 1 1 | | 0 |

| SR Flip-Flop | | | | | | | |
|---|---|---|---|---|---|---|---|
| S R | | $Q^+$ | | Q $Q^+$ | | S | R |
| 0 0 | | Q | | 0 0 | | 0 | x |
| 0 1 | | 0 | | 0 1 | | 1 | 0 |
| 1 0 | | 1 | | 1 0 | | 0 | 1 |
| 1 1 | | ? | | 1 1 | | x | 0 |

**JK Flip-Flop**

| JK | $Q^+$ | $QQ^+$ | J | K |
|---|---|---|---|---|
| 0 0 | Q | 0 0 | 0 | x |
| 0 1 | 0 | 0 1 | 1 | x |
| 1 0 | 1 | 1 0 | x | 1 |
| 1 1 | Q' | 1 1 | x | 0 |

Each table consists of two columns, Q and $Q^+$ and a column for each input to show how the required transition is achieved. There are four possible transitions from present state Q to next state $Q^+$. The required input conditions for each of these transitions are derived from the information available in the characteristic tables. The symbol **X** in the tables represents a don't care conditions; that is, it does not matter whether the input to the flip-flop is 0 or 1. The reason for the don't care conditions in the excitation tables is that there are two ways of achieving the required transition. For example, in a JK flip-flop, a transition from present state of 0 to a next state of 0 can be achieved by having inputs J and K equal to 0 or by letting J = 0 and K =1 to clear the flip-flop. In both cases J must be 0, but K is 0 in the first case and 1 in the second. The required transition will occur in either case, we mark the K input with a don't care **X** and let the designer choose either 0 or 1 for the K input, whichever is convenient.

## 1.7. SEQUENTIAL CIRCUITS

Digital electronics is classified in two; one is combinational logic and second is sequential logic. Combination logic's output depends on the inputs levels. Where as output of sequential logic depend stored levels and also the input levels.

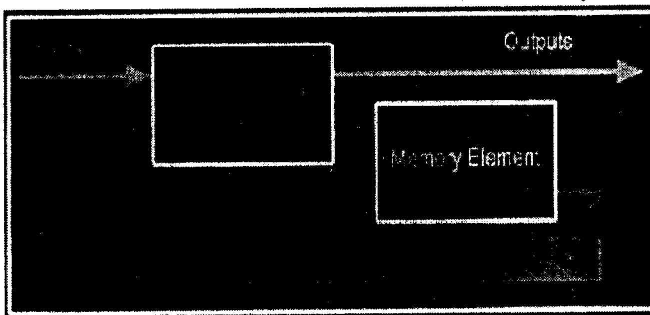



**Figure 1.12 Block diagram of a sequential Circuits**

The memory elements are devices capable of storing binary info. The binary info stored in the memory elements at any given time defines the state of the sequential circuit. The input and the present state of the memory element determine the output. The next state of the memory elements is also a function of the external inputs and the present state. A sequential circuit is specified by a time sequence of inputs, outputs, and internal states.

**Sequential Circuits Design**

We saw in the combinational circuits section how to design a combinational circuits from the given problem. We convert the problem into truth table, then draw K-map for the truth table, and then finally draw the gate level circuit for the problem. Similarly we have a flow for the sequential circuit design. The steps are as given below.

1. Excitation equations
2. Next state equations
3. Output equations
4. State table
5. State diagram

Looks like sequential design flow is very much same as combinational cir

Example of a sequential circuit

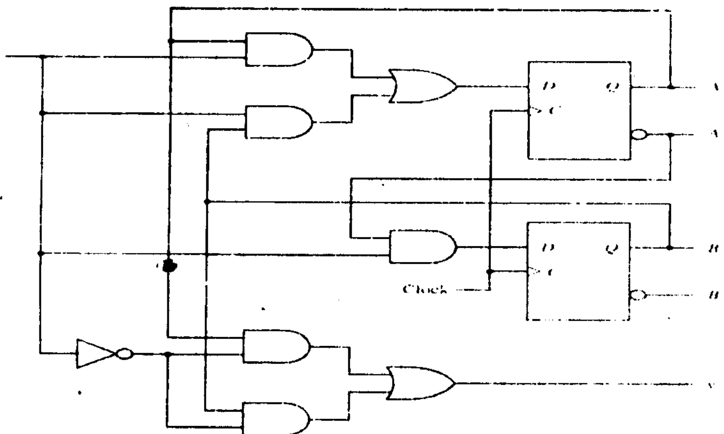


**Figure 1.13 Example of a sequential circuit**

It has one input variable x, one output variable y, and two clocked D flip-flops. The AND gates, OR gates, and inverter from the combinational logic part of the circuit. The interconnections among the gates in the combinational circuit can be specified by a set of Boolean expressions. The part of the combinational circuit that generates the inputs to flip-flops is described by a set of Boolean expressions called flip-flop input equations. We have two input equations, designated $D_A$ and $D_B$. The first letter in each symbol denotes the D input of a D flip-flop. The subscript letter is the symbol denotes the D input of a D flip-flop. The first input equations as

$D_A = Ax + Bx$

Where A and B are the outputs of the two flip-flops and x is the external input. The second input equation is derived from the single AND gate whose output is connected to the D input of the flip-flop B:

$D_B = A'x$

The sequential circuit also has an external output, which is a function of the input variable and the state of the flip-flops. This output can be specified algebraically by the expression

$Y = A'x + B'x$

**State Table**

The behavior of a sequential circuit is determined from the inputs, the outputs and the state of its flip-flops. Both the outputs and the next state are a function of the inputs and the present state. A sequential circuit is specified by a state table that relates outputs and the next states as a function of inputs and present states. In clocked sequential circuits, the transition from present state to next state is activated by the presence of a clock signal. The state table for the above diagram is as follows:

**State Table for the above Circuit**

| Present State | | Input | Next State | | Output |
|---|---|---|---|---|---|
| A | B | X | A | B | Y |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

The present-state section shows the states of flip-flops A and B at any given time t. The input section gives a value of x for each possible present state. The next-state section shows the states of the flip-flops one clock period later at time t+1. The output section gives the value of y for each present state and input condition. The derivation of a state table consists of first listing all possible binary combinations of present state and inputs. In this case we have eight binary combinations form 000 to 111. The next-state values are then determined from the logic diagram or from the input equations. The input equation for flip-flop A is

$D_A = Ax + Bx$

The next-state value of a each flip-flop is equal to its D input value in the present state. The transition from present state to next state occurs after application of a clock signal. Therefore,

the next state of A is equal to 1 when the present state and input values satisfy the conditions Ax=1 or Bx=1, which makes DA equal to 1. Similarly, the input equation for flip-flop B is

$D_B = A'x$

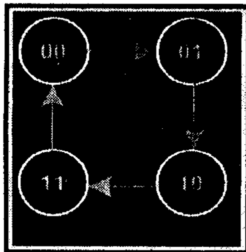
The next state of B in the state table is equal to 1 when the present state of A is 0 and input x is equal to 1. The output column is derived from the output equation

$Y = A'x + B'x$

**State Diagram**

State diagram is constructed using all the states of the sequential circuit in question. It builds up the relationship between various states and also shows how inputs affect the states. For the easy of following the tutorial, lets consider of designing the 2 bit up counter (Binary counter is one which counts binary sequence) using T flip-flop Below is the state diagram of 2-bit binary counter using T flip-flop.

Bellow is the state diagram of 2-bit binary counter.

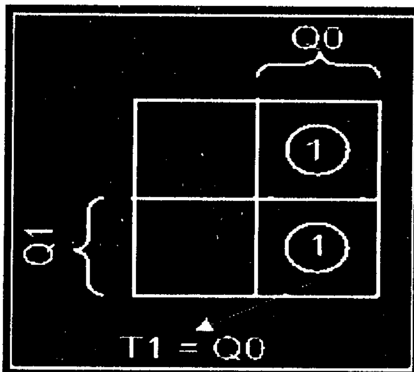


**State Table**

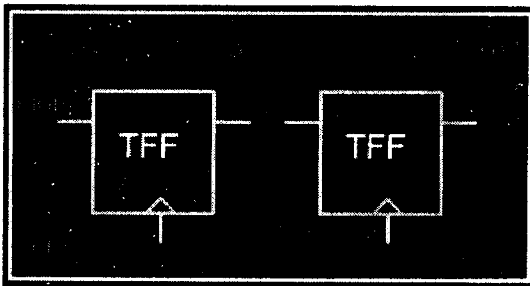
State table is same as the excitation table of a flip-flop, i.e. what inputs needs to be applied to get the output that is required. In other words this table gives the inputs that are required to produce the specific outputs.

| Q1 | Q0 | Q1+ | Q0+ | T1 | T0 |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | [illegible] | [illegible] |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | [illegible] | 1 |
| 1 | 1 | 0 | 0 | 1 | 1 |

Drawing K-map is same as the combinational circuits K-maps. Only difference here is, we draw K-map for the inputs i.e. T1 and T0 in above table. From the table we deduct that we don't need to draw K-map for T0, as it is high for all the state combinations. But for T1 we need to draw K-map as shown below using SOP.




There is nothing special of drawing circuits; it is same as any circuit drawing from K-map output. Below is the circuit of 2-bit up counter using T flip-flop.




## 1.8. Summary

In this lesson you learnt the basic functions of Gates. In this chapter we introduced the concept of Boolean Algebra and Map Simplification mechanism. We have learnt the steps to design a Combinational Circuits and Sequential Circuits. In this chapter we have learnt the importance of excitation table.

## 1.9. MODEL QUESTIONS

1. Define gate? Explain in detail about basic, Universal gates and Realized gates?
2. Simplify the following Boolean functions using three-variable maps.
   $F(x, y, z) = \Sigma(1,2,3,6,7)$
3. Explain in detail about JK flip-flop
4. Using DeMorgan's theorem, show that
   $(A+B)' = (A'+B')' = 0$
   $A+A'B+A'B'=1$
5. Define Combinational Circuit? Explain about Full Adder?
6. Explain the steps to design a Sequential Circuit?

## 1.10. REFERENCES

1. Computer System Architecture – M. Morris Mano
2. Digital Design – M. Morris Mano

**Sri. C.V.P.R.PRASAD, M.C.A.,**

Lesson 2

# Digital Components

## 2.0 Objective

In this lesson it explains in detail the logical operation of the most common standard digital components. It includes Decoders, Multiplexers, Registers, Counters and Memories. These digital components are used as building for the design of larger units.

## Structure of the Lesson


2.1. Integrated Circuits

2.2. Decoders

2.3. Multiplexers

2.4. Registers

2.5. Binary Counters

2.6. Memory Unit

2.7. Summary

2.8. Model Questions

2.9. References


## 2.1. INTEGRATED CIRCUITS

**Integrated Circuit: Integrated circuit (IC)** or often referred to as a **microchip** or simply **chip,** is a miniaturized electronic circuit on the surface of a thin substrate of semiconductor material.

**Integrated circuits** are used for a variety of devices including audio and video equipment, automobiles, and microprocessors. **Integrated circuits are often classified** according to the number of transistors and other electronic components they contain, including SSI, MSI, LSI, and VLSI. A **basic integrated circuit** is often referred to as a microchip or simply a chip. Essentially it is a miniaturized electronic circuit that has been manufactured on a thin wafer of semiconductor. **Types of integrated circuits** include analog, digital, and mixed signal chips. The growth of **complexity of integrated circuits** follows Moore's Law, which states that the number of transistors in an integrated circuit doubles every two years. **Integrated circuits are fabricated** in a layer process, which includes imaging, deposition and etching. The **earliest integrated circuits** were packaged in ceramic flat packs, which continued to be used by the military for their reliability and small size for many years. The number of pins may range from 14 in a small IC package to 100 or more in a larger package. As the technology of IC' s has improved, the number of gates that can be put in a single chip has increased considerably. The differentiation between those chips that have a few internal gates and those having hundreds or

thousands of gates is made by a customary reference to a package as being either a small-, medium- or large-scale integration device.

**Small-Scale Integration (SSI):**

Small-Scale Integration (SSI) ICs had small number of devices on a single chip – diodes, transistors, resistors and capacitors, making it possible to fabricate one or more logic gates on a single device.

**Medium–Scale Integration (MSI):**

Medium – Scale Integration (MIS) devices contain hundreds of transistors on each chip

**Large-Scale Integration (SSI):**

Large-Scale Integration (LSI) contain tens of thousands of transistors per chip.

LSI circuits began to be produced in large quantities around 1970, for computer main memories and pocket calculators.

**Very Large-Scale Integration (VLSI):**

Very Large-Scale Integration (VLSI), with hundreds of thousands of transistors, and more.

Digital integrated circuits are classified not only by their logic operations but also by the specific circuit technology to which they belong. The circuit technology is referred to as a digital logic family.

The types of logic devices are classified in "families", of which the most important are TTL and CMOS. The main families are:

- TTL (Transistor-Transistor Logic) made of bipolar transistors.
- CMOS (Complementary Metal Oxide Semiconductor) made from MOSFET's.
- ECL (Emitter Coupled Logic) for extremely high speeds.
- MOS (Metal-oxide semiconductor) for circuits that need high component density.

## 2.2. DECODER

A decoder is a combinational circuit that converts binary information from n input lines to $2^n$ output lines. Only one of the outputs is active at any given time, based on the value of the n select lines. The purpose of a decoder is to generate less than or equal to $2^n$ minterms of n input variables. In the following 3x8 decoder, the 3 select lines $A_2$, $A_1$, $A_0$ determine which of the 8 outputs is active. Only one of the outputs is active for any given input combination:

| INPUT | | | OUTPUT | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $A_0$ | $A_1$ | $A_2$ | $D_0$ | $D_1$ | $D_2$ | $D_3$ | $D_4$ | $D_5$ | $D_6$ | $D_7$ |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

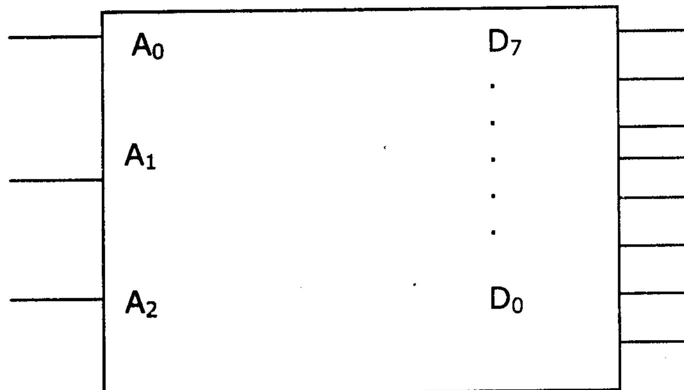



**Figure 2.1 Block diagram of a Decoder**

Decoders can be used as code detectors.

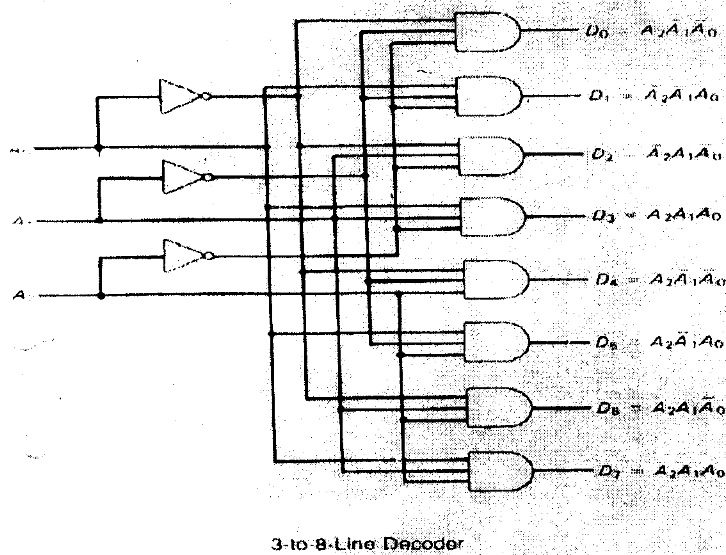



**Figure 2.2 3 to 8 line Decoder**

## 2.3. MULTIPLEXER

A Multiplexer typically has n select lines, $2^n$ input lines, and 1 output. Only one of the inputs is gated through to the output, based on the value of the n select lines. In the following 4x1 multiplexer, you are given 2 control lines ($S_1$, $S_0$) and $2^2$ = 4 input lines ($I_0..I_3$). The output is selected based upon the value of the control lines S1 S0: A multiplexer is also called a "data selector". A multiplexer can be implemented with an n-to-2n Decoder with the addition of $2^n$ input lines.

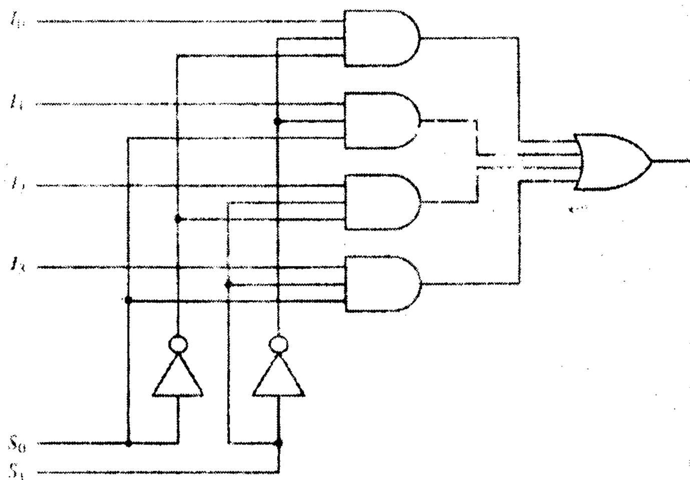


**Figure 2.3 4-to-1-line multiplexer**

The relation demonstrates the relationship between the four data inputs and the single output as a function of the selection inputs $S_1$ and $S_0$.

When the selection inputs are equal to 00, output Y is equal to input $I_0$. When the selection inputs are equal to o1, output Y is equal to input $I_1$.
When the selection inputs are equal to 10, output Y is equal to input $I_2$. When the selection inputs are equal to $I_1$, output Y is equal to input $I_3$.

Function Table for 4 to 1 Line Multiplexer

| Select | | Output |
|---|---|---|
| $S_1$ | $S_0$ | Y |
| 0 | 0 | $I_0$ |
| 0 | 1 | $I_1$ |
| 1 | 0 | $I_2$ |
| 1 | 1 | $I_3$ |

## 2.4. REGISTERS

A binary 0 or 1 is called a binary bit. A bit can be stored in a Flip-flop. A flip-flop is a basic memory element. A set of n number of bits is said to be a binary word. Ex:- 10110 is a set of five bits. Hence it is called a five bit binary word. A flip-flop is used to store a binary bit where as a register is used to store a binary word. A register is a set of flip-flops, which is used to store a n bit binary word.
The word length of a registers depends upon the number of flip-flops, it has. A register containing n flip-flops is said to be an n-bit binary register. It will modify the stored word by shifting its bits left or right. All the flip-flops are connected generally in parallel with a clock pulse.

**Shift Register:**

A register loads the data by shifting its contents within itself, without changing the order of bits. The data is shifted one bit at a time when the clock pulse is applied. In a shift register the output of one flip-flop is connected to the input of another flip-flop. The logical configuration of a shift register consists of a chain of flip-flops in cascade, with the output of one flip-flop connected to the input of the next flip-flop. All flip-flops receive common clock pulses that initiate the shift from one stage to the next. The output of a given flip-flop is connected to the D input of the flip-flop at its right. The clock is common to all flip-flops. The serial input determines what goes into the leftmost position during the shift. The serial output is taken from the output of the rightmost flip-flop.

**Bi-directional Shift Register With Parallel Load:**

A register capable of shifting both right and left is called a bi-directional shift register. Now if this same register has both bi-directional shift and parallel-load capabilities, it is called a bi-directional shift register with parallel load.
The circuit diagram below performs this above-mentioned operation.

A brief description of the operation of the above circuit is as follows:

1. A "clear" control is used to clear the register to 0.
2. A "clock" input for clock pulses to synchronize all operations.
3. Two mode control signals " S0 & S1" are used to control the mode of operations, as per the table.

**Functional Table for Register**

| Mode Control | | Register Operation |
|---|---|---|
| $S_1$ | $S_0$ | |
| 0 | 0 | No Change |
| 0 | 1 | Shift Right (down) |
| 1 | 0 | Shift Left (up) |
| 1 | 1 | Parallel Load |

4. A serial input associated with shift-right operation mode.
5. A serial input associated with shift-left operation mode.
6. A parallel-load with 4 input lines I4, I3, I2, and I1 with I1 and I4 being LSB & MSB respectively.
7. 4 parallel output lines A4, A3, A2, and A1 with A1 and A4 being the LSB & MSB respectively.

A 4-bit bidirectional shift register with parallel load is shown in figure. Each stage consists of a D Flip-flop and a 4 X 1 multiplexer. The two selection inputs $S_1$ and $S_0$ select one of the multiplexer data inputs for the D flip-flop. The selection lines control the mode of operation of the register according to the function table. When the mode control $S_1S_0$=00, data input 0 of each multiplexer is selected. This condition forms a path from the output of each flip-flop into the input of the same flip-flop. The next clock transition transfers into each flip-flop the binary value it held previously, and no change of status occurs. When $S_1S_0$=01, the terminal marked 1 in each multiplexer has path to the D input of the corresponding flip-flop. This causes a shift-right operation, with the serial input data transferred into flip-flop $A_0$=10 and the content of each flip-flop $A_{i-1}$ transferred into flip-flop $A_i$ for i=1,2,3. When $S_1S_0$=10, a shift –left operation results, with the other serial input data going into flip-flop A3 and the content of flip-flop $A_{i+1}$ transferred into flip-flop $A_i$ for i=1,2. When $S_1S_0$=11, the binary information from each input $I_0$ through $I_3$ is ...sferred into the corresponding flip-flop, resulting in a parallel load operation.

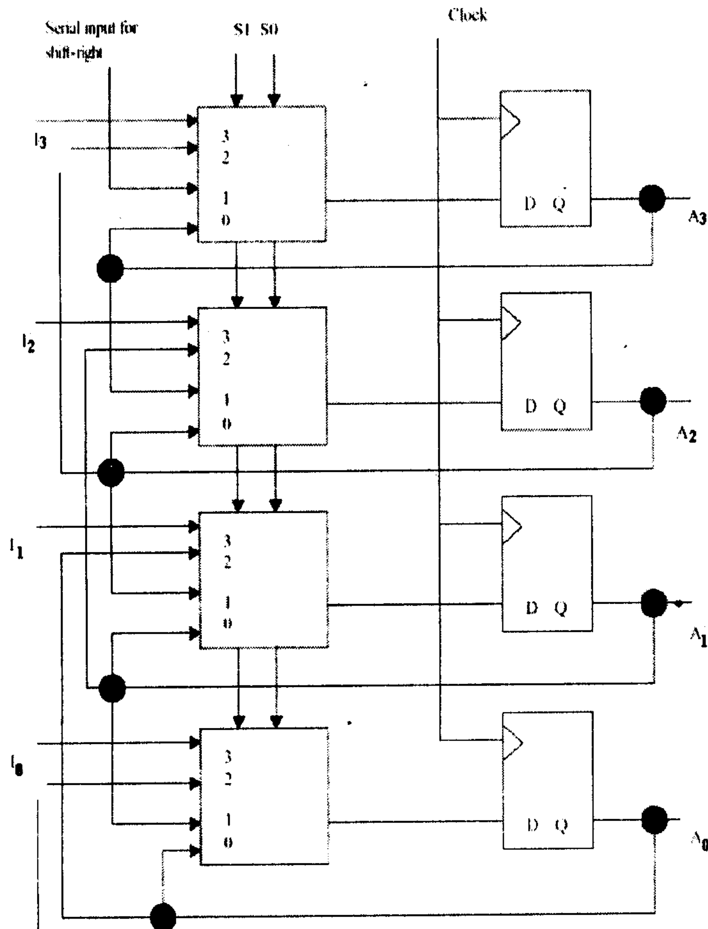



**Figure 2.4 Bi-directional Shift Register**

## 2.5. BINARY COUNTERS

A counter is a register capable of counting the number of clock pulses that have arrived at its clock input. In its simplest from it is the electronic equivalent of the binary counter. Generally counters are built with J-K flip-flops.

**Synchronous 4 – bit Binary Counter:**

A 4-bit synchronous binary counter counts the clock pulses from 0000 to 1111. It consists of 4 T Flip-flops or 4 J-K flip-flops $F_0$, $F_1$, $F_2$, and F3. If J-K is used, we can convert to T flip-flop by joining J and K terminals. These flip-flops are negative edge triggered and connected synchronously to clock line and reset line. $F_0$ Flip-flop is connected to high voltage. The output of the counter is from $Q_A$, $Q_B$, $Q_C$, and $Q_D$, where as the inputs is $T_A$, $T_B$, $T_C$, and $T_D$. $Q_A$ drives $T_B$, $Q_B$ and $Q_A$ drives $T_C$ with respect to an AND gate. $Q_A$, $Q_B$ and $Q_C$ drives $T_D$ through an AND gate.

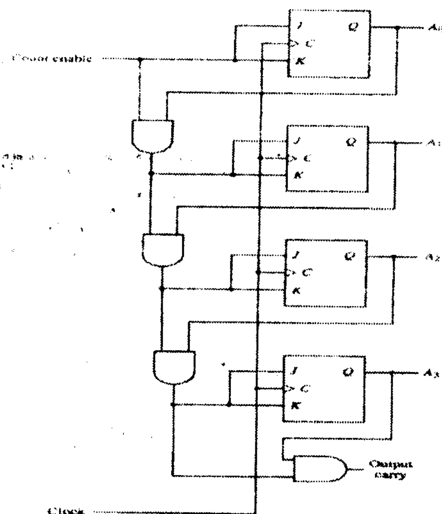


**Figure 2.5 4-bit synchronous binary counter**

Functioning: The counter is made to reset by putting 0 on CLR line and then making high. A Clock pulse will drive all flip-flops simultaneously in its trailing edge, as they are synchronously connected. The LSB flip-flop responds to every negative edge where as remaining flip-flops will respond the negative edge under certain conditions. $F_2$ flip-flop toggles when $Q_A$ and $Q_B$ are 1's. $F_3$ toggles when $Q_C=Q_B=Q_A=1$. All flip-flops toggle on the next negative edge.

Summary:

1. A low clear line resets the flip-flops to 0000.
2. As soon as clear line is high, the counter is ready to work.
3. The first negative clock edge sets QA to 1. The output word is 0001 since QA is high F1is able to toggle in the second clock edge.
4. When the second clock edge arrives QA and QB simultaneously toggle giving the output 0019.
5. The third clock edge advances count to 0011.
6. Because QA and AB are high in the next clock pulse F2 toggles.
7. In the fourth negative edge of clock pulse, F0 and F2 toggle giving output 0100.
8. Similarly in the fifteenth negative edge of the clock pulse output word is 1111.
9. The Sixteenth negative edge if given changes the output word to 0000 and the process starts again.

## 2.6. MEMORY UNIT

A memory unit is a collection of storage cells together associated circuits needed to transfer information in and out of storage. The memory stores information in groups of bits called words.

Word: A word in memory is an entity of bits that move in and out of storage as a unit. A memory word is a group of 1's and 0's and may represent a number, an instruction code, one or more alphanumeric characters, or any other binary-coded information.

Byte: A group of eight bits is called a byte. Most computer memories use words whose number of bits is a multiple of 8.
16-bit word contains two bytes.
32-bit word contains four bytes.

Each word in memory is assigned an identification number, called an address, starting from 0 and continuing with 1, 2, 3, up to $2^k$-1 where k is the number of address lines. Applying the k-bit binary address to the address lines does the selection of a specific word inside the memory. A decoder inside the memory accepts this address and opens the paths needed to select the bits of the specified word.

Computer memory may range from 1024 words, requiring an address of 10 bits, to $2^{32}$ words, requiring 32 address bits.
Number of words (or bytes) in a memory can be referred as letter K (kilo= $2^{10}$), M (Mega = $2^{20}$) or G (Giga = $2^{30}$).
Two major types of memories are used in computer systems: Random Access Memory (RAM) and Read-Only Memory (ROM).

**Random-Access Memory (RAM):**

RAM provides large quantities of temporary storage in a computer system. A RAM can store *many* values. An address will specify which memory value we're interested in. Each value can be a multiple-bit word (e.g., 32 bits).

RAM should be able to:

- Store many words, one per address.
- Read the word that was saved at a particular address.
- Change the word that's saved at a particular address.

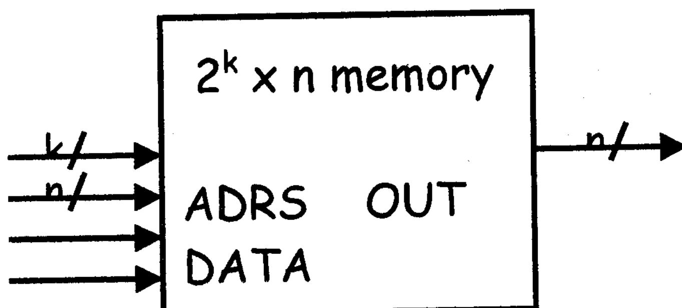



**Figure 2.6 Block diagram of RAM**

| CS | WR | Memory operation |
|---|---|---|
| 0 | x | None |
| 1 | 0 | Read selected word |
| 1 | 1 | Write selected word |

This block diagram introduces the main interface to RAM.
–A Chip Select, CS, enables or disables the RAM.
–ADRS specifies the address or location to read from or write to.
–WR selects between reading from or writing to the memory.
To read from memory, WR should be set to 0.
OUT will be the n-bit value stored at ADRS.
To write to memory, we set WR = 1.
DATA is the n-bit value to save in memory.
We refer to this as a $2^k \times n$ memory.
–There are k *address lines*, which can specify one of $2^k$ addresses.
–Each address contains an n-bit word.
For example, a $2^{24} \times 16$ RAM contains $2^{24}$ = 16M words, each 16 bits long.
–The RAM would need 24 address lines.
–The total storage capacity is $2^{24} \times 16 = 2^{28}$ bits.

**Read-Only Memory:**

One major type of memory that is used in PCs is called *read-only memory*, or *ROM* for short. ROM is a type of memory that normally can only be read, as opposed to RAM, which can be both read and written. There are two main reasons that read-only memory is used for certain functions within the PC.

**Permanence:** The values stored in ROM are always there, whether the power is ON or not. ROM can be removed from the PC, stored for an indefinite period of time, and then replaced, and the data it contains will still be there. For this reason, it is called *non-volatile storage.* A hard disk is also non-volatile, for the same reason, but regular RAM is not.

**Security:** The fact that ROM cannot easily be modified provides a measure of security against accidental (or malicious) changes to its contents. You are not going to find virus infecting true ROMs, it's just not possible. (It's technically possible with *erasable* EPROMs, though in practice never seen.)

Read-only memory is most commonly used to store system-level programs that we want to make it available to the PC at all times. The most common example is the system BIOS program, which is stored in a ROM, called (amazingly enough) the *system BIOS ROM.* Having this in a permanent ROM means it is available when the power is turned ON so that the PC can use it to boot up the system. Remember that when you first turn on the PC the system memory is empty, so there has to be *something* for the PC to use when it starts up.

While the whole point of a ROM is supposed to be that the contents cannot be changed, there are times when being able to change the contents of a ROM can be very useful. There are several ROM variants that can be changed under certain circumstances; these can be thought of as "*mostly* read-only memory". The following are the different types of ROMs with a description of their relative modifiability:

**ROM:** A regular ROM is constructed from hard-wired logic, encoded in the silicon itself, much the way that a processor is. It is designed to perform a specific function and cannot be changed. This is inflexible and so regular ROMs are only used generally for programs that are static (not changing often) and mass-produced.

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

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

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

## 2.7. SUMMARY

In this lesson we learnt the concept of Integrated Circuits. In this lesson we introduced the concept of Decoders, Multiplexer, Registers and Counters. In this lesson we have learnt the different types of memories like RAM, ROM.

## 2.8. MODEL QUESTIONS

1. Define Integrated Circuits? Explain the family of Integrated Circuits?
2. What is a Decoder? Write the steps to construct a Decoder?
3. Define Register? Explain the concept of Shift Register?
4. Explain the differences between RAM and ROM?

## 2.9. REFERENCES

1. Computer System Architecture – M. Morris Mano – Eastern Economy Edition.
2. Digital Design - M.Morris Mano – Englewood Cliffs.

**Sri. C.V.P.R.PRASAD,** M.C.A.,

**Lesson 3**

# Data Representation

## 3.0 Objective

- To describe the Number Systems
- To describe the representation of Negative numbers
- To describe the representation of fixed point numbers
- To describe the representation of floating point numbers
- To describe other binary codes
- To identifying the errors by using error detection codes

## Structure of the Lesson

**3.1 Introduction**

**3.2 Data Types**
**3.3 Complements**
**3.4 Fixed point Representation**
**3.5 Floating point Representation**
**3.6 Binary codes**
**3.7 Error codes**
**3.8 Summary**
**3.9 Model questions**
**3.10 References**

## 3.1 Introduction

How does the information is represented in a computer? Well, it is in the form of Binary Digit popularly call Bit. But how do the arithmetic calculations are performed through these bits? How the words like ABCD are then stored in computers? This chapter will try to highlight all these points. But before we try to answer these questions, let us first recapitulate the number system.

## 3.2 Data Types

- Number Systems
- Decimal Representation
- Alphanumeric Representation
- Complements

**Number Systems**: A number system of base (also called radix) r is a system, which has r distinct symbols for r digits. Radix or base is the distinct symbols used in that system. A string of these symbolic digits represents a number. To determine the quantity that the represents number, we multiply the number by an integer power of r depending on the place it is located and then find the number of weighted digits. Most commonly used number systems are

| Number System | Radix Or Base | Symbols |
|---|---|---|
| Binary | 2 | 0,1 |
| Decimal | 10 | 0,1,2,3,4,5,6,7,8,9 |
| 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 |

**Table 3.1 : Number Systems**

**Decimal Numbers:** Decimal number system has ten digits represented by 0,1,2,3,4,5,6,7,8 and 9. Any decimal number can be represented as a string of these digits and since there are ten decimal digits, therefore the base or radix of this system is 10.

Thus, a string of number 234.5 can be represented in quantity as:

$2x\ 10^2 + 3x10^1 + 4x\ 10^0 + 5x\ 10^{-1}$

**Binary Numbers:** In binary numbers we have two digits 0 and 1 and they can also be represented as a string of these two-digits called bits. The base of binary number system is 2.

For converting the value of binary number to decimal equivalent we have to find its quantity, which is found by multiplying a digit by its place value. For example binary number 101010 is equivalent to

$1x2^5+0x2^4+1x2^3+0x2^2+1x2^1+0x2^0$
=1x32+0x16+1x8+0x4+1x2+0x1
=32+8+2
=42 in decimal

**Octal Numbers**: An octal system has eight digit represented as 0,1,2,3,4,5,6,7. For formatting equivalent decimal number of an octal number one have to be find the quantity of the octal number, which is again calculated as:

Octal number $(23.4)_8$

(Please note the subscript 8 indicates it is an octal number, similarly a subscript 2 will indicate binary, 10 will indicate decimal and H will indicate hexadecimal number, in case no subscript is specified then number should be treated as decimal number or else what ever number system is specified before it.)

**Decimal equivalent**

$(23.4)_8 = 2x8^1+3x8^0+4x8^{-1}$
$= 2x8+3x1+4x1/8$
$=16+3+0.5$
$= (23.4)_{10}$

**Hexadecimal Numbers:** The hexadecimal system has 16 digits, which are represented as 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F. A number $(F2)_H$ is equivalent to

$F x16^1+2x16^0$
(As F is equivalent to 15 for decimal)
=240+2
$=(242)_{10}$

**Conversions:**

**Decimal Number to Binary Number: For** converting a decimal number to binary number, the integer and fractional part are handled separately. For Integer Part the given decimal number is divided by the binary radix 2 until the value becomes 0. Collect all the remainders from bottom to top (reverse order) for the corresponding binary number.
e.g.: Convert the decimal number 75 to a binary value

2 | 75
2 | 37 -- 1
2 | 18 -- 1
2 | 9 -- 0
2 | 4 -- 1
2 | 2 -- 0
2 | 1 -- 0
2 | 0 -- 1

$(75)_{10} = (1001011)_2$

For fraction part, multiplying the fraction with the radix 2 repeatedly and separating the integer as you get it till you have all zeros in fraction or six bits are collected.

E.g.: Convert 0.35 to binary value

0.35 * 2 = 0.70 -- 0
0.70 * 2 = 1.40 -- 1

0.40 * 2 = 0.80 -- 0

0.80 * 2 = 1.60 -- 1

0.60 * 2 = 1.20 -- 1

0.20 * 2 = 0.40 -- 0

$(0.35)_{10} = (.010110)_2$

E.g.: Convert the decimal number 43.125 to binary

| | Integer part | | Fraction part |
|---|---|---|---|
| 2 | 43 | | 0.125 * 2 = 0.250 -- 0 |
| 2 | 21 | -- 1 | 0.250 * 2 = 0.500 -- 0 |
| 2 | 10 | -- 1 | 0.500 * 2 = 1.000 -- 1 |
| 2 | 5 | -- 0 | |
| 2 | 2 | -- 1 | |
| 2 | 1 | -- 0 | |
| 2 | 0 | -- 1 | |

The integer equivalent binary is 101011
The fraction equivalent binary is .001
$(43.125)_{10} = (101011.001)_2$

**Binary To Decimal**: Each digit in the integral part of a binary number is multiplied with the powers of radix 2 starting from right. If the number is a floating-point number then each digit after the decimal is divided with powers of Radix 2.

**E.g.:** Convert $(10101011)_2$ to a decimal number
$1 * 2^7 + 0 * 2^6 + 1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$
= 128+32+8+2+1
= 141
$(10101011)_2 = (141)_{10}$

**E.g.:** Convert $(101101.101)_2$ to a decimal number
Integral part:
$1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0$
= 32 + 8 + 4 + 1
= 45
Fraction part: $1 * 2^{-1} + 0 * 2^{-2} + 1 * 2^{-3}$

1 * 0.5 + 0 * 0.25 + 1 * 0.125
= 0.625
$(101101.101)_2 = (45.625)_{10}$

**Decimal To Octal:** The given decimal number is divided by the octal system radix 8 until the value becomes 0. Collect all the remainders in the reverse order (from bottom to top) for the corresponding octal number.

E.g.: Convert 175 to an octal value

```
8| 175
8| 21 -- 7
8|  2 -- 5
8|  0 -- 2
```

$(175)_{10} = (257)_8$

**Octal to Decimal:** Each digit in the integral part of a octal number is multiplied with the powers of radix 8 starting from right. If the number is a floating-point number then each digit after the decimal is divided with powers of Radix 8.

E.g.: Convert $(231)_8$ to a decimal number

Integral part: $2 * 8^2 + 3 * 8^1 + 1 * 8^0$

$= 128 + 24 + 1 = 153$

$(231)_8 = (153)_{10}$

**Binary to Octal**: For converting binary to octal the binary number is divided into groups of three, which are then combined by place value to generate equivalent octal. For example the number

0 101 011. 001 01

001 101 011. 001 010

(Please note the number is unchanged we have added 0 to complete the grouping. Also note the style of grouping before and after decimal. We count three numbers from right to left while after the decimal from left to right.)

The 011 $= 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
$= 0 + 2 + 1$
$= 3$

101 $= 1 \times 4 + 0 \times 2 + 1 \times 1 = 5$

01 $= 0 \times 4 + 0 \times 2 + 1 \times 1 = 1$

10 $= 0 \times 4 + 1 \times 2 + 0 \times 1 = 2$

$(1101011.00101)_2 = (153.12)_8$

**E.g.:** Convert $(010101101)_2$ to a octal number

010 101 101 = 2 5 5

$(010101101)_2 = (255)_8$

**Octal To Binary:** Each digit in octal number is converted to a three bit binary code to form corresponding binary value.

E.g.: Convert $(2357)_8$ to binary number
2 3 5 7 = 010 011 101 111
$(2357)_8 = (010011101111)_2$

**Decimal To Hexadecimal:** The given decimal number is divided by the hexadecimal system radix 16 until the value becomes 0. Collect all the remainders in the reverse order for the corresponding octal number.

eg.: convert 684 to a octal value

16 | 684
16 | 42 – 12 C
16 | 2 – 10 A
16 | 0 – 2

$(684)_{10} = (2AC)_{16}$

**Hexadecimal to Decimal**: Each digit in the integral part of a hexadecimal number is multiplied with the powers of radix 16 starting from right. If the number is a Floating point number then each digit after the decimal is divided with powers of Radix 16.

E.g.: Convert $(A23C)_{16}$ to a decimal number
Integral part: $A * 16^3 + 2 * 16^2 + 3 * 16^1 + C * 16^0$
$= 10 * 16^3 + 2 * 16^2 + 3 * 16^1 + 12 * 16^0$
$= 40960 + 512 + 48 + 12$
$= 41532$
$(A23C)_{16} = (41532)_{10}$

**Binary to Hexadecimal**: The rules for these conversions are straightforward. For converting binary to Hexadecimal the binary number is divided into groups of four binary digits and finding equivalent hexadecimal digit for it can make the hexadecimal conversion.

|   | 110 | 1011 | . 0010 1 | |
|---|---|---|---|---|
| = | 0110 | 1011. | 0010 1000 | |
| = | 6 | 11 | . 2 | 8 |
| = | 6 | B | . 2 | 8 (11 in |

hexadecimal is B)

$(1101011.00101)_2 = (6B.28)_{16}$

Conversely, we can conclude that a hexadecimal digit can broken down into a string of a binary having 4 places and an octal can be broken down into string of binary having 3 place. The following figure gives the binary equivalents of octal and hexadecimal numbers.

| Octal Number | Binary Coded Octal | Hexadecimal | | Binary-coded Hexadecimal |
|---|---|---|---|---|
| 0 | 000 | 0 | | 0000 |
| 1 | 001 | 1 | | 0001 |
| 2 | 010 | 2 | | 0010 |
| 3 | 011 | 3 | | 0011 |
| 4 | 100 | 4 | | 0100 |
| 5 | 101 | 5 | | 0101 |
| 6 | 110 | 6 | | 0110 |
| 7 | 111 | 7 | | 0111 |
| | | 8 | | 1000 |
| | | 9 | | 1001 |
| | | Decimal- | | |
| | | A | 10 | 1010 |
| | | B | 11 | 1011 |
| | | C | 12 | 1100 |
| | | D | 13 | 1101 |
| | | E | 14 | 1110 |
| | | F | 15 | 1111 |

**Table 3.2 : Binary equivalent of octal and hexadecimal numbers**

**Hexadecimal To Binary:** Each digit in hexadecimal number is converted to a four bit binary code to form corresponding binary value.

e.g.: Convert $(A23C)_{16}$ to Binary number

A 2 3 C = 1010 0010 0011 1100

$(A23C)_{16} = (1010001000111100)_2$

**Decimal Representation In Computers**: The binary number system is most natural for computer because of the two stable states of its components. But, unfortunately, this is not a very natural system for us as we work with decimal number system. Then how does the

computer do the arithmetic? One of the solutions, which are followed in most of the computers, is to convert all input values to binary. Then the computer performs arithmetic operations and finally converts the result back to the decimal number so that we can interpret it easily. Is there any alternative to this scheme? Yes, there exists an alternative way of performing computation in decimal form but it requires that the decimal number should be coded suitably before performing these computations. Normally, the decimal digits are coded in 6 – 8 bits as alphanumeric characters but for the purpose of arithmetic calculations the decimal digits are treated as 4- bit binary code.

As we know 2 binary bits can represent $2^2 = 4$ different combination, 3 bits can represent $2^3 = 8$ combination and 4 bits can represent $2^4 = 16$ combination.

To represent decimal digits into binary form we require 10 combinations only, but we need to have 4-digit code.

One of the common representations is to use first 10 binary combinations to represent the 10 decimal digits.
These are popularly known as binary coded decimals (BCD). The following figure shows the binary coded decimal numbers.

Let us represent 43.125 in BCD. It is 0100 0011.0001.0010 0101 .

| Decimal | Binary Coded Decimal |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| 1 | 0010 |
| 2 | 0011 |
| 3 | 0100 |
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |
| 10 | 0001 0000 |
| 11 | 0001 0001 |
| 12 | 0001 0010 |
| 13 | 001 0011 |

**Table 3.3 : Binary Coded Decimal**

**Alphanumeric Representation:** But what about alphabets and special characters like +,- etc ? How do we represent these in computer? A set containing alphabets, the decimal digits and special characters (roughly 10-15 in numbers) consist of at least 70-80 elements. One such code generated for this set and is popularly used in ASCII (American standard code for

information interchange). This code uses 7 bits to represent 128 characters. Now an extended ASCII is used having 8-bit character representation code on microcomputers.

Similarly binary codes can be formulated for any set of discrete elements E.g.: Colors, the spectrum, the musical notes, chess board positions etc...in addition these binary codes are also used to formulate instructions, which are advanced form of data representation. We will discuss about instructions in more details in the latter blocks.

| Character | Binary Cod | Character | Binary Code |
|---|---|---|---|
| A | 100 0001 | 0 | 011 0000 |
| B | 100 0010 | 1 | 011 0001 |
| C | 100 0011 | 2 | 011 0010 |
| D | 100 0 00 | 3 | 011 0011 |
| E | 100 0101 | 4 | 011 0100 |
| F | 100 0110 | 5 | 011 0101 |
| G | 100 0111 | 6 | 011 0110 |
| H | 100 1000 | 7 | 011 0111 |
| I | 100 1001 | 8 | 011 1000 |
| J | 100 10 10 | 9 | 011 1001 |
| K | 100 1011 | | |
| L | 100 1100 | | |
| M | 100 1101 | | 010 0000 |
| N | 100 1110 | Space | 010 1110 |
| 0 | 100 1111 | ( | 010 1000 |
| p | 10 1 0000 | + | 010 1011 |
| Q | 101 0001 | $ | 010 0100 |
| R | 1010010 | * | 010 1010 |
| S | 101 0100 | ) | 010 1001 |
| T | 100 0010 | - | 010 1101 |
| U | 101 0101 | / | 010 1111 |
| V | 101 0110 | . | 010 1100 |
| W | 101 0111 | = | 011 1101 |
| X | 101 1000 | | |
| Y | 101 1001 | | |
| Z | 1011010 | | |

**Table 3.4 American Standard Code For Information Interchange (ASCII)**

The ASCII code is the standard code commonly used for the transmission of binary information. Each character is represented by a 7-bit code and usually an eight bit is inserted for parity. The code consists of 128 characters. Ninety-five characters represent graphic symbols that include upper and lower case letters, numerals 0-9,punctuation marks and special symbols.

Twenty-three characters represent format effectors, which are functional characters for controlling the layout of printing, or display devices such as carriage return, line feed, horizontal

tabulation, and back space. The other 10 characters are used to direct the data communication flow and report its status.

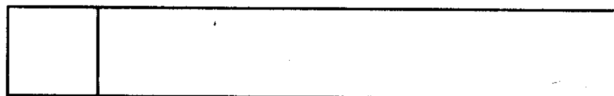
Another alphanumeric (sometimes called alphameric) code used in IBM equipment is the EBCDIC (Extended BCD Interchange Code). It uses eight bits for each character (and a ninth bit for parity). EBCDIC has the same character symbols as ASCII but the bit assignment to characters is different.

**Data Representation:** Till now we have discussed about various number systems and BCD and alphanumeric representation but now does these codes are actually used to represent data for scientific calculations? The computer is a discrete digital device and store information in flip-flops which are two state devices, in binary form. Basic requirements of the computational data representation in binary form are:

- Representation of sign
- Representation of magnitude
- If the number is fractional then binary or decimal point, and exponent.

The solution to sign representation is easy, because sign can be either positive or negative, therefore, One bit can be used to represent sign. By default it should be the left most bit. Thus a number of n bits can be represented as n+1 bit number, where n+1th bit is the sign bit and rest n bits represent its magnitude.

n+1 n n-1 ......................................3 2 1



1 bit ←--------n bits ----------------→

Sign Magnitude

A (n+1) bit Number

The decimal position can be represented by a position between the flip-flops (storage cells in computer). But, how can one determine this decimal position? Well to simplify the representation aspect two methods were suggested:(1)fixed point representation where the decimal position Is assumed either at the beginning or at the end of a number, and(2) floating point representation where a second register is used to keep the value of exponent that determines the position of the binary or decimal point in the number.

But before discussing these two representations let us first discuss the term "complement" of a number.

## 3.3 Complements

Complements may be used to represent negative numbers in digital computers. These are used for simplifying the subtraction operations and for logic manipulations.

There are two types of complements for a number of base r, these are called r's complement and (r-1)`s complement.

For example for decimal numbers the base is 10. Therefore, complements will be 10`s complement and (10-1)=9`s complements. For binary numbers we talk about 2`s and 1`s complements.

But how to obtain complements and what does these complements means? Let us discuss these issues with the help of following example.

1. **(r – 1)'s complement :** $(r^n - 1) - N$

   Here N -- is given number.
   n – number of digits in a given number.
   r – radix or base .

(r-1)'s complement for decimal number is 9's complement.
(r-1)'s complement for binary number is 1's complement.
(r-1)'s complement for octal number is 7's complement.
(r-1)'s complement for hexadecimal number is 15's complement.

1. 9's complement of a decimal number is obtained by subtracting the each digit from 9.
   Example: 9' complement of 12389 is
   99999 – 12389 = 87610 .

2. 1's complement of a binary number is obtained by subtracting each digit from 1 that is equal to changing 1's into 0's and 0's into 1's.
   Example: 1's complement of 1011001 is 0100110 .

3. 7's complement of octal number is obtained by subtracting each digit from 7.
   Example: 7's complement of octal number is 14562 is
   77777 - 14562 = 63215.

4. 15's complement of hexadecimal number is subtracting each digit from F.

**Example:**

15's complement of 1A2B3 is FFFF – 1A2B3 = E5D4C.

**2. r's complement** :

The r's complement of an n- digit number N in base r is defined as $r^n - N$ for N not equals to ZERO.
And for N=0 is $r^n$ - N = $[(r^n - 1) - N] + 1$.

That is r's complement is obtained by adding 1 to the ( r –1 )'s complement .

10's complement of decimal number is obtained by adding 1 to the 9's complement value.

E.g.: the 10's complement of the decimal number 2389 is
9's (2389) +1
= 7610 + 1 = 7611.

2's complement is obtained by adding 1 to 1's complement value.

E.g.: the 2's complement of binary number 101100 is
010011 + 1 = 010100

## 3.4 Fixed Point Representation

The fixed-point numbers in binary uses a sign bit. A positive number have a sign bit 0, while the negative number has a sign bit 1.in the fixed-point numbers we assume that the position of the binary point is at the end. It implies that all the represented numbers should be integers. a negative number can be represented in one of the following ways.

. Signed magnitude representation
. Signed 1`s complement representation, or
. Signed 2`s complement representation.

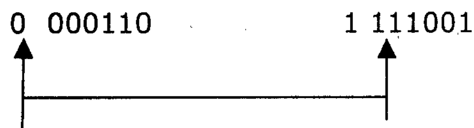
(Assumption size of register =7 bit,8th bit is used for error checking and correction or other purposes)

### Signed Magnitude Representation

| +6 | -6 |
|---|---|
| 0 000110 | 1 000110 |
| ↑ Sign bit | ↑ Sign bit |

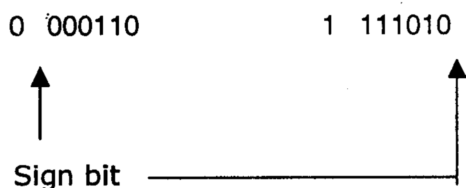
(no change in the magnitude only sign bit changes)

**Signed 1`S Complement**




(Complement all the Sign bit of the positive no to Obtain its complement **negative no)**

**Signed 2'S Complement**




(2's complement of the +ve number including sign bit)

**Arithmetic Addition:** The complexity of arithmetic addition is dependent on the representation, which have been followed. Let us discuss this with the help of following example.

**E.g.:** Add 25 and –30 in binary using 7 bit register in signed magnitude representation signed 1`s complement
Signed 2`s complement.

**Solution:**

25 or +25 is

0 0 1 1 0 0 1

-30 in signed magnitude representation is:

+30 is 0 0 1 1 1 1 0,

Therefore -30 is 1 0 1 1 1 1 0

To do the arithmetic addition with one negative number we have to check the magnitude of the no's.
The number having smaller magnitude is then subtracted from the bigger number and the sign of bigger number is selected. The implementation of such a scheme in digital hardware will require a long sequence of control decisions as well as circuits that will add, compare and subtract numbers. Is there a better alternative than this scheme? Let us first try the signed 2`s complement.

-30 in signed 2`s complement notation will be

+ 30 is 0011110

| | | |
|---|---|---|
| - 30 is | 1100010 | (2's complement of 30 including sign bit) |
| + 25 is | 0 0 1 1 0 0 1 | |
| - 25 is | 1 1 0 0 1 1 1 | |

**Addition**

1)

| | | |
|---|---|---|
| +25 | 0 011 001 | |
| +30 | 0 011 110 | |
| +55 | 0 110 111 | |

2)

| | | |
|---|---|---|
| +25 | 0 011 001 | |
| -30 | 1 100 010 | |
| -05 | 1 111 011 | 2's complement Representation of -5 |

3)

| | | | |
|---|---|---|---|
| - 25 | | 1 100111 | |
| +30 | ↑ | 0 011110 | (just add the two nos.) |
| +05 | 1 | 0 000101 | |

Discard the carry out of the sign bit.

4)

| | |
|---|---|
| -25 | 1 100111 |
| -30 | 1 100010 |
| -55 | 1 001001 |

Please note how easy is to add two numbers using signed 2`s complement. This procedure requires only control decision and only one circuit for adding the two numbers. But it require additional condition that the negative numbers should be stored in signed 2`s complement form in the registers. This can be achieved by complementing the positive number bit by bit and then incrementing the resultant by 1 to get signed 2`s complement.

**Signed 1'S Complement Representation:** Another possibility, which also is simple, is use of signed 1`s complement. Signed 1`s complement has a rule. Add the two numbers, including the

sign bit. If carry out of the most significant bit or sign bit is one, then increment the result by 1 and discard the carry over. Let us repeat all the operations with 1's complement.

---

```
+25      0  011001
- 30     1  100 001
-----   -----------------
-5       1  111 010
-----   -----------------

+5  is                    0   000101
-5  in  1's complement 1    111010

-25                       1 100110
-30                       1 100001
----                    ----------------
-55                     1 1 000111
----                    ----------------
                           ↑
                        carry out
```

Since the carry out is 1, so add 1 to sum and discard the carry

```
 1 000   111
           1
---------------
1 001    000
----------------
+55 is                        0  110111
-55   is 1`s complement       1  001001
```

Another interesting feature about these representation is the representation of 0.in signed magnitude and 1's complement there are two representations for zero as:

| | +0 | -0 |
|---|---|---|
| signed magnitude | 0 000000 | 1 000000 |
| Signed 1's complement | 0 000000 | 1 111111 |

But in signed 2's complement there is just one zero and there is no positive or negative zero.

---

+0 000000 -0 2's complement of

```
+0 = 1  111111
             1
------------------
     0  000000
```

discard the carry.

Thus both +0 and –0 are same in 2's complement notation. This is an added advantage in favor of 2's complement notation. The highest numbers, which can be accommodated in a register, also depend on the type of representation. In general, in a 8 bit register 1 bit is used as sign, therefore, rest 7 bits can be used for representing the value the highest and the lowest number, which can be represented, are:

For signed magnitude representation and for signed 1's complement

$2^7$ -1 to -$2^7$ -1

= 128-1 to – (128-1)

= 127 to –127

But, for signed 2's complement we can represent +127 to –128. The –128 is represented in signed 2's complement notation as 10000000.

**Arithmetic Subtraction:** The subtraction can be easily done using the 2's complement by taking the 2's complement of the subtracted (inclusive of sign bit) and then adding the two numbers.

Signed 2's complement provide very simple way for adding and subtracting two no's, thus, many computer (including IBM PC)adopt signed 2's complement notation. The reason why signed 2's complement is preferred over signed 1's complement is because it has only one representation for zero.

**Overflow:** An overflow is said to have occurred when the sum of two n digits number occupies n+1 digits. This definition is valid for both binary as well as decimal digits. But what is the significance of overflow for binary no's since it is not a problem for the cases when we add two numbers? Well the answer is in the limits of representation of numbers.

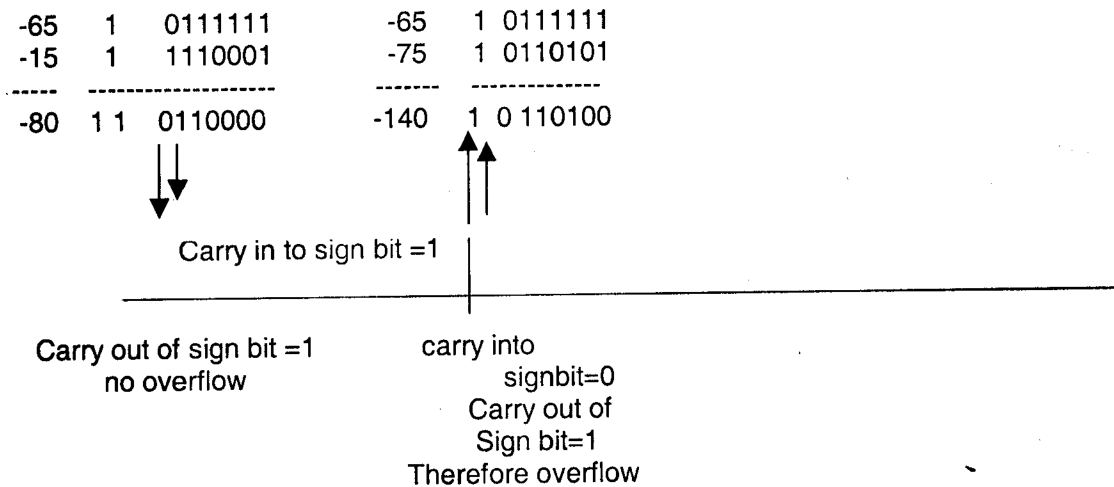
Every computer employs limit for representing number. In our examples we are using 8 bit registers for calculating the sum. But what will happen if the sum of the two numbers can be accommodated in 9 bits? Where are we going to store the 9$^{th}$ bit? The problem will be clearer by the following example.

**Example:** Add the numbers 65 and 75 in 8 bit register in signed 2's complement notation.

```
65   0 1000001
75   0 1001011
```

140 1 0001100

The expected result is +140 but the binary sum is a negative number and is equal to – 116,Which obviously is a wrong result. This has occurred because of Overflow. How do the computer know that overflow has occurred? If the carry into the sign bit is not equal to the carry out of the sign bit then overflow must have occurred




Thus, overflow has occurred, That is the arithmetic results so calculated have exceeded the capacity of the representation. This overflow also implies that the calculated results might be erroneous.

**Decimal Fixed Point Representation:** A decimal digit is represented as a combination of four bits; thus, a four digit decimal number will requires 16 bits for decimal digits representation and additional 1 bit for sign. Normally to keep the convention of one decimal digit to 4 bits, the sign sometimes is also assigned a 4-bit code. This code can be the bit combination, which has not been used to represent decimal digit e.g.. 1100 may represent plus and 1101 can represent minus.

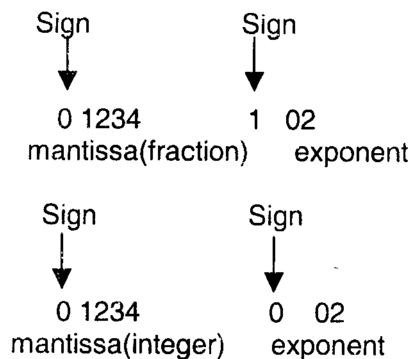
Although this scheme wastes considerable amount of storage space yet it do not require conversion of a decimal number to binary. Thus, can be used at places where the amount of computer arithmetic is less than the amount of input/output of data e.g. calculators or business data processing.
The arithmetic in decimal can also be performed as in binary except that instead of signed ones complement, signed nine's complement is used and instead of signed 2`s complement signed 10`s complement is used.

## 3.5 Floating Point Representation

Floating point number representation consists of two parts. The first of the number is a signed fixed-point number, which is termed as mantissa, and the second part specifies the decimal or binary point position and is termed as an exponent .the mantissa can be an integer or a fraction.

Please note that the position of decimal or binary point is assumed and it is not a physical point therefore, wherever we are representing a point it is only the assumed position.

Example 1: A decimal +12.34 in a typical floating point notation is:




This number in any of the above form (if represented in B C D) requires 17 bits for mantissa (1 for sign and 4 for each decimal digit as BCD). please note that the exponent indicates the correct decimal location . In the first case where exponent is +2, indicates that actual position of the decimal point is two places to the right of the assumed position, while exponent –2 indicates that the assumed position of the point is two places towards the left of assumed position. The assumption of the position of point is normally the same in a computer resulting in a consistent computational environment.

Floating-point numbers are often represented in normalized forms. A floating-point number when mantissa does not contain zero as the, most significant digit of the number is considered to be in normalized form. For example, a BCD mantissa + 370 which is 0 0011 0111 0000 is in normalized form because these leading zeros are not part of zero digit. On the hand a binary number 001100 is not in a normalized form. The normalized form of this number is: 1100. A floating binary number +1010.001 in a 16-bit register can be represented in normalized form (assuming 10 bits for mantissa and 6 bits for exponent)

| 0 | · 101000100 | 0 | 00100 |
|---|---|---|---|
| Sign bit | mantissa(fraction) | sign bit | exponent |

A zero can't be normalized as all the digits in mantissa in this case has to be zero Arithmetic operations involved with floating point numbers are more complex in nature, take longer time for execution and require complex hardware. Yet the floating-point representation is a must as it is useful in scientific calculations. Real numbers are normally represented as floating point numbers.

## 3.6 OTHER BINARY CODES

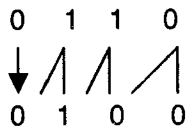
Other binary codes for decimal numbers and alphanumeric characters are sometimes used. A few additional binary codes encounters in digital computers are

**Gray Code**: The reflected binary or gray code shown in the table 3.5 is sometimes used for the converted digital data. The advantage of the gray code over the 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.

**Table 3.5 : Bit gray code**

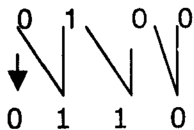
| binary Code | decimal equivalent | binary code | decimal equivalent |
|---|---|---|---|
| 0000 | 0 | 1100 | 8 |
| 0001 | 1 | 1101 | 9 |
| 0011 | 2 | 1111 | 10 |
| 0010 | 3 | 1110 | 11 |
| 0110 | 4 | 1010 | 12 |
| 0111 | 5 | 1011 | 13 |
| 0101 | 6 | 1001 | 14 |
| 0100 | 7 | 1000 | 15 |

**Conversion from Gray to Binary:**




**Procedure:** Most significant bit is as it is as the Gray code. The next most significant of Binary number is obtained by adding most significant bit of binary number to the next most significant bit of gray code. Consider only the sum and discard carry bit if any.

**Conversion from Binary to Gray:**



Procedure: The most significant bit of the Gray code is as it as the binary number. The next most significant bit of the Gray code is obtained by adding the most significant bit of the binary number to the next most significant bit of the binary number

**Other Decimal Codes:** Binary codes for decimal digits require a minimum of four digits.

| 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 |
| Unusedbit 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 |

**TABLE 3.6 : Four Different Binary Codes for the Decimal Digit**

One disadvantage of using BCD is the difficulty encountered when the 9's compliment of the number is to be computed. On the other hand, the 9's compliment of a number is easily obtained with 2421 and excess –3 codes. These two codes have a self - complementing property which means that the 9's complement of a decimal number, when represented in one of these codes, is easily obtained by changing 1's to 0's and 0's to 1's.This property is useful when arithmetic operations are done in signed-complements representation.

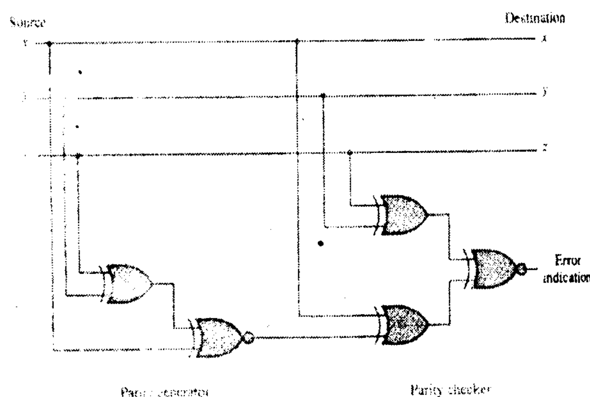
The 2421 is an example of a weighted code. In a weighted code, the bits are multiplied by the weights indicated and the sum of weighted bits gives the decimal digit. For example, the bit combination 1101,when weighted by the respective digits 2421,gives the decimal equivalent of 2x1+4x1+2x0+1+1=7.

The excess –3 codes is a decimal code that has been used in older computers. This is an un weighted code. Its binary code assignment is obtained from the corresponding BCD equivalent binary number after the addition of binary 3(0011).

## 3.7 Error Detection Codes

Binary information transmitted through some form of communication medium is subject to external noise that could change bits from 1 to 0 , and vice versa. An error detection code is a binary code that detects digital errors during transmission. The detected errors cannot be corrected but their presence is indicated. The usual procedure is to observe the frequency of errors. If errors occur infrequently at random, the particular erroneous information is transmitted again. If the error occur too often, the system is checked for malfunction.

**Figure 3.1: error detection code**




The most common error detection code used is the parity bit. A parity bit is an extra bit included with a binary message to make the total number of 1's either odd or even. A message of three bits and two possible parity bits is shown in table 3-7.The P (odd) bit is chosen in such a way as to make the sum of 1's (in all four bits) odd. The P (even) bit is chosen to make the sum of all 1's even. In either case, the sum is taken over the message and the P bit.

In even parity scheme has the disadvantage of having the bit combination of all 0's, while in the odd parity there is always one bit (of the four bits that constitute the message and P) that is 1. Note that the P (odd)is the complement of the P(even).

During transfer of information from one location to another, the parity bit is handled as follows. At the sending end, the message (in this case three bits)is applied to a parity generation, where the required parity bit is generated.

The message, including the parity bit, is transmitted to its destination. At the receiving end, all the incoming bits(in this case, four)are applied to a parity checker that checks a proper parity adopted(odd or even).

An error is detected if the checked parity does not conform to the adopted parity. The parity method detects the presence of one, three, or any odd number of errors. An even number of errors is not detected.

**Table 3.7 : Parity Bit Generation**

| Message xyz | p(odd) | p(even) |
|---|---|---|
| 000 | 1 | 0 |
| 001 | 0 | 1 |
| 010 | 0 | 1 |
| 011 | 1 | 0 |
| 100 | 0 | 1 |
| 101 | 1 | 0 |
| 110 | 1 | 0 |
| 111 | 0 | 1 |

## 3.8 Summary

This completes our discussion on Data Representation .The information given on various topics such as various Number systems, signed, 1's complement and 2's complement representation of negative numbers, Representation of floating point numbers, various binary codes, which are used for representing alphanumeric information and error codes etc. although is exhaustive yet can be supplemented with additional readings.

In fact, a course in an area of computer Science must be supplemented by further readings to keep your knowledge up to date, as the computer world is changing with leaps and bounds. In addition to further readings the student is advised to study several Indian Journals on computers to enhance his knowledge.

## 3.9 Model Questions

1. Discuss about different number systems?
2. Convert the following binary numbers to decimal: 101110,110011.
3. Convert the following decimal numbers to binary 4567, 9876.
4. Convert the following decimal numbers to the base indicated.

   a) 1234 to binary b) 9876 to octal c) 5678 to Hexadecimal
5. Convert the hexadecimal number D3C4A2 to binary.
6. Show the value of all bits of a 12-bit register that hold the number Equivalent to decimal 345 in a) binary b) binary-coded octal c) binary-coded hexadecimal d) binary-coded decimal (BCD)

7. Write your name in ASCII using an 8-bit code with the leftmost bit always 0. Include a space between names.

8. Obtain the 9's complement and 10's complement of the following decimal numbers

a) 23456701 b) 78901234

9. Obtain the 1's complement and 2's complement of the following binary numbers.

a) 10101011 b) 1110101

10. Perform the arithmetic operations (+67) + (- 32) and (-67) - (-32) in binary using signed 2's complement representation for negative numbers.

11. Perform the following arithmetic operations with the decimal numbers using signed – 10's complement representation for negative numbers.

a) (-987) +(+234) b) (-987) – (+234)

12. Represent the number $(+46.5)_{10}$ as a floating-point binary number with 24 bits. The normalized fraction mantissa has 16 bits and the exponent has 8 bits.

13. Represent the decimal number 8620 in a) BCD b)excess-3code c) 2421code d) as a binary number.

14. Discuss how negative numbers are represented ?

15. What are the advantages of 2's complement representation than 1's complement representation of negative numbers ?

16. Discuss how fixed-point numbers are represented ?

17. Discuss about floating point representation ?

18. Discuss various binary codes, which are used for storing alphanumeric Information ?

19. Explain error detection codes in detail ?

20. Derive the circuits for a 3-bit parity generator and 4- bit parity checker using an even-parity bit.

## 3.10 References

1. Computer System Architecture
Third Edition Prentice-Hall of India **by M. Morris Mano**

2. Digital Design
Second edition Prentice-Hall of India **by M.M.Mano**

Lesson 4

# Register Transfer And Microoperations

## 4.0 Objective :

At the end of this Chapter, you should be able to

- Describe the register organization of the CPU
- Differentiate among various structures of the CPU
- Define what is a microoperation
- Differentiate among various microoperations
- Discuss how the microoperations can be implemented

## Structure of the Lesson


**4.1 Introduction**

**4.2 Register Transfer Language**

**4.3 Arithmetic Micro operations**

**4.4 Logic Microoperations**

**4.5 Shift Microoperations**

**4.6 Summary**

**4.7 Model Questions**

**4.8 References**


## 4.1 Introduction

In this Chapter, first we will define the register transfer language, the basic symbols for register transfer and then about various microoperations and their hardware implementation from where we will move on to design of a very simple circuit of an arithmetic logic unit.

## 4.2 Register Transfer Language

A digital system is an interconnection of digital hardware modules. The various modules are interconnected with common data and control paths to form a digital computer system.

The registers they contain and the operations that are performed on the data stored in them define digital modules.

A micro operation is an elementary operation performed on the information stored in one or more registers. Examples of micro operations are shift, count, clear and load.

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

1) The set of registers it contains and their function.

2) The sequence of microoperations performed on the binary Information stored in the registers.
3) The control that initiates the sequence of micro operations.

It is more convenient to adopt a suitable symbology to describe the sequence of transfers between registers and the various arithmetic and logic microoperations associated with the transfers.

The symbolic notation used to describe the microoperation transfers among registers is called a register transfer language.

**Table 4.1: Basic Symbols for Register Transfers**

| Symbol | Description | Examples |
|---|---|---|
| Letters (And 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 |
| Comma | Separates two micro Operations | R2←R1, R1← R2 |

## 4.3 Arithmetic Microoperations

A micro operation is an elementary operation performed with the data stored in registers. The microoperations most often encountered in digital computers are classified into four categories:

1. Register transfer microoperations transfer binary information from one register to another.
2. Arithmetic microoperations perform arithmetic operations on numeric data stored in registers.
3. Logic microoperations perform bit manipulation operations on non-numeric data stored in registers.
4. Shift microoperations perform shift operations on data stored in registers.

The register transfer microoperation does not change the information content when the binary information moves from the source register to the destination register. The other three types of microoperations change the information content during the transfer.

The basic arithmetic microoperations are addition, subtraction, increment, decrement and shift. The arithmetic microoperation defined by the statement, specifies an add

R3 R1 ← R2

microoperation. It states that the contents of register R1 are added to the contents of register R2 and the sum transferred to register R3.

To implement this statement with hardware we need three registers and the digital component that performs the addition operation. The other basic arithmetic microoperations are listed in Table 4.2.

**Table 4.2: 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 |
| $R3 \leftarrow R1+\overline{R2}+1$ | R1 plus 2's complement of R2 |
| $R1 \leftarrow R1+1$ | Increments the contents R1 by one |
| $R2 \leftarrow R1 - 1$ | Decrements the contents r1 by one |

Subtraction is most often implemented through complementation and addition. Instead of using the minus operator, we can specify the subtraction by the following statement.

$$R3 \leftarrow R1 + \overline{R2} + 1$$

$\underline{R2}$ is the symbol for the 1's complement of R2.

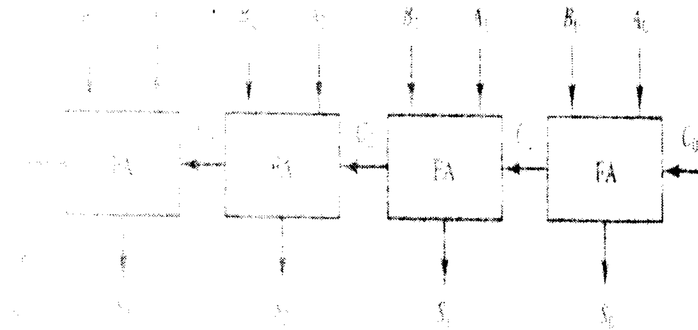
Adding 1 to the 1's complement produces the 2's complement. Adding the contents of R1 to 2's complement of R2 is equivalent to R1-R2.

The increment and decrement micro operations are symbolized by plus and minus operations, respectively. These micro operations are implemented with a combinational circuit or with a binary up-down counter. In most computers, the multiplication operation is implemented with a sequence of add and shift micro operations. Division is implemented with a sequence of subtract and shift micro operations.

**Binary Adder:** The hardware implementation of the add microoperation require the registers that hold the data and the digital component that performs the arithmetic addition. The digital circuit that forms the arithmetic sum of two bits and a previous carry is called a full-adder. The digital circuit that generates the arithmetic sum of two binary numbers of any length is called a

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

**Fig 4.1: 4-Bit Binary Adder**




The fig.4.1 shows the inter connections 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 lower-order bit. The carries are connected in a chain though the full-adders.
The input carry to the binary adder is $C_0$ and the output carry is $C_4$. The S outputs of 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. The *n* data bits for the A inputs come from one register and the *n* data bits for the B inputs come from another register. The sum can be transferred to a third register or to one of the source registers (R1 or R2), replacing its previous content.

**Binary Adder – Subtractor:** The subtraction of binary numbers can be done most conveniently by means of complements. The subtraction A – B can be done by taking the 2's complement of B and adding it to A. Taking 1's complement and adding one to the least significant pair of bits can obtain the 2's complement.

The 1's complement can be implemented with inverters and a 1 can be added to the sum through the input carry.
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-sub tractor circuit is shown in Fig.4.2.

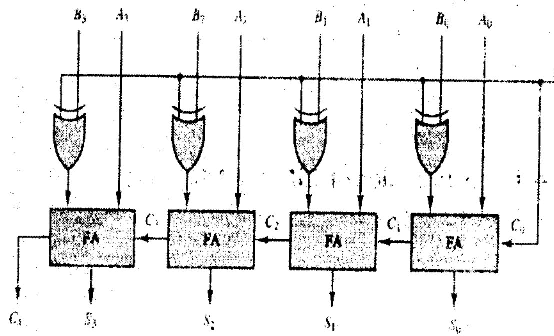



**Fig.4.2: 4-bit Binary adder-Subtractor**

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 0= B. The full-adder⊕eceive the value of B, the input carry is 0 and the 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.

Binary Incrementer: The increment microoperation 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. This can be accomplished by means of half-adders connected in cascade.

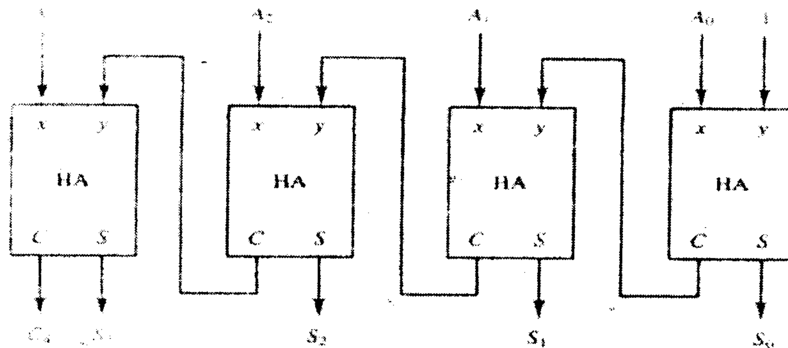



**Fig.4.3 :4-Bit Binary Incrementer**

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 $A_3$, adds 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 also causes outputs $S_0$ through $S_3$ to go to 0.

The circuit of fig. 4.3 ca be extended to an 4 bit Binary Incrementer by extending the diagram to include n half-adders. The least significant bit must have one input connected to login-1. The other inputs receive the number to be incremented or the carry from the previous stage.

**Arithmetic Circuit:** The arithmetic microoperations listed in Table 4.2 can be implemented in one composite arithmetic circuit.

The basic component of an arithmetic circuit is the parallel adder. By controlling the data inputs to the adder, it is possible to obtain different types of arithmetic operations.

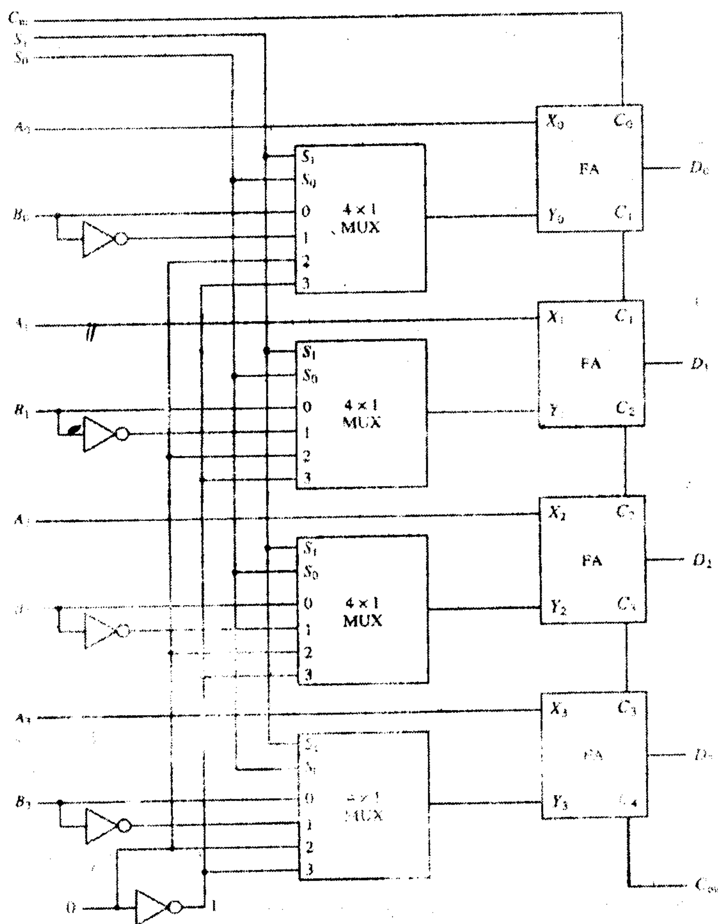



Fig. 4.4: 4-Bit Arithmetic circuit

**The diagram of a 4-bit Arithmetic circuit is shown in fig. 4.4.**
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 are connected from one stage to the next.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. Cin 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 Cin equal to 0 or 1, it is possible to generate the eight arithmetic microoperations listed in table 4.3.

**Table 4.3: Arithmetic Circuit Function Table**

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

**Addition:** 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+1.

**Subtraction:** 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 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.

**Increment:** When $s_1s_0$=10, the inputs from B are neglected. All 0's are inserted into the y inputs. The output becomes D=A+0+$C_{in}$. When $C_{in}$=0,then D=A, when $C_{in}$=1,then D=A+1.

**Decrement:** 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 equivalent to the 2's complement of 1.

## 4.4 Logic Micro operations

Logic micro operations specify binary operations for strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables. For example the following statement symbolizes the exclusive–OR microoperation with the content of two registers R1 and R2

$$P: \quad R1 \leftarrow R1 \oplus R2$$

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

**Example:** Assume that each register has four bits. Let the content of R1 be 1010 and the content of R2 be 1100.The exclusive-OR micro operation stated above symbolizes the following computation:

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

Special symbols will be adopted for the logic microoperations OR, AND, and complement, the symbol V will be used to denote an OR micro operation and the symbol ^ to denote an AND microoperation. The complement micro operation is the same as 1's complement and uses a bar on top of the symbol that denotes the register name.

**List of Logic Microoperations:** There are 16 different logic operations that can be performed with two binary variables. The 16 Boolean functions of two variables x and y are expressed in algebraic form in the first column of Table 4.5.

| 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 |

**Table 4.4 : Truth Tables for 16 Functions of Two Variables**

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

**TABLE 4.5—Sixteen Logic Microoperations**

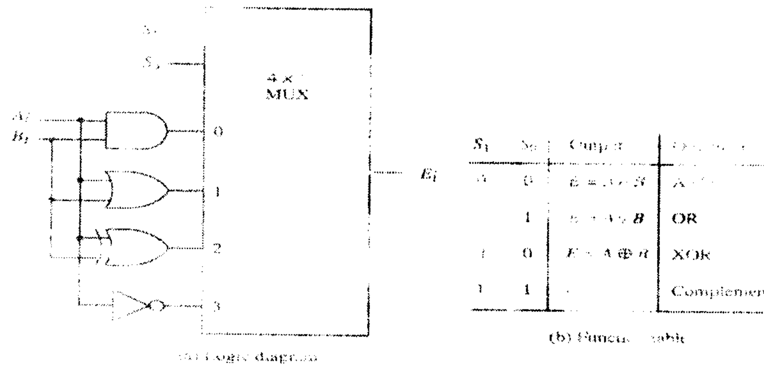



**Fig.4.5 :One stage of logic circuit**

**Hardware Implementation:** Fig.4.5 shows one stage of a circuit that generates the four basic logic microoperations.

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 lists the logic microoperations obtained for each combination of the selection variables.

**Some Applications:** Logic microoperations are very useful for manipulating individual bits or a portion of a word stored in a register. They can be used to change bit values, delete a group of bits, or insert new bit values into register.

Register A is a processor register and the bits of register B constitute a logic operand extracted from memory and placed in register B.

The ***selective-set*** operation sets to 1 the bits in register A where there are corresponding 1's in register B. It does not affect bit positions that have 0's in B.

**Example:**

| | |
|---|---|
| **1010** | **A before** |
| **1100** | **B (logic operand)** |
| 1110 | **A after** |

The bits of A after the operation are obtained from logic-OR operation of bits of B and Previous values of A. Therefore, the OR microoperation can be used to selectively set bits of a register. The ***selective-complement*** operation complements bits in A where there are corresponding 1's in B. It does not effect bit positions that have 0's in B.

**Example:**

| | |
|---|---|
| **1010** | **A before** |
| **1100** | **B(logic operand)** |
| **0110** | **A after** |

The exclusive-OR microoperation can be used to selective complement bits of a register.

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

| | |
|---|---|
| **1010** | **A before** |
| **1100** | **B (logic operand)** |
| **0010** | **A after** |

The corresponding logic microoperation is

$$A \leftarrow A \wedge \overline{B}$$

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

**Example:**

| | |
|---|---|
| **1010** | **A before** |
| **1100** | **B (Logic operand)** |
| **1000** | **A after masking** |

The **insert** operation inserts a new value into a group of bits. This is done by first masking the bits and then OR'ing them with the required value.

**Example:** Suppose that an A register contains eight bits, 0110 1010. To replace the four leftmost bits by the value 1001 we first mask the four unwanted bits:

| | |
|---|---|
| **0110 1010** | **A before** |
| **0000 1111** | **B(mask)** |
| **0000 1010** | **A after masking** |

and then insert the new value:

```
0000 1010    A before
 1001 0000   B(insert)
 1001 1010   A after insertion
```

The mask operation is an AND microoperation and the insert operation is an OR microoperation.

The *clear* operation compares the words A and B and products an all 0's result if the two numbers are equal. This operation is achieved by an exclusive-OR microoperation
Example:

```
 1010 A
 1010 B
 0000 A ← A ⊕ B
```

## 4.5 Shift Micro Operations

Shift micro operations are used for serial transfer of data. The contents of a register can be shifted to the left or the right. During the shift-left operation the serial input transfers a bit into the rightmost position. During a shift-right operation the serial input transfers a bit into the leftmost position. There are three types of shifts: Logical, Circular and Arithmetic.

We will adopt the symbols for shift-left and shift-right micro operations.

| Symbolic designation | Description |
|---|---|
| R ← Shl R | Shift-left register R |
| R ← Shr R | Shift-right register R |
| R ← cil R | Circular shift-left register R |
| R ← cir R | Circular shift-right register R |
| R ← ashl R | Arithmetic shift-left R |
| R ← ashr R | Arithmetic shift-right R |

**Table 4.6: Shift Microoperations**

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

R1 ← shl R1
R2 ← shr R2

The bit transferred to the end position through the serial input is assumed to be 0 during a logical shift.

Example for Logical Shift left

R1 -->

| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

Shl R1→

| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
|---|---|---|---|---|---|---|---|

Example for Logical Shift right

R1 →

| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|

Shr R1→

| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

The **circular shift** circulates the bits of the register around the two ends without loss of information. This is accomplished by connecting the serial output of shift register to its serial input. We will use the symbols cil and cir for the circular shift let and right, respectively.

Example for Circular Shift Left:

R1 →

| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

Cil R1 →

| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|

Example for Circular Shift Right:

R1→

| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

Cir R1→

| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

An **arithmetic shift** is a microoperation that shifts a signed binary number to the left or right. An arithmetic shift-left multiplies a signed binary number by 2. An arithmetic shift-right divides the 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 arithmetic shift-right leaves the sign bit unchanged and shifts the number to the right. Thus $R_{n-1}$ remains the same, $R_{n-2}$ receives the bit from $R_{n-1}$ and so on for the other bits in the register. The bit in $R_0$ is lost.

The arithmetic shift-left inserts a 0 into $R_0$ and shifts all other bits to the left. The initial bit of $R_{n-1}$ is lost and replaced by the bit from $R_{n-2}$ . A sign reversal occurs if the bit in $R_{n-1}$ changes in value after the shift. This happens if the multiplication by 2 causes an overflow.
An overflow occurs after an arithmetic shift left if initially, before the shift, $R_{n-1}$ not equal to $R_{n-2}$.
An overflow flip-flop $V_s$ can be used to detect an arithmetic shift-left overflow.

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

If $V_s$=0,there is no overflow, but if $V_s$=1, there is an overflow.
Example for Arithmetic shift left:

R1 →

| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

Ashl R1 →

| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|

Example for Arithmetic Shift right:

R1 →

| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
|---|---|---|---|---|---|---|---|

Ashr R1 →

| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

**Hardware Implementation**: The content of a register that has to be shifted is first placed onto a common bus whose outpu connected to the combinational shifter, and the shifted number is then loaded back into the register.

A combinational circuit shifter can be constructed with multiplexers as Shown in Fig 4.6

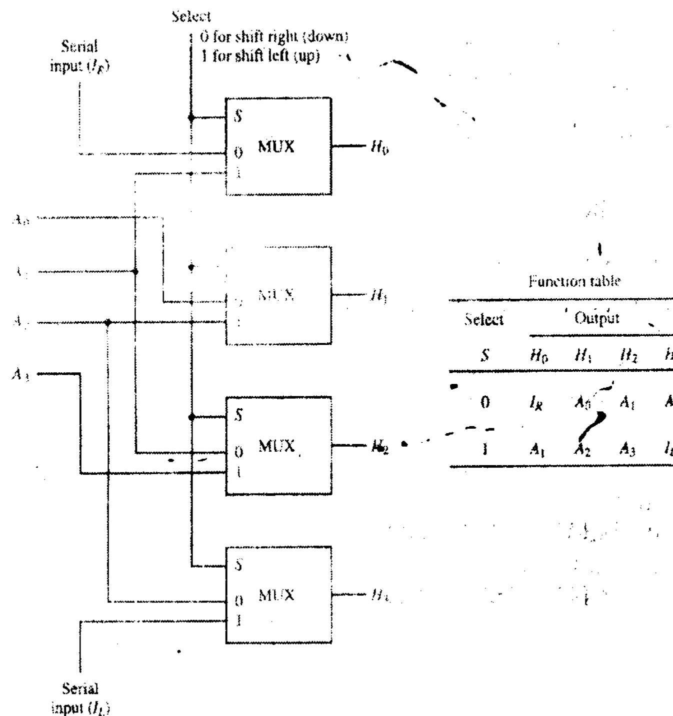



**Figure 4.6 : 4-Bit Combinational Circuit Shifter**

The 4-bit shifter has four da... ..s, $A_0$ through $A_3$, and four data outputs, $H_0$ through $H_3$. There are two serial inputs, one f... ... left($I_L$) and the other for shift right ($I_R$). When the selection input S=0, the input data are ... ..d right. When S=1, the input data are shifted left.

Figure 4.6 shows, which input goes to each output after the shift. A shifter with n data inputs and outputs requires n multiplexers.

## 4.6 Summary

In this Chapter, we have discussed in detail about the register organization. After this we have discussed in details about the micro-operations and their implementation in hardware using simple logic circuits. While discussing about microoperations, our main emphasis was on simple arithmetic, logic and shift microoperations, in addition to register transfer.

## 4.7 Model Questions

1. Explain Arithmetic Microoperations in detail.
2. Draw a 4-bit binary adder and explain it in detail.
3. Draw a 4-bit binary adder-subtractor and explain it in detail.
4. Draw a 4-bit incrementer and explain it in detail.
5. Explain 4-bit arithmetic circuit with neat diagram.
6. Explain logic Microoperations.
7. Explain applications of logic Microoperations.
8. Explain Shift Microoperations with examples.
9. Starting from an in... value of R = 11011101,Sequence o binary values in R after a logical shift-left, followed by a Circular shift - right, followed by a logical shift-right and a circular Shift-left.
10. An 8-bit register contains the binary value 10011100. What is the Register value after an arithmetic shift right ? Starting from the Initial number 10011100, determine the register value after an Arithmetic shift left, and state whether there is an overflow.

## 4.8 REFERENCES

1. Computer System Architecture
by **M.MORRIS MANO**
2. Computer Engineering: Hardware Design
by **M.MORRIS MANO**

**T. ANURADHA,** M. Sc., M.S.,

Lesson 5

# Basic Computer Organization

## 5.0 Objective :

At the end of this chapter, you should be able to

- Describe basic computer instruction formats
- Describe different instructions for the basic computer
- Describe timing and control
- Describe about execution of instructions
- Discuss about Input-output configuration
- Discuss about program interrupt

## Structure of the Lesson


5.1 Introduction

5.2 Computer Instructions

5.3 Timing and control

5.4 Execution of Instruction

5.5 Input-output and Interrupt

5.6 Summary

5.7 Model questions

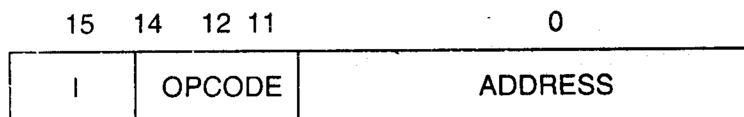
5.8 References


## 5.1 Introduction

In this chapter we introduce a basic computer and show how its operation can be specified with register statements. Its internal registers, the timing and control structure, the set of instructions that it uses and also defines the organization of the computer. This chapter also discusses about execution of an instruction and program interrupt.

## 5.2 Computer Instructions

The basic computer has three instruction code formats, as shown in figure5.1



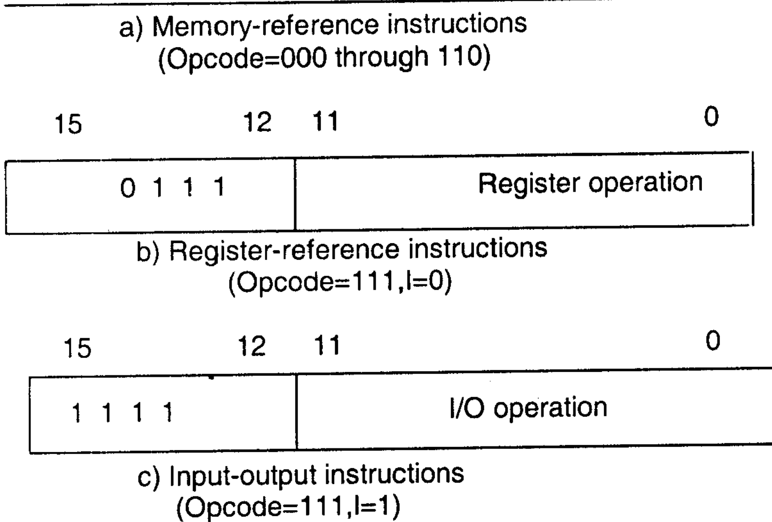



**Figure 5.1 : Basic Computer Instruction Formats**

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

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

The register reference instructions are recognized by the operation code 111 with a 0 in the leftmost bit (bit15) of the instruction. A register –reference instruction specifies an operation on or a test of the AC register. An operand from memory is not needed; therefore, the other 12 bits are used to specify the operation or test to be executed.

Similarly, an input-output instruction does not need a reference to memory and is recognized by the operation code 111 with a 1 in the leftmost bit of the instruction. The remaining 12 bits are used to specify the type of input-output operation or test performed.

The total number of instructions chosen for the basic computer is equal to 25. The instructions for the basic computer are listed in the following Table 5-1

**Table 5.1: Basic Computer Instructions**

| SYMBOL | Hexadecimal | | Description |
|---|---|---|---|
| | I=0 | I=1 | |
| AND | 0XXX | 8XX X | AND memory word to AC |
| ADD | 1XXX | 9XXX | Add memory word to AC |
| LDA | 2XXX | AXXX | Load memory word to AC |
| STA | 3XXX | BXXX | Store content of AC in 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 instruction if AC + ve |
| SNA | 7008 | Skip next instruction if AC -ve |
| SZA | 7004 | Skip next instruction if AC is 0 |
| SZE | 7002 | Skip next instruction if E is 0 |
| 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 |

By using the hexadecimal equivalent we reduced the 16 bits of an instruction code to four digits with each hexadecimal digit being equivalent to four bits. A memory-reference instruction has an address part of 12 bits. The address part is denoted by three x's and stands for the three hexadecimal digits corresponding to the 12-bit address. The last bit of the instruction is designated by the symbol I.

Register-reference instructions use 16 bits to specify an operation. The leftmost four bits are always 0111, which is equivalent to hexadecimal 7. The other three hexadecimal digits give the binary equivalent of the remaining 12 bits. The input-output instructions also use all 16 bits to specify an operation. The leftmost four bits are always 1111, equivalent to hexadecimal F.

**Instruction set completeness:** The set of instructions are said to be complete if the computer includes a sufficient number of instructions in each of the following categories:

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

The instructions listed in Table 5-1 constitute a minimum set that provides all the capabilities mentioned above. There is one arithmetic instruction ADD and two related instructions complement AC (CMA) and increment AC (INC). With these three instructions we can add and subtract binary numbers when negative numbers are in signed-2's complement representation. The circulate instructions Cir and Cil, can be used for arithmetic shifts as well as any other type of shifts desired. Multiplication and division can be performed using addition, subtraction, and shifting.

There are three logic operations: AND, complement AC (CMA), and clear AC (CLA). The AND and complement provide a NAND operation. It can be shown that with the NAND operation it is possible to implement all the other logic operations. Moving information from memory to AC is accomplished with the load AC (LDA) instruction.

Storing information from AC into memory is done with the store AC (STA) instruction. The branch instructions BUN, BSA, and ISZ, together with the four skip instructions, provide capabilities for program control and checking of status conditions.

The input (INP) and output (OUT) instructions cause information to be transferred between the computer and external devices.

## 5.3 Timing and Control

A master clock generator controls the timing for all registers in the basic computer. The clock pulses are applied to all flip-flops and registers in the system, including the flip-flops and registers in the control unit. The clock pulses do not change the state of a register unless the register is enabled by a control signal. The control signals are generated in the control unit.

There are two major types of control organization: hardwired control and micro programmed control. In the hardwired organization, the control logic is implemented with gates, flip-flops, decoders, and other digital circuits. It has the advantage that it can be optimized to produce a fast mode of operation. In the micro-programmed organization, the control information is stored in a control memory.

The control memory is programmed to initiate the required sequence of micro operations. A hardwired control as the name implies, requires changes in the wiring among the various components, if the design has to be modified or changed.

In the micro-programmed control, updating the micro program in control memory can do any required changes or modifications. A hardwired control for the basic computer is presented in the section.

The block diagram of the control unit is shown in Fig. 5-2. It consists of two decoders, a sequence counter, and a number of control logic gates. An instruction read from memory is placed in the instruction register (IR).

The instruction register is shown in Fig. 5-2, where it is divided into three parts: the I bit, the operation code, and bits 0 through 11. The operation code in bits 12 through 14 is decoded with a 3 * 8 decoder. The eight outputs of the decoder are designated by the symbols $D_0$ through $D_7$.

Bit 15 of the instruction is transferred to a flip-flop designated by the symbol I. Bits 0 through 11 are applied to the control logic gates.

The 4-bit sequence counter can count in binary from 0 through 15. The outputs of the counter are decoded into 16 timing signals $T_0$ through $T_{15}$.

The sequence counter SC can be incremented or cleared synchronously. Most of the time, the counter is incremented to provide the sequence of timing signals out of the 4 * 16 decoder. Once in a while, the counter is cleared to 0, causing the next active timing signal to be $T_0$.

As an example, consider the case where SC is incremented to provide timing signals $T_0$, $T_1$, $T_2$, $T_3$, and $T_4$ in sequence. At time T4, SC is cleared to 0 if decoder output $D_3$ is active. This is expressed symbolically by the statement $D_3T_4$: SC ←0

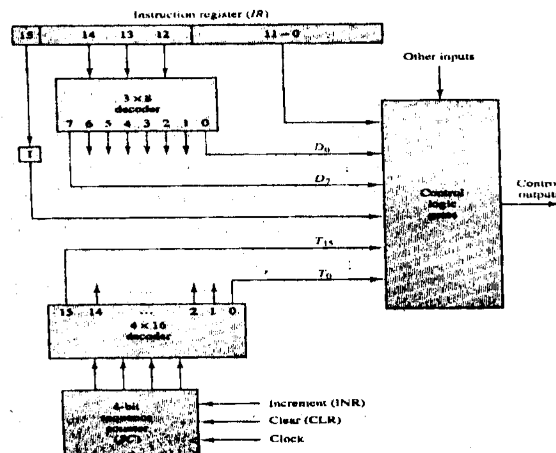



**Fig 5.2 Control Unit Of Basic Computer**

## 5.4 Execution of Instructions

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.

$T_0$: AR ← PC
$T_1$: IR ← M [AR], PC ← PC + 1
$T_2$: D0... D7 ← Decode IR (12-14),
AR IR ← (0-11), I ← IR (15)

**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.

Decoder output $D_7$ is equal to 1 if the operation code is equal to binary 111. From Fig.5-1, we determine that 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:

$D_7I\ T_3$: AR ← M [AR]
$D_7I'T_3$: Nothing
$D_7I'T_3$: Execute a register-reference instruction
$D_7IT_3$: Execute an input-output instruction

**Register-Reference Instructions :**

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

| | | | |
|---|---|---|---|
| | $r$: | $SC \leftarrow 0$ | Clear $SC$ |
| CLA | $rB_{11}$: | $AC \leftarrow 0$ | Clear $AC$ |
| CLE | $rB_{10}$: | $E \leftarrow 0$ | Clear $E$ |
| CMA | $rB_9$: | $AC \leftarrow \overline{AC}$ | Complement $AC$ |
| CME | $rB_8$: | $E \leftarrow \overline{E}$ | Complement $E$ |
| CIR | $rB_7$: | $AC \leftarrow \text{shr } AC,\ AC(15) \leftarrow E,\ E \leftarrow AC(0)$ | 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 |

**Table 5.2: execution of Register-Reference Instructions**

**TABLE 5.3: Memory-Reference Instructions**

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

The control functions and micro operations for the register-reference instructions are listed in Table 5.2. These instructions are executed with the clock transition associated with timing variable $T_3$. Each control function needs the Boolean relation $D_7$ I' $T_3$, which we designate for convenience by the symbol r. The control function is distinguished by one of the bits in IR (0-11). By assigning the symbol Bi to bit I of IR, all control functions can be simply denoted by rBi.

**Memory-Reference Instructions:** Table 5-3 lists the seven memory-reference instructions. The decoded output Di 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 $T_4$.

## 5.5 Input-Output And Interrupt

A computer can serve no useful purpose unless it communicates with the external environment. Instructions and data stored in memory must come from some input device. Computational results must be transmitted to the user through some output device.

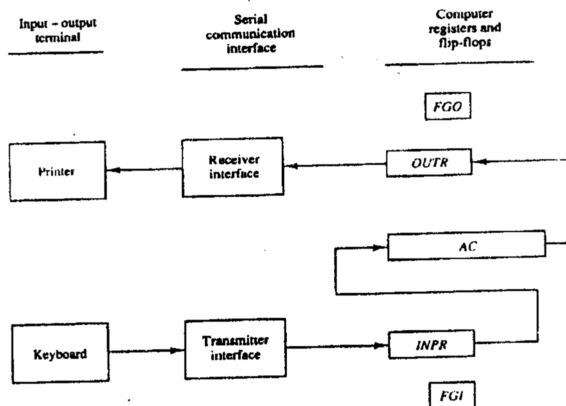



**Fig 5.3 : Input-output configuration**

The terminal sends and receives serial information. Each quantity of information has eight bits of an alphanumeric code. The serial information from the keyboard is shifted into the input register INPR. The serial information for the printer is stored in the output register OUTR. These two registers communicate with a communication interface serially and with the AC in parallel. The input-output configuration is shown in Fig. 5-3.

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.

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

The process of information transfer is as follows. Initially, the input flag FGI is cleared to 0. When a key is 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 computer checks the flag bit; if it is 1, the information from INPR is transferred in parallel into AC and FGI is cleared to 0. Once the flag is cleared, new information can be shifted into INPR by striking another key.

The output-register OUTR works similarly but the direction of information flow is reversed. Initially, the output flag FGO is set to 1.

The computer checks the flag bit; if it is 1, the information from AC 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 the operation is completed, it sets FGO to 1. The computer does not load a new character into OUTR when FGO is 0 because this condition indicates that the output device is in the process of printing the character

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

| | | | |
|---|---|---|---|
| | $p$: | $SC \leftarrow 0$ | Clear SC |
| INP | $pB_{11}$: | $AC(0\text{-}7) \leftarrow INPR, \quad FGI \leftarrow 0$ | Input character |
| OUT | $pB_{10}$: | $OUTR \leftarrow AC(0\text{-}7), \quad FGO \leftarrow 0$ | 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 |

**Table 5.4:Input-output Instructions**

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 5.4.

These instructions are executed with the clock transition associated with timing signal $T_3$. Each control function needs a Boolean relation $D_7IT_3$which we designate for convenience by the symbol p.

The control function is distinguished by one of the bits in IR (6-11), by assigning the symbol B, to bit I of IR, all control functions can be denoted by pB, for I = 6 though 11. The sequence counter SC is cleared to 0 when $p = D_7 I T_3 = 1$.

**Program Interrupt:** In programmed control transfer the process of communication in the computer keeps checking the flag bit, and when it finds it set, it initiates an information transfer. The difference of information flow rate between the computer and that of the input-output device

makes this type of transfer inefficient. The computer is wasting time while checking the flag instead of doing some other useful processing task.

An alternative to the programmed controlled procedure is to let the external device inform the computer when it is ready for the transfer. In the meantime the computer can be busy with other tasks. This type of transfer uses the interrupt facility. While the computer is running a program, it does not check the flags. However, when a flag is set, the computer is momentarily interrupted from proceeding with the current program and is informed of the fact that a flag has been set. The computer deviates momentarily from what it is doing to take care of the input or output transfer. It then returns to the current program to continue what it was doing before the interrupt.

The interrupt enable flip-flop IEN can be set and cleared with two instructions. When IEN is cleared to 0 (with the IOF instruction), the flags cannot interrupt the computer. When IEN is set to 1 (with the ION instruction), the computer can be interrupted. These two instructions provide the programmer with the capability of making a decision as to whether or not to use the interrupt facility.

The way that the interrupt is handled by the computer can be explained by means of the flowchart of Fig. 5-4.

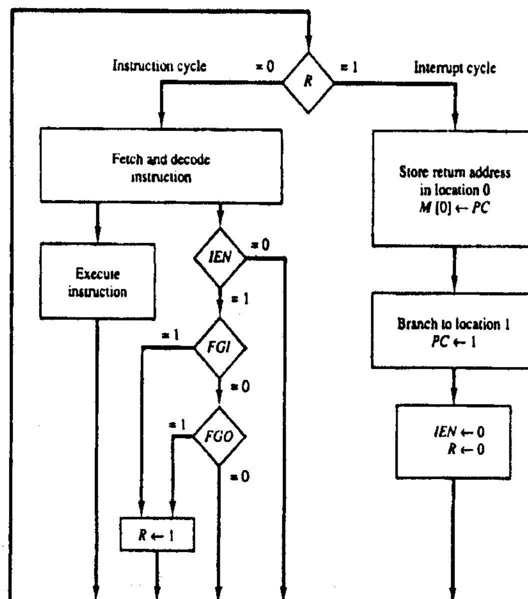



**Fig 5.4: Flowchart for Interrupt cycle**

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

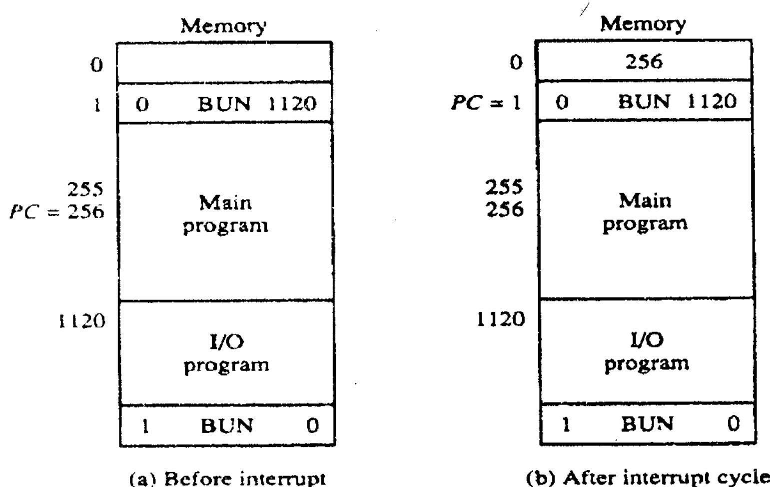
next instruction cycle. If IEN is 1, control checks the flag bits. If both flags are 0, it indicates that neither the input nor the output registers are ready for transfer of information. In this case, control continues with the next instruction cycle. If either flag is set to 1 while IEN = 1, flip-flop 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.

The interrupt cycle is a hardware implementation of a branch and save return address operation. The return address available in PC is stored in a specific location where it can be found later when the program returns to the instruction at which it was interrupted.

This location may be a processor register, a memory stack, or a specific memory location. Here we choose the memory location at address 0 as the place for storing the return address. Control then inserts address 1 into PC and clears JEN and R so that no more interruptions can occur until the interrupt request from the flag has been serviced.

An example that shows what happens during the interrupt cycle is shown in Fig. 5.5.

**Fig 5.5: Demonstration of the Interrupt cycle**




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 Fig. 5.5(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 JEN to 1 (to enable further interrupts), and the program returns to the location where it was interrupted. This is shown in Fig. 5-14(b). The instruction that returns the computer to the original place in the main program is a branch indirect instruction with an address part of 0. This instruction is placed at the end of the I/O service program. After this instruction is read from memory during the fetch phase, control goes to the indirect phase (because I == 1) to read the effective address. The effective address is in location 0 and is the return address that was stored there during the previous interrupt cycle. The execution of the indirect BUN instruction results in placing into PC the return address from location 0.

## 5.6 Summary

In this Chapter, we have discussed in detail about computer Instructions, timing and control, Execution of instruction i.e., the different phases in instruction cycle. After this we have discussed in detail about Input-Output configuration and program interrupt.

## 5.7 Model Questions

1. Discuss about Basic Computer Instruction formats.
2. Explain Instruction cycle in detail.
3. Draw a flowchart for Instruction cycle.
4. List out Register reference Instructions and explain in detail.
5. List out Memory reference Instructions and explain in detail.
6. List out Input-output Instructions and explain in detail.
7. Explain Input-output configuration in detail.
8. Draw a flowchart for Interrupt cycle and explain in detail.
9. What is the difference between a direct and an indirect address instruction. How many references to memory are needed for each type of instruction to bring an operand into a processor register.
10. A computer uses a memory unit with 256k words of 32 bits each. A binary instruction code is stored in one word of memory. The instruction has four parts, an indirect bit, an operation code, a register code part to specify one of 64 registers, and an address part.

- How many bits are there in the operation code, the register code part, and the address part.
- Draw the instruction word format and indicate the no. of bits in each part.
- How many bits are there in the data and address inputs of the memory.

## 5.8 REFERENCES

1. Computer System Architecture
   **By M.MORRIS MANO**
2. Computer Engineering: Hardware Design
   **By M.MORRIS MANO**

**Ms. T. ANURADHA, M.Sc., M.S.,**

Lesson 6

# Central Processing Organization-I

## 6.0 Objective :

This chapter describes the organization and architecture of the CPU. We briefly describe how the registers communicate with the ALU through buses and explain the operation of the memory stack. We then present the type of instruction formats available, the addressing modes used to retrieve data from memory.

## Structure of the Lesson

**6.1 Introduction**
**6.2 General Register Organization**
**6.3 Stack organization**
**6.4 Instruction formats**
**6.5 Addressing Modes**
**6.6 Summary**
**6.7 Model Questions**
**6.8 References**

## 6.1 Introduction

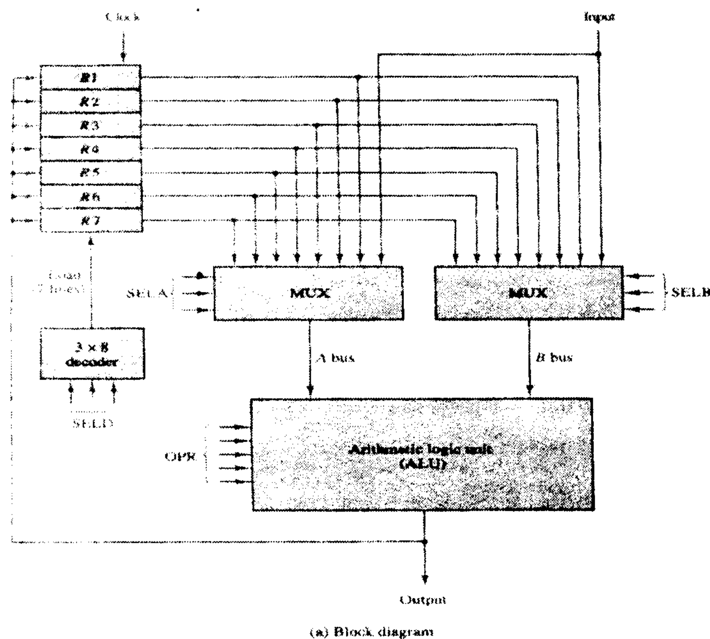
The part of the computer that performs the bulk of data-processing operations is called the central processing unit and is referred to as the CPU. The CPU is made up of three major parts. The register set stores intermediate data used during the execution of the instructions. The arithmetic logic unit (ALU) performs the required micro operations for executing the instructions. The control unit supervises the transfer of information among the registers and instructs the ALU as to which operations to perform.

Computer architecture includes the instruction formats and addressing modes.

## 6.2 General Register Organization

Memory locations are needed for storing pointers, counters, return addresses, temporary results, and partial products during multiplications. Referring to memory locations for such applications is time consuming because memory access is the most time-consuming operation in a computer. It is more convenient and more efficient to store these intermediate values in processor registers. When a large number of registers are included in the CPU, it is most efficient to connect them through a common bus system.

A bus organization for seven CPU registers is shown in the following figure 6.1(a)




(a) Block diagram

**Figure 6.1(a): Register Set With Common ALU**

The output of each register is connected to two multiplexers (MUX) to form the two buses A and B. The selection lines in each multiplexer select one register or the input data for the particular bus. The buses A and B form the inputs to a common arithmetic logic unit (ALU). The operation selected in the ALU determines the arithmetic or logic micro operation that is to be performed. The result of the micro operation is available for output data and also goes into the input of all the registers. A decoder selects the register that receives the information from the output bus. The decoder activates one of the register and inputs, thus providing a transfer path between the data in the output bus and the inputs of the selected destination register.

For example, to perform the operation

$$R1 \leftarrow R2 + R3$$

The control must provide binary selection variable to the following selector inputs:

1. MUX A selector (SELA): to place the content of R2 into bus A.
2. MUX B selector (SELB): to place the content of R3 into bus B.
3. ALU operation selector (OPR): to provide the arithmetic addition A + B.
4. Decoder destination selector (SELD): to transfer the content of the output bus into R1.

The four control selection variables are generated in the control unit and must be available at the beginning of a clock cycle.

**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 the following figure 6.1b

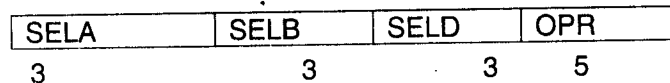



**Figure 6.1(b):Control Word**

It consists of four fields. Three fields contain three bits each, and one field has five bits. The three bits of SELA selects a source register for the A input of the ALU.

The three bits of SELB selects a register for the B input of the ALU. The three bits of SELD selects a destination register using the decoder and its seven load outputs.

The five bits of OPR selects one of the operations in the ALU. The 14-bit control word when applied to the selection inputs specifies a particular micro operation.

The encoding of the register selection is specified in following table 6.1.

| 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 6.1: Encoding of Register Selection Fields**

The three bit binary code listed in the first column of the table 6.1 specifies the binary code for each of the three fields. When SELA or SELB is 000 the corresponding multiplexer selects the external input data. When SELD=000, no destination register is selected but the content of the output bus are available in the external output.The encoding of the ALU operations for the CPU is specified in the following table 6.2.

| 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 | Compliment A | COMA |
| 10000 | Shift right A | SHRA |
| 11000 | Shift left A | SHLA |

**Table 6.2 : Encoding of ALU Operations**

The OPR field has five bits and each operation is designated with a symbolic name. The control words for some micro operations are listed in table 6.3.

| | Symbolic Designation | | | | |
|---|---|---|---|---|---|
| Microoperation | SELA | SELB | SELD | OPR | Control Word |
| R1 ← R2-R3 | R2 | R3 | R1 | SUB | 001 011 001 00101 |
| R4 ← R2 V 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 | 000 000 000 00000 |
| Output ← I/p | 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 6.3 : Examples of Microoperations for the CPU**

The most efficient way to generate control words with a large number of bits is to store them in a memory unit. A memory unit that stores control words is referred to as a control memory.

## 6.3 Stack Organization

A useful feature that is included in the CPU of most computers is a stack or last-in first-out (LIFO) list. A stack is a storage device that stores information in such a manner that the item stored list is the first item retrieved.

The stack in digital computers is essentially a memory unit with an address register. The register that holds the address for the stack is called stack pointer(SP) because its value always points at the top item in the stack.

The two operations of a stack are the insertion and deletion of items. The operation of insertion is called *push*. The operation of deletion is called *pop*. These operations are simulated by incrementing or decrementing the stack pointer register.

**Register Stack:** Stack can be placed in a portion of a large memory or it can be organized as a collection of finite number of memory words or registers.

Figure 6.2 shows the organization of 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.

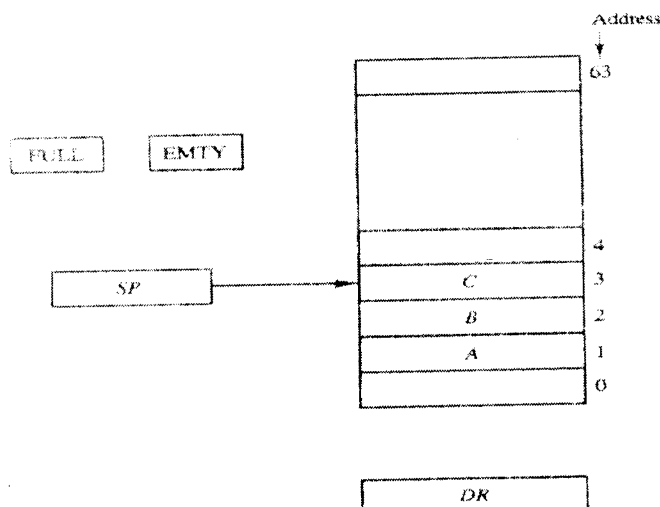



**Figure 6.2: Block diagram of a 64-word stack**

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 the 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.In a 64-word stack, the stack pointer contains 6 bits because $2^6$ =64. Since SP as only six bits, it cannot exceed a number greater than 63 (111111 in binary). When 63 is incremented by 1, the result is 0 since 111111 + 1 =1000000 in binary, but SP can accommodate on the six least significant bits. Similarly, when 000000 is decrement by 1, the result is 111111. The one bit register FULL is set to 1 when the stack is full, and the one-bit register EMTY is set to 1 when the stack is empty of items. DR is the data register that holds the binary data to be written into or read out of the stack.

Initially, SP is cleared to 0, EMTY is set to 1, and FULL is cleared to 0, 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:

| | |
|---|---|
| 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 |

A new item is declared from the stack if stack is not empty ( if EMTY = 0 ). The pop operatio consists of the following sequence of micro operations:

| | |
|---|---|
| DR ← M[SP] | Read item from the top of stack |
| SP ← SP-1 | Decrement stack pointer |
| If (SP = 0) then (EMTY ← 1) | Check if stack is empty |
| FULL ← 0 | Mark the stack not full |

**Memory Stack:** A stack can be implemented in a random access memory attached to a CPU. The implementation of stack in the CPU is done by assigning a portion of memory to a stack operation and using a processor register as a stack pointer.

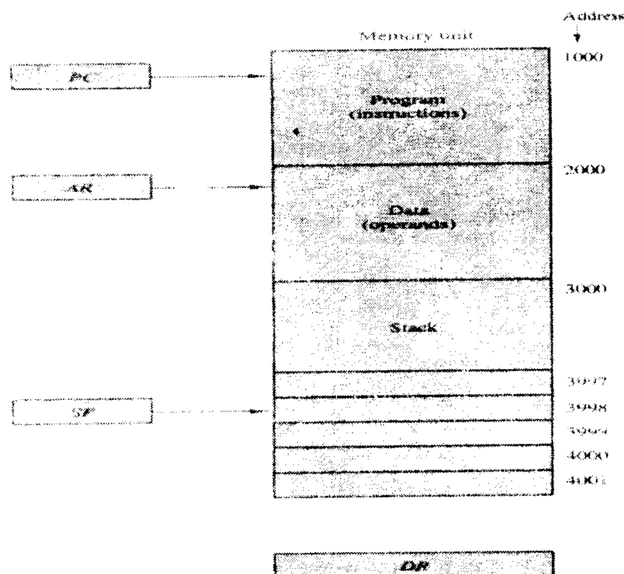



**Figure 6.3: Computer Memory With Program, Data, and Stack Segments**

A portion of computer memory is 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.

The initial value of SP is 4001 and the stack grows with decreasing addresses. Thus the first item stored in the stack is at address 4000,the second item is stored at 3999, and the last address that can be used for the stack is 3000. No provision is available for stack limit checks.

We assume that the item in the stack communicate with a data register DR. a new item is inserted with the push operation as follows:

SP ← SP-1
M[SP] ← DR

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.

**Reverse Polish Notation:** A stack organization is very effective for evaluating arithmetic expressions. The common arithmetic expressions are written in *infix,* notation with each operator written between the operands. Consider the simple arithmetic expression.

A * B + C * D

To evaluate arithmetic expressions in infix notation it is necessary to scan back and forth along the expression to determine the next operation to be performed.

Prefix notation often referred to as polish notation, places the operator before the operands. The *postfix notation* referred to as *reverse polish notation* places the operator after the operands. The following examples demonstrate the three representations:

| | |
|---|---|
| A+B | Infix notation |
| +AB | prefix or polish notation |
| AB+ | postfix or reverse polish notation |

The 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 *+

and is evaluated as follows: Scan the expression from left to right. When an operator is reached, perform the operation when the two operands found on the left side of the operator. Remove the two operands and the operator and replace them by the number obtained from the result of the operation. Continue to scan the expression and repeat the procedure for every operator encountered until there are no more operators.

The conversion from infix notation to reverse polish notation must take it to consideration the operational hierarchy adopted for infix notation. This hierarchy dictates that we first perform all arithmetic inside inner parentheses, then inside outer parentheses, and do multiplication and division operations. Consider the expression

(A + B)*[C * (D + E) + F]

the converted expression is

AB + DE + C * F + *

**Evaluation of Arithmetic Expressions:** Reverse polish notation, combined with a stack arrangement of registers, is the most efficient way known for evaluating arithmetic expressions. The procedure consists of first converting the arithmetic expression into its equivalent reverse polish notation. The operands are pushed into the stack in the order in which they appear.

1) The two top most operands in the stack are used for the operation, and
2) The stack is popped and the result of the operation replaces the lower operand.

By pushing the operands into stack continuously and performing the operations as defined above, the expression is evaluated in the proper order and the final result remains on the top of the stack.

The following numerical example may clarify this procedure. Consider the arithmetic expression

(3 * 4) + (5 * 6)

In reverse polish notation, is expressed as

34 * 56 * +

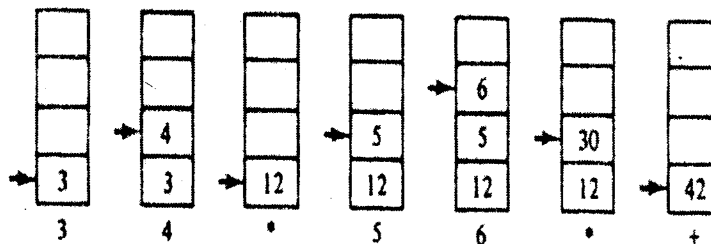



**Figure 6.4: Stack Operation To Evaluate 3 * 4 + 5 * 6**

Each box represents one stack operation and the arrow always points the top of the stack. Scanning the expression from left to right, we encounter two operands. First the number 3 is pushed into the stack, then the number 4. The next symbol is the multiplication operator *. This causes the multiplication of two top most items in the stack. The stack is then popped and the product is placed on the top of the stack, replacing the two original operands. Next we encounter the two operands 5 and 6, so they are pushed into the stack. The stack operation that results from the next * replaces these two numbers by their product. The last operation causes an arithmetic addition of the two topmost numbers in the stack to produce the final result of 42.

## 6.4 Instruction Formats

The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are:

1. An operation code field that specifies the operation to be performed.
2. An address field that designates a memory address or a processor registers.
3. A mode field that specifies the way the operand or the effective address is determined.

Other special fields are sometimes employed under certain circumstances, as for example a field that gives the number of shifts in a shift-type instruction. In this section we are concerned with the address field of an instruction format and consider the effect of including multiple address field in an instruction.

The number of address fields in the instruction format of computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organizations:

1. Single accumulator organization
2. General register organization
3. Stack organization.

In an accumulator type organization all operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field.

In general register type of organization the instruction format needs three register address fields.

*ADD R1, R2, R3*

The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers. Thus the instruction

*ADD R1, R2*

Would the operation R1 $\leftarrow$ R1+ R2.

General register type computer employ two or three address fields in their instruction format. Each address field may specify a processor register or a memory word.

Computers with stack organization would have PUSH and POP instructions, which require an address field. Thus the instruction

*PUSH X*

Will push the word at address X to the top of the stack. The stack pointer is updated automatically. There is no need to specify operands with an address field since all operands are implied to be in the stack. The arithmetic statement

$$X = (A + B) * (C + D)$$

Using zero, one, or three address instructions. We will use the symbols ADD, SUB, 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 address A, B, C and D, and result must be stored in memory address at X.

**Three-Address Instructions**

| | | |
|---|---|---|
| ADD | R1, A, B | R1 $\leftarrow$ M[A] + M[B] |
| ADD | R2, C, D | R2 $\leftarrow$ M[C] + M[D] |
| MUL | X, R1,R2 | M[X] $\leftarrow$ R1*R2 |

It is assumed that the computer has two processor registers, R1 and R2. The symbol M[A] denotes the operand at memory address symbolized by A.

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

**Two Address Instructions :** The program to evaluate X= (A + B)*(C + D) is as follows:

| | | |
|---|---|---|
| MOV | R1, A | R1 ← M[A] |
| ADD | R1, B | R1 ← R1+M[B] |
| MOV | R2, C | R2 ← M[C] |
| ADD | R2, D | R2 ← R2+M[D] |
| MUL | R1, R2 | R1 ← R1*R2 |
| MOV | X, R1 | M[A] ← R1 |

The MOV instruction moves or transfers the operands to and from memory and processor registers. The first symbol listed in an instruction is assumed to be both a source and the destination where the result of the operation is transferred.

**One-Address Instructions:** One-address instructions use an implied accumulator (AC) register for all data manipulation. The program to evaluate X= (A + B)*(C + D) is

| | | |
|---|---|---|
| LOAD | A | AC ← M[A] |
| ADD | B | AC ← AC+M[B] |
| STORE | T | M[C] ← AC |
| LOAD | C | AC ← M[C] |
| ADD | D | AC ← AC+M[D] |
| MUL | T | AC ← AC*M[T] |
| STORE | X | M[A] ← AC |

All operations are done between the AC register and a memory operand. T is the address of a temporary memory location required for storing the intermediate result.

**Zero-Address Instructions:** A stack organized computer does not use an address field for the instructions ADD and MUL. The PUSH and POP instructions, however, need an address field to specify the operand that communicates with the stack. The following program shows how X = (A + B)*(C + D) will be written for a stack-organized computer. (TOS stands for top of the stack.)

| | | |
|---|---|---|
| PUSH | A | TOS ← A |
| PUSH | B | TOS ← B |
| ADD | | TOS ← (A + B) |
| PUSH | C | TOS ← C |
| PUSH | D | TOS ← D |
| ADD | | TOS ← (C + D) |
| MUL | | TOS ← (C + D) * (A + B) |
| POP | X | M[X] ← TOS |

## 6.5 Addressing Modes :

Operation field of an instruction specifies the operation to be performed. This operation must be executed on some data stored in computer registers or memory words. The way the operands are chosen during program execution is dependent on the addressing mode of the instruction. The addressing mode specifies a rule for interpreting or modifying the address field of the instruction before the operand is actually referenced.

**Implied Mode:** In this mode the operands are specified implicitly in the definition of the instruction.

For example, the instruction "complement accumulator" is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction.

**Figure 6-5 Instruction format with mode field.**

| Opcode | Mode | Address |
|---|---|---|

In fact, all register reference instructions that use an accumulator are implied-mode instructions. Zero-address instructions in a stack-organized computer are implied-mode instructions since the operands are implied to be on top of the stack.

**Immediate Mode:** In this mode the operand is specified in the instruction itself. In other words, an immediate-mode instruction has an operand field rather than an address field. Immediate-mode instructions are useful for initializing registers to a constant value.

**Register Mode:** when the address field specifies a processor register, the instruction is said to be in the register mode. In this mode the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction. A k-bit field can specify any one of $2^k$ registers.

**Register Indirect Mode:** In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory.

In other words, the selected register contains the address of the operand rather than the operand itself.

**Auto increment or Auto decrement Mode:** This is similar to the register indirect mode except that the register is incremented or decremented after (or before) its value is used to access memory. When the address stored in the register refers to a table of data in memory, it is necessary to increment or decrement the register after every access to the table. This can be achieved by using the increment or decrement instruction. The effective address is defined to be the memory address obtained from the computation dictated by the given addressing mode.

**Direct Address Mode:** In this mode the effective address is equal to the address part of the instruction. The operand resides in memory and its address is given directly by the address field of the instruction.

**Indirect Address Mode:** In this mode the address field of the instruction gives the address where the effective address is stored in memory. Control fetches the instruction from memory and uses its address part to access memory again to read the effective address.

A few addressing modes require that the address field of the instruction be added to the content of a specific register in the CPU. The effective address in these modes is obtained from the following computation:

effective address = address part of instruction + content of CPU register

**Relative Address Mode:** In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address.

Relative addressing is often used with branch-type instructions when the branch address is in the area surrounding the instruction word itself.

**Indexed Addressing Mode:** In this mode the content of an index register is added to the address part of the instruction to obtain the effective address. The index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory. Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stored in the index register.

**Base Register Addressing Mode:** In this mode the content of a base register is added to the address part of the instruction to obtain the effective address. This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register.

The difference between the two modes is in the way they are used rather than in the way that they are computed. An index register is assumed to hold an index number that is relative to the address part of the instruction.

A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address. The base register-addressing mode is used in computers to facilitate the relocation of programs in memory.

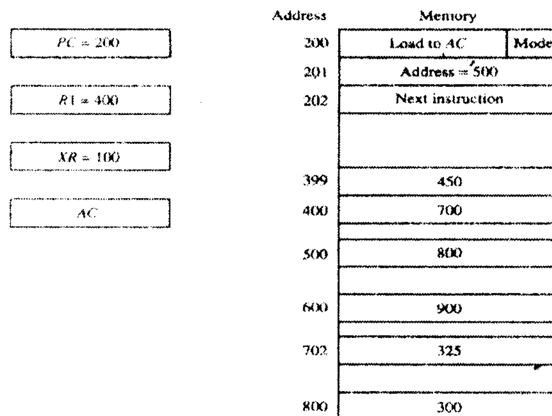



**Figure 6.6: Numerical example for addressing modes**

**Numerical Example**: To show the differences between the various modes, we will show the effect of the addressing modes on the instruction defined in Fig. 6.6. The two-word instruction at address 200 and 201 is a "load to AC" instruction with an address field equal to 500.

The first word of the instruction specifies the operation code and mode, and the second word specifies the address part. PC has the value 200 for fetching this instruction. The content of processor register RI is 400, and the content of an index register XR is 100. AC receives the operand after the instruction is executed. The mode field of the instruction can specify any one of a number of modes. For each possible mode we calculate the effective address and the operand that must be loaded into AC.

In the direct address mode the effective address is the address part of the instruction 500 and the operand to be loaded into AC is 800. In the immediate mode the second word of the instruction is taken as the operand rather than an address, so 500 is loaded into AC. (The effective address in this case is 201) In the indirect mode the effective address is stored in memory at address 500. Therefore, the effective address is 800 and the operand is 300. In the relative mode the effective address is 500 + 202 = 702 and the operand is 325.

(Note that the value in PC after the fetch phase and during the execute phase is 202.) In the index mode the effective address is XR+500=100+500=600 and the operand is 900. In the register mode the operand is in R1 and 400 is loaded into AC. (There is no effective address in this case.) In the register indirect mode the effective address is 400, equal to the content of RI and the operand loaded into AC is 700.

**Table 6.4: Tabular List of Numerical Example**

| Addressing Mode | Effective Address | Content of AC |
|---|---|---|
| Direct address | 500 | 800 |
| Immediate operand | 201 | 500 |
| Indirect address | 800 | 300 |
| Relative address | 702 | 325 |
| Indexed address | 600 | 900 |
| Register | — | 400 |
| Register indirect | 400 | 700 |
| Autoincrement | 400 | 700 |
| Autodecrement | 399 | 450 |

The Auto increment mode is the same as the register indirect mode except that RI is incremented to 401 after the execution of the instruction. The Auto decrement mode decrements RI to 399 prior to the execution of the instruction. The operand loaded into AC is now 450. Table 6-4 lists the values of the effective address and the operand loaded into AC for the nine addressing modes.

## 6.9 Summary

This completes our discussion on some topics of central Process organization. The information given on various topics such as various General Register organizations, stack organization, Instruction formats, Addressing modes.

## 6.10 Model Questions

1. Explain Stack organization in detail.
2. How to Evaluate Arithmetic Expressions.
3. Convert the following arithmetic expressions from infix to reverse polish notation.

   A * B + c * D + E * F

   A * B + A * ( B * D + C * E)

   A + B *[C * D + E * (F + G)]
4. Explai Instruction formats with examples.
5. Explain different Addressing Modes in detail.

## 6.11 References

1. Computer System Architecture
**by M.MORRIS MANO**

2.Computer Engineering: Hardware Design
**by M.MORRIS MANO**

**Ms. T.ANURADHA,** M. Sc., M.S.,

Lesson 7

# Central Processing Organization-II

## 7.0 Objective :

This chapter describes the data transfer manipulation instructions commonly incorporated in computers. Program control instructions and parallel processing which are used to denote a large class of techniques, that are used to provide simultaneous data processing tasks for the purpose of increasing the computational speed of a computer system.

## Structure of the Lesson


**7.1 Introduction**
**7.2 Data transfer Insructions**
**7.3 Data manipulation Instructions**
**7.3.1 Arithmetic Instructions**
**7.3.2 Logical and Bit Manipulation Instructions**
**7.3.3 Shift Instructions**
**7.4 Program control Instructions**
**7.5 Parallel processing**
**7.6 Summary**
**7.7 Model Questions**
**7.8 References**


## 7.1 Introduction

Computers provide an extensive set of instructions to give the user the flexibility to carry out various computational tasks. The instruction sets of different computers differ with each other mostly in the way the operands are determined from the address and mode fields. The actual operations available in the instruction set are not very different from one computer to another. It so happens that the binary code assignments in the operation code field are different in different computers, even for the same operation. It may also happen that the symbolic name given to instructions in the assembly language notation is different in different computers, even for the same instruction. The basic set of operations available in a typical computer is the subject covered in this chapter.

## 7.2 Data transfer Instructions

Most computer instructions can be classified into three categories:

1. Data transfer instructions
2. Data manipulation instructions
3. 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 themselves. Table 7.1gives a list of eight data transfer instructions used in many computers. Accompanying each instruction is a mnemonic symbol. Different computers use different mnemonics for the same instruction name.

**Table 7.1: Typical Data Transfer Instructions**

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

The load instruction has been used mostly to designate a transfer from memory to a processor register, usually an accumulator. The *store* instruction designates a transfer from a processor register into memory. The move instruction has been used in computers with multiple CPU registers to designate a transfer from one register to another. It has also been used for data transfers between CPU registers and memory or between two memory words. The exchange instruction swaps information between two registers or a register and a memory word. The *input* and *output* instructions transfer data among processor registers and input or output terminals. The *push* and *pop* instructions transfer data between processor registers and a memory stack.

Some assembly language conventions modify the mnemonic symbol to differentiate between the different addressing modes. For example, the mnemonic for *load immediate* becomes LDI. Other assembly language conventions use a special character to designate the addressing mode. For example, the immediate mode is recognized from a pound sign placed before the operand. In any case, the important thing is to realize that each instruction can occur with a variety of addressing modes. As an example, consider the *load to accumulator* instruction when used with eight different addressing modes.

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

**Table 7.2: Eight Addressing Modes for the Load Instruction**

Table 7.2 shows the recommended assembly language convention and the actual transfer accomplished in each case.

*ADR* stands for an address, NBR is a number or operand, X is an index register, *Rl* is a processor register, and AC is the accumulator register. The @ character symbolizes an indirect address. The $ character before an address makes the address relative to the program counter PC. The # character precedes the operand in an immediate-mode instruction.

A register that is placed in parentheses after the symbolic address recognizes an indexed mode instruction. The register mode is symbolized by giving the name of a processor register. In the register indirect mode, the name of the register that holds the memory address is enclosed in parentheses. The Auto increment mode is distinguished from the register indirect mode by placing a plus after the parenthesized register. The Auto decrement mode would use a minus instead. To be able to write assembly language programs for a computer, it is necessary to know the type of instructions available and also to be familiar with the addressing modes used in the particular computer.

## 7.3 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 instructions
3. Shift instructions

It must be realized, however, that each instruction when executed in the computer must go through the fetch phase to read its binary code value from memory. The operands

must also be brought into processor registers according to the rules of the instruction-addressing mode. The last step is to execute the instruction in the processor.

### 7.3.1 Arithmetic Instructions

The four basic arithmetic operations are addition, subtraction, multiplication, and division. Most computers provide instructions for all four operations. Some small computers have only addition and possibly subtraction instructions. The multiplication and division must then be generated by means of software subroutines. The four basic arithmetic operations are sufficient for formulating solutions to scientific problems when expressed in terms of numerical analysis methods.

A list of typical arithmetic instructions is given in Table7.3. The increment instruction adds 1 to the value stored in a register or memory word. One common characteristic of the increment operations when executed in processor registers is that a binary number of all 1's when incremented produces a result of all 0's. The decrement instruction subtracts 1 from a value stored in a register or memory word. A number with all 1's, when decremented, produces a number with all 1's.

Add, subtract, multiply, and divide instructions may be available for different types of data. The data type assumed to be in processor registers during the execution of these arithmetic operations is included in the definition of the operation code.

An arithmetic instruction may specify fixed-point or floating-point data, binary or decimal data, single-precision or double-precision data.

| Name | Mnemonic |
|---|---|
| Increment | INC |
| Decrement | DEC |
| Add | ADD |
| Subtract | SUB |
| Multiply | MUL |
| Divide | DIV |
| Add With Carry | ADDC |
| Subtract With Barrow | SUBB |
| Negate (2's Complement) | NEG |

**7.3 Arithmetic Instructions**

It is not uncommon to find computers with three or more add instructions one for binary integers, one for floating-point operands, and one for decimal operands. 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

A special carry flip-flop is used to store the carry from an operation. The instruction "add with carry" performs the addition on two operands plus the value of the carry from the previous computation.

Similarly, the "subtract with borrow" instruction subtracts two words and a borrow which may have resulted from a previous subtract operation. The negate instruction forms the 2's complement of a number, effectively reversing the sign of an integer when represented in the signed-2's complement form.

### 7.3.2 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 a 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 7.4. The clear instruction causes the specified operand to be replaced by 0's. The complement instruction produces the 1's complement by inverting all the bits of the operand. The AND, OR, and XOR instructions produce the corresponding logical operations on individual bits of the operands.

Although they perform Boolean operations, when used in computer instructions, the logical instructions should be considered as performing bit manipulation operations. Three bit manipulation operations are possible: a selected bit can be cleared to 0, or can be set to 1, or can be complemented. 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=0 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.

| Name | Mnemonic |
|---|---|
| Clear | CLR |
| Complement | COM |
| AND | AND |
| OR | OR |

| | |
|---|---|
| Exclusive-OR | XOR |
| Clear carry | CLRC |
| Set carry | SETC |
| Complement carry | COMC |
| Enable interrrupt | EI |
| Disable interrupt | DI |

**Table7.4: Typical Logical and Bit Manipulation Instructions**

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.

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

A few other bit manipulation instructions are included in Table7.4. Individual bits such as a carry can be cleared, set, or complemented with appropriate instructions. Another example is a flip-flop that controls the interrupt facility and is either enabled or disabled by means of bit manipulation instructions.

### 7.3.3 Shift Instructions

Shifts are operations in which the bits of a word are moved to the left or right. The bit shifted 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 7.5 lists four types of shift instructions. 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. The arithmetic shift-right instruction must preserve the sign bit in the leftmost position. The sign bit is shifted to the right together with the rest of the number, but the sign bit itself remains unchanged. This is a shift-right operation with the end bit remaining the same.

| 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 |

**Table 7.5: Typical Shift Instructions**

The arithmetic shift-left instruction inserts 0 to the end position and is identical to the logical shift-left instruction. For this reason many computers do not provide a distinct arithmetic shift-left instruction when the logical shift-left instruction is already available.

The rotate instructions produce a circular shift. Bits shifted out at one end of the word are not lost as in a logical shift but are circulated back into the other end. The rotate through carry instruction treats a carry bit as an extension of the register whose word is being rotated.

Thus a rotate-left through carry instruction transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit position into the carry, and at the same time, shifts the entire register to the left.

Some computers have a multiple-field format for the shift instructions. One field contains the operation code and the others specify the type of shift and the number of times that an operand is to be shifted. 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 fourbit field specifying shifts. With such a format, it is possible to specify the type of shift, the direction, and the number of shifts, all in one instruction.

## 7.4 Program Control Instructions

Instructions are always stored in successive memory locations. When processed in the CPU, the instructions are fetched from consecutive memory locations and executed. Each time an instruction is fetched from memory, the program counter is incremented so that it contains the address of the next instruction in sequence.

After the execution of a data transfer or data manipulation instruction, control returns to the fetch cycle with the program counter containing the address of the instruction next in sequence. On the other hand, a program control type of instruction, when executed, may change the address value in the program counter and cause the flow of control to be altered.

In other words, program control instructions specify conditions for altering the content of the program counter, while data transfer and manipulation instructions specify conditions for data processing operations.

The change in value of the program counter as a result of the execution of a program control instruction causes a break in the sequence of instruction execution. This is an important feature in digital computers, as it 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 7.6.

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

**Table 7.6: Typical Program Control Instructions**

The branch and jump instructions are used interchangeably to mean the same thing, but sometimes they are used to denote different addressing modes. The branch is usually a one-address instruction. It is written in assembly language as BR ADR, where ADR is a symbolic name for an address. When executed, the branch instruction causes a transfer of the value of ADR into the program counter. Since the program counter contains the address of the instruction to be executed, the next instruction will come from location ADR.

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 instruction will skip the next instruction if the condition is met. This is accomplished by incrementing the program counter during the execute phase in addition to its being incremented during the fetch phase. If the condition is not met, control proceeds with the next instruction in sequence where the programmer inserts an unconditional branch instruction. Thus a skip-branch pair of instructions causes a branch if the condition is not met, while a single conditional branch instruction causes a branch if the condition is met. The call and return instructions are used in conjunction with subroutines. The compare and test instructions do not change the program sequence directly.

They are listed in Table 7.6 because of their application in setting conditions for subsequent conditional branch instructions. The compare instruction performs a subtraction between two operands, but the result of the operation is not retained.

However, certain status bit conditions are set as a result of the operation. Similarly, the test instruction performs the logical AND of two operands and updates certain status bits without retaining the result or changing the operands. The status bits of interest are the carry bit, the sign bit, a zero indication, and an overflow condition. The generation of these status bits will be discussed first and then we will show how they are used in conditional branch instructions.

**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. The following figure 7.1 shows the block diagram of an 8-bit ALU with a 4-bit status register.

The four status bits are symbolized as 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 0.
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 0.
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. This is the condition for an overflow when negative numbers are in 2's complement. For the 8-bit ALU, V = 1 if the output is greater than +127 or less than -128.

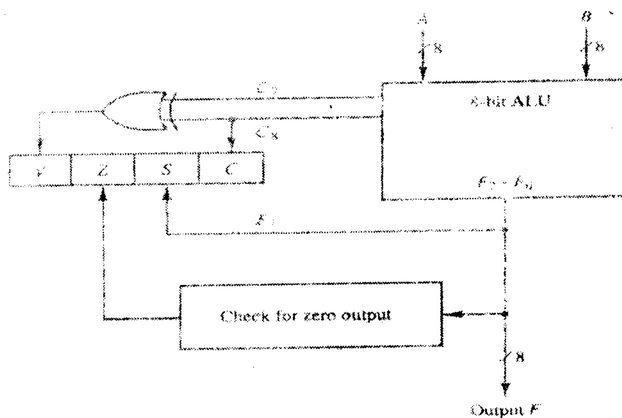



**Figure7.1: Status Register Bits**

The status bits can be checked after an ALU operation to determine certain relationships that exist between the values of A and B. If bit V is set after the addition of two signed numbers, it indicates an overflow condition. If Z is set after an exclusive-OR operation, it indicates that A = B. This is so because x XOR x = 0, and the exclusive-OR of two equal operands gives an all-0's result which sets the Z bit.

A single bit in A can be checked to determine if it is 0 or 1 by masking all bits except the bit in question and then checking the Z status bit. For example, let A = 101x1100, where *x* is the bit to be checked. The AND operation of A with B = 00010000 produces a result 000x0000. If *x* = 0, the Z status bit is set, but if x = 1, the Z bit is cleared since the result is not zero. The AND operation can be generated with the TEST instruction if the original content of A must be preserved.

**Conditional Branch Instructions:** Table 7.7 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.

**Table 7.7: 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>=B |
| BLO | Branch if lower | A<B |
| BLOE | Branch if lower or equal | A<=B |
| BE | Branch if equal | A==B |
| BNE | Branch if not equal | A!=B |

**Signed Compare Conditions (A – B)**

| BGT | Branch if greater than | A>B |
|---|---|---|
| BGE | Branch if greater or equal | A>=B |
| BLT | Branch if less than | A<B |
| BLE | Branch if less or equal | A<=B |
| BE | Branch if equal | A==B |
| BNE | Branch if not equal | A!=B |

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. It is also used in conjunction with the rotate instructions to check the bit shifted from the end position of a register into the carry position.

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. Therefore, a branch on plus checks for a sign bit of 0 and a branch on minus checks for a sign bit of 1.

It must be realized, however, that these two conditional branch instructions can be used to check the value of the most significant bit whether it represents a sign or not. The overflow bit is used in conjunction with arithmetic operations done on signed numbers in 2's complement representation.

As stated previously, the compare instruction performs a subtraction of two operands say, A-B. The result of the operation is not transferred into a destination register, but the status bits are affected. The status register provides information about the relative magnitude of *A* and B. Some computers provide conditional branch instructions that can be applied right after the execution of a compare instruction.

The specific conditions to be tested depend on whether, the two numbers A and B are considered to be unsigned or signed numbers. Table 7.7 gives a list of such conditional branch instructions. Note that we use the words higher and lower to denote the relations between unsigned numbers. Greater and less than for signed numbers.

The relative magnitude shown under the tested condition column in the table seems to be the same for unsigned and signed numbers. However, this is not the case since each must be considered separately as explained in the following numerical example.

The largest unsigned number that can be accommodated in 8 bits is 255. The range of signed numbers is between +127 and -128. The subtraction of two numbers is the same whether they are unsigned or in signed-2's complement representation.
Let A = 11110000 and B = 00010100. To perform A - B, the ALU takes the 2's complement of B and adds it to A.

A: 11110000

B + 1: +11101100

A - B: 11011100 C=l S = 1 V = 0 Z=0

The compare instruction updates the status bits, C = 1 because there is a carry out of the last stage, *S* = 1 because the leftmost bit is 1, V = 0 because the last two carries are both equal to 1, and Z = 0 because the result is not equal to 0.

If we assume unsigned numbers, the decimal equivalent of A is 240 and that of B is 20. The subtraction in decimal is 240 - 20 = 220. The binary result 11011100 is indeed the equivalent of decimal 220. Since 240 > 20, we have that A > B and A!=*B*. These two relations can also be derived from the fact that status bit C is equal to 1 and bit Z is equal to 0. The instructions that will cause, a branch after this comparison are BHI (branch if higher), BHE (branch if higher or equal), and BNE (branch if not equal).

If we assume signed numbers, the decimal equivalent of A is —16. This is because the sign of A is negative and 11110000 is the 2's complement of 00010000, which is the decimal equivalent of +16. The decimal equivalent of B is +20. The subtraction in decimal is (-16) - (+20) = -36. The binary result 11011100 (the 2's complement of 00100100) is indeed the equivalent of decimal -36.Since (-16)< (+20) we have that A<B and A != B. These two relations can also be derived from the fact that.status bits S = 1 (negative), *V* = 0 (no overflow), and Z = 0 (not zero).

The instructions that will cause a branch after this comparison are BLT (branch if less than), BLE (branch if less or equal), and BNE (branch if not equal).

It should be noted that the instruction BNE and BNZ (branch if not zero) are identical. Similarly, the two instructions BE (branch if equal) and BZ (branch if zero) are also identical. Each is repeated three times in Table 7.7 for the purpose of clarity and completeness.

**Subroutine Call and Return:** A subroutine is a self-contained sequence of instructions that performs a given computational task. During the execution of a program, a subroutine may be called to perform its function many times at various points in the main program. Each time a subroutine is called, a branch is executed to the beginning of the subroutine to start executing its set of instructions. After the subroutine has been executed, a branch is made back to the main program.

The instruction that transfers program control to a subroutine is known by different names. The most common names used are *call subroutine, jump to subroutine, branch to subroutine,* or *branch and save address.* A call subroutine instruction consists of an operation code together with an address that specifies the beginning of the subroutine.

The instruction is executed by performing two operations: (1) the address of the next instruction available in the program counter (the return address) is stored in a temporary location so the subroutine knows where to return, and (2) control is transferred to the beginning of the subroutine.

The last instruction of every subroutine, commonly called *return from subroutine,* transfers the return address from the temporary location into the program counter. Different computers use a different temporary location for storing the return address.

Some store the return address in the first memory location of the subroutine, some store it in a fixed location in memory, some store it in a processor register, and some store it in a memory stack. The most efficient way is to store the return address in a memory stack.

The advantage of using a stack for the return address is that when a succession of subroutines is called, the sequential return addresses can be pushed into the stack. 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:

$SP \leftarrow SP-1$ Decrement stack pointer

$M[SP] \leftarrow PC$ Push content of PC onto the stack

$PC \leftarrow$ 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:

$PC \leftarrow M[SP]$ Pop stack and transfer to PC

$SP \leftarrow SP + 1$ Increment stack Pointer

By using a subroutine stack, all return addresses are automatically stored by the hardware in one unit

**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.

The interrupt procedure is, in principle, quite similar to a subroutine call except for three variations: (1) The interrupt is usually initiated by an internal or external signal rather than from the execution of an instruction (except for software interrupt as explained later); (2) the address of the interrupt service program is determined by the hardware rather than from the address field of an instruction; and (3) an interrupt procedure usually stores all the information necessary to define the state of the CPU rather than storing only the program counter. These three procedural concepts are clarified further below.

After a program has been interrupted and the service routine been executed, the CPU must return to exactly the same state that it was when the interrupt occurred. Only if this happens will the interrupted program be able to resume exactly as if nothing had happened. The state of the CPU at the end of the execute cycle (when the interrupt is recognized) is determined from:

1. The content of the program counter
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. Typically, it includes the status bits from the last ALU operation and it specifies the interrupts that are allowed to occur and whether the CPU is operating in a supervisor or user mode. Many computers have a resident operating system that controls and supervises all other programs in the computer.

When the CPU is executing a program that is part of the operating system, it is said to be in the supervisor or system mode. Certain instructions are privileged and can be executed in this mode only. The CPU is normally in the user mode when executing user programs. The mode that the CPU is operating at any given time is determined from special status bits in the PSW.

Some computers store only the program counter when responding to an interrupt. The service program must then include instructions to store status and register content before these resources are used. Only a few computers store both program counter and all status and register content in response to an interrupt.

Most computers just store the program counter and the PSW. In some cases, there exist two sets of processor registers within the computer, one for each CPU mode. In this way, when the program switches from the user to the supervisor mode (or vice versa) in response to an interrupt, it is not necessary to store the contents of processor registers as each mode uses its own set of registers.

The hardware procedure for processing an interrupt is very similar to the execution of a subroutine call instruction. The state of the CPU is pushed into a memory stack and the beginning address of the service routine is transferred to the program counter. The beginning address of the service routine is determined by, the hardware rather than the address field of an instruction. Some computers assign one memory location where interrupts are always transferred. The service routine must then determine what caused the interrupt and proceed to service it. Some computers assign a memory location for each possible interrupt.

Sometimes, the hardware interrupt provides its own address that directs the CPU to the desired service routine. In any case, the CPU must possess some form of hardware procedure for selecting a branch address for servicing the interrupt.

The CPU does not respond to an interrupt until the end of an instruction execution. Just before going to the next fetch phase, control checks for any interrupt signals. If an interrupt is pending, control goes to a hardware interrupt cycle. During this cycle, the contents of PC and PSW are pushed onto the stack.

The branch address for the particular interrupt is then transferred to PC and a new PSW is loaded into the status register.

The service program can now be executed starting from the branch address and having a CPU mode as specified in the new PSW.The last instruction in the service program is, a *return from interrupt* instruction. When this instruction is executed, the stack is popped to retrieve the old PSW and the return address. The PSW is transferred to the status register and the return address to the program counter. Thus the CPU state is restored and the original program can continue executing.

**Types of Interrupts:** There are three major types of interrupts that cause a break in the normal execution of a program. They can be classified as:

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

External interrupts come from input-output (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 I/O device requesting transfer of data, I/O device finished transfer of data, elapsed time of an event, or power failure.

Timeout interrupt may result from a program that is in an endless loop and thus exceeded its time allocation. Power failure interrupt may have as its service routine a program that transfers the complete state of the CPU into a nondestructive memory in the few milliseconds before power ceases.

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.

These error conditions usually occur as a result of a premature termination of the instruction execution. The service program that processes the internal interrupt determines the corrective measure to be taken. The difference between internal and external interrupts is that the internal interrupt is initiated by some exceptional condition caused by the program itself rather than by an external event.

Internal interrupts are synchronous with the program while external interrupts are asynchronous. If the program is rerun, the internal interrupts will occur in the same place each time. External interrupts depend on external conditions that are independent of the program being executed at the time.

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.

The programmer to initiate an interrupt procedure at any desired point in the program can use it. 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. Certain operations in the computer may be assigned to the supervisor mode only, as for example, a complex input or output transfer procedure.

A program written by a user must run in the user mode. When an input or output transfer is required, the supervisor mode is requested by means of a supervisor call instruction. This instruction causes a software interrupt that stores the old CPU state and brings in a new PSW that belongs to the supervisor mode. The calling program must pass information to the operating system in order to specify the particular task requested.

## 7.5 Parallel Processing

Parallel Processing is a term used to denote a large class of techniques that are used to provide simultaneous data processing tasks for the purpose of increasing the computational speed of a computer system. Instead of processing each instruction sequentially as in a conventional computer parallel-processing system is able to perform concurrent data processing to achieve faster execution time.

For example, while an instruction is being executed in the ALU, the next instruction can be read from memory. The system may have two or more ALU's and be able to execute two or more instructions at the same time. Furthermore, the system may have two or more processors operating concurrently.

The purpose of parallel processing is to speed up the computer processing capability and increase it's throughput, i.e., the amount of processing that can be accomplished during a given interval of time. The amount of hardware increases with parallel processing and with it the cost of the system increases.

However, technological developments have reduced the hardware costs to the point where parallel processing techniques are economically feasible.

Parallel processing can be viewed from various levels of complexity. At the lowest level, we distinguish between parallel and serial operations by the type of registers used. Shift registers operate in a serial fashion one bit at a time, while registers with parallel load operate with all the bits of word simultaneously.

Parallel processing at a higher level of complexity can be achieved by having a multiplicity of functional units that perform identical or different operations simultaneously. Parallel processing established by distributing the data among the multiple functional units.

For example, the arithmetic, logic and shift operations can be separated into three units and the operands diverted to each unit under the supervision of a control unit.Figure7.2 shows one possible way of separating the execution unit into eight functional units operating in parallel. The operands in the registers are applied to one of the units depending on the operation specified by the instruction associated with the operands.

The operation performed in each functional unit is indicated in each block of the diagram. The adder and integer multiplier perform the arithmetic operations with integer numbers. The floating-point operations are separated into three circuits operating in parallel. The logic, shift and increment operations can be performed concurrently on different data.

All units are independent of each other, so one number can be shifted while another number is being incremented multifunctional organization is usually associated with a complex control unit to coordinate all the activities among the various components.

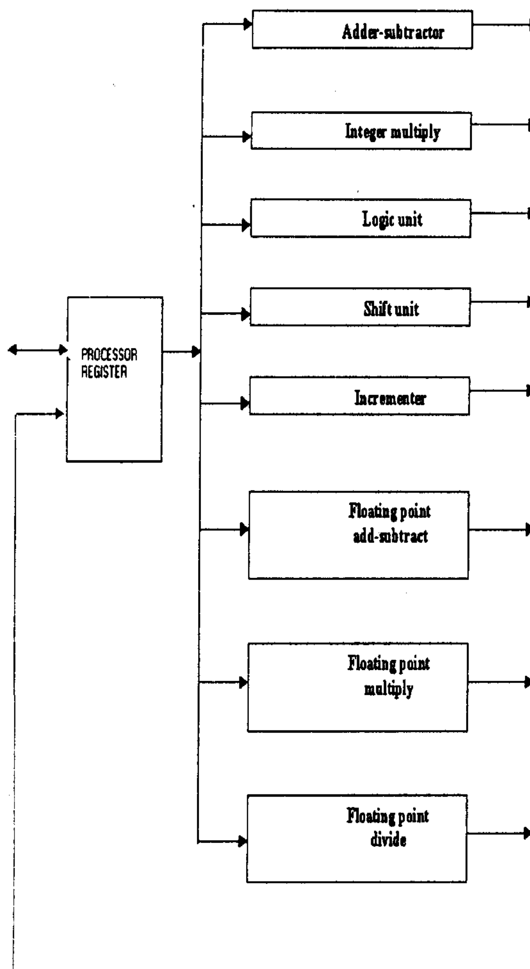
There are a variety of ways that parallel processing cab be classified. It can be considered from the internal organization of the processors, from the flow of information through the system. One classification introduced by M.J.Flynn considers the organization of a computer system by the number of instructions and data items that are manipulated simultaneously.

The normal operation of a computer is to fetch instructions from memory and execute them in a processor the sequence of instuctions read from memory constitutes an instruction stream.

The operations performed on the data in the processor, constitutes a data stream. Parallel processing may occur in the instruction stream, in the data stream or in both. Flynn's classification divides computers into four major groups.

Single instruction stream, single data stream (SISD)
Single instruction stream, multiple data stream (SIMD)
Multiple instruction streams, single data stream (MISD)
Multiple instruction streams, multiple data stream (MIMD)

**Figure: 7.2 Processor With Multiple Functional Units**




SISD represents the organization of a single computer containing a control unit, 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 common control unit. All processors receive the same instruction from the control unit operate on different types of data.

The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously. MISD structure is only theoretical interest since no practical system has been constructed using this organization. MIMD organization refers to a computer system capable of processing several programs at the same time.

Flynn's classification depends on the distinction between the performance of the control unit and the data processing unit. It emphasizes the behavioral characteristics of the computer system rather than its operational and structural interconnections. One type of parallel processing that does not fit Flynn's classification is pipelining. The only catagories used from the classification are SIMD array processors and MIMD multiprocessors.

We consider the parallel processing under the following main topics:

1.Pipeline processing
2.Vector processing
3.Array processors.

Pipeline processing is an implementation technique where arithematic sub-operations or the phases of a computer instruction cycle overlap in execution. Vector processing deals with computations involving large vectors and matrices. Array processors perform computations on large arrays of data.

## 7.6 Summary

This completes our discussion on Data transfer and manipulation instructions and Program control instructions.

In fact, a course in an area of computer Science must be supplemented by further readings to keep your knowledge up to date, as the computer world is changing with leaps and bounds.

In addition to further readings the student is advised to study several Indian Journals on computers to enhance his knowledge.

## 7.7 Model Questions

1. Explain Data tranfer Instructions.
2. Explain Arithmetic Instructions.
3. Explain Logical and Bit manipulation Instructions.
4. Explain Shift Instructions.
5. Explain Program Control Instructions.
6. Explain different types of Interrupts.
7. Explain Parallel processing.

## 7.8 REFERENCES

1. Computer System Architecture

**by M.MORRIS MANO**

2. Computer Engineering: Hardware Design
**by M.MORRIS MANO**

**Ms. T. ANURADHA,** M. Sc., M.S.,

Lesson 8:

# Arithmetic Processor Design And Arithmetic Algorithms - I

## 8.0 Objective :

The main objectives of this chapter are

- To perform addition with signed magnitude data and signed 2's compliment data.
- To perform subtraction with signed magnitude data and signed 2's compliment data.
- Multiplication of two binary numbers.
- Division of two binary numbers.

## Structure of the Lesson :


**8.1 Introduction**
**8.2 Addition And Subtraction With Signed-Magnitude Data**
**8.3 Addition And Subtraction With Signed-2's Complement Data**
**8.4 Multiplication Algcrithms:**
**8.4.1 Booth Multiplication Algorithm**
**8.4.2 Array Multiplier**
**8.5 Division Algorithms**
**8.6 Summary**
**8.7 Model Questic..s**
**8.8 References**


## 8.1 Introduction

Arithmetic instructions in digital computers manipulate data to produce results necessary for the solution of computational problems. These instructions perform arithmetic calculations. The four basic arithmetic operations are addition, subtraction, multiplication and division. From these four basic operations, it is possible to formulate other arithmetic functions and solve scientific problems by means of numerical analysis methods. An arithmetic processor is the part of a processor unit that executes arithmetic operations.

## 8.2 Addition And Subtraction With Signed- Magnitude Data

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-2's complement representation when performing arithmetic operations with integers.

The representation of numbers in signed-magnitude is familiar because it is used in everyday arithmetic calculations.

In this we designate the magnitude of the two numbers by A and B. When 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 8-1.

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 prevent a negative zero. In other words, when two equal numbers are subtracted, the result should be +0 not -0.

**Addition Algorithm:** When the signs of A and *B* are identical add the two magnitudes and attach the sign of A to the result. When the signs of A and B are different, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A if A > B or the complement of the sign of A if A < B. If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

**Subtraction Algorithm:** When the signs of A and *B* are different, add the two magnitudes and attach the sign of A to the result. When the signs of A and B are identical, compare the magnitudes and subtract the smaller number from the larger. Choose the sign of the result to be the same as A if A > B or the complement of the sign of A if A < B. If the two magnitudes are equal, subtract B from A and make the sign of the result positive.

| Operation | Add Magnitudes | Subtract Magnitudes | | |
|---|---|---|---|---|
| | | When $A > B$ | When $A < B$ | When $A = B$ |
| $(+A) + (+B)$ | $+(A + B)$ | | | |
| $(+A) + (-B)$ | | $+(A - B)$ | $-(B - A)$ | $+(A - B)$ |
| $(-A) + (+B)$ | | $-(A - B)$ | $+(B - A)$ | $+(A - B)$ |
| $(-A) + (-B)$ | $-(A + B)$ | | | |
| $(+A) - (+B)$ | | $+(A - B)$ | $-(B - A)$ | $+(A - B)$ |
| $(+A) - (-B)$ | $+(A + B)$ | | | |
| $(-A) - (+B)$ | $-(A + B)$ | | | |
| $(-A) - (-B)$ | | $-(A - B)$ | $+(B - A)$ | $+(A - B)$ |

**TABLE 8.1: Addition And Subtraction Of Signed-Magnitude Numbers**

**Hardware Implementation:** To implement the two arithmetic operations with hardware, it is first necessary that the two numbers be stored in registers. Let A and B be two registers that hold the magnitudes of the numbers, and $A_s$ and $B_s$ be two flip-flops that hold the corresponding signs. The result of the operation may be transferred to a third register or one of the source registers. Consider now the hardware implementation of the algorithms above.

First, a parallel-adder is needed to perform the micro operation A + B. Second, a comparator circuit is needed to establish if A >B, A = B, or A <B. Third, two parallel-subtractor circuits are needed to perform the micro operations A — B and B — A. The sign relationship can be determined from an exclusive-OR gate with $A_s$ and $B_s$ as inputs.

This procedure requires a magnitude comparator, an adder, and two subtractors. Another procedure which uses 2's complement for subtraction and comparison that requires only an adder and a complementer.

Figure 8-1 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 $A_s$ and $B_s$. 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 the two numbers. The add-overflow flip-flop *AVF* holds the overflow bit when A and B are added.

The addition of A plus B is done through the parallel adder. The S (sum) output of the adder is applied to the input of the A register. The complementer provides an output of B or the complement of B depending on the state of the mode control M. The complementer consists of exclusive-OR gates and the parallel adder consists of full-adder circuits.

The M signal is also applied to the input carry of the adder. When M = 0, the output of B is transferred to the adder, the input carry is 0, and the output of the adder is equal to the sum *A* + *B*. When *M* = 1, the 1's complement of *B is* applied to the adder, the input carry is 1, and output S = *A*+B+1. This is equal to A plus the 2's complement of B, which is equivalent to the subtraction A –B.

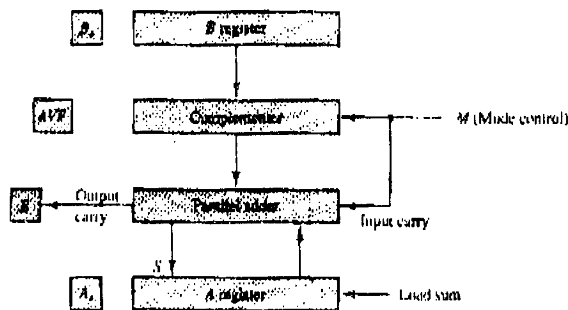



**Figure 8.1: Hardware For signed-Magnitude Addition And Subtraction**

**Hardware Algorithm:** The flowchart for the hardware algorithm is presented in Fig. 8-2. The two signs $A_s$ and $B_s$ 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 micro operation *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 complement of B.

No overflow can occur if the numbers are subtracted so *AVF* is cleared to 0. A 1 in E indicates that A >= B and the number in A is the correct result. If this number is zero, the sign $A_s$ must be made positive to avoid a negative zero. A 0 in E indicates that A < B. For this case it is necessary to take the 2's complement of the value in A. This operation can be done with one micro operation A ← A+1. However, we assume that the A register has circuits for micro

operations *complement* and *increment,* so the 2's complement is obtained from these two micro operations. In other paths of the flowchart, the sign of the result is the same as the sign of A, so no change in $A_s$ 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 $A_s$ to obtain the correct sign.

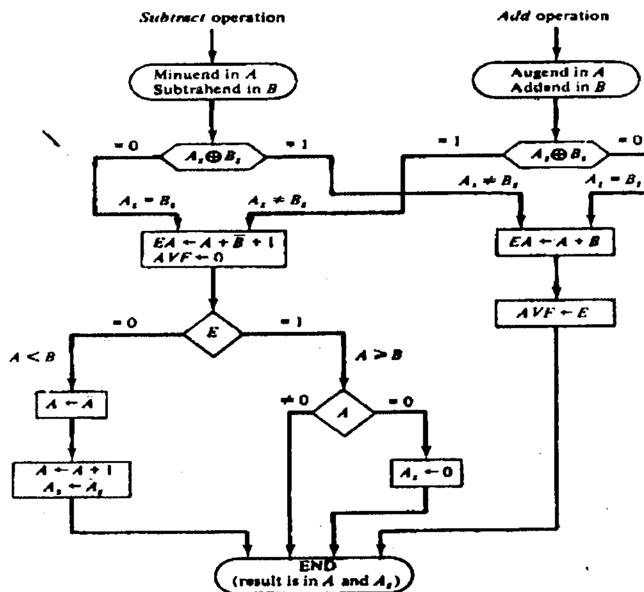



**Figure 8.2: Flowchart For Add And Subtract Operations**

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

## 8.3 Addition and Subtraction With Signed-2's Complement Data

The signed-2's complement representation of numbers together with arithmetic algorithms for addition and subtraction are summarized here for easy reference. The leftmost bit of a binary number represents the sign bit: 0 for positive and 1 for negative.

If the sign bit is 1, the entire number is represented in 2's complement form. Thus +33 is represented as 00100001 and -33 as 11011111. Note that 11011111, is the 2's complement of 00100001, and vice versa.

The addition of two numbers in signed-2's complement form consists of adding the numbers with the sign bits treated the same as the other bits of the number. A carry-out of the sign-bit position is discarded. The subtraction consists of first taking the 2's complement of the subtrahend and then adding it to the minuend.

When two numbers of $n$ digits each are added and the sum occupies $n + 1$ digits, we say that an overflow occurred. An overflow can be detected by inspecting the last two carries out of the addition. When the two carries are applied to an exclusive-OR gate, the overflow is detected

when the output of the gate is equal to 1.

The register configuration for the hardware implementation is shown in Fig. 8-3. This is the same configuration as in Fig. 8-1 except that the sign bits are not separated from the rest of the registers. We name the A register AC (accumulator) and the B register BR.

The leftmost bit in AC and *BR* represent the sign bits of the numbers. The two sign bits are added or subtracted together with the other bits in the complementer and parallel adder. The overflow flip-flop *V* is set to 1 if there is an overflow.

The algorithm for adding and subtracting two binary numbers in signed-2's complement representation is shown in the flowchart of Fig. 8-4. The sum is obtained by adding the contents of AC and *BR* (including their sign bits). The overflow bit *V* is set to 1 if the exclusive-OR of the last two carries is 1, and it is cleared to 0 otherwise.

The subtraction operation is accomplished by adding the content of AC to the 2's complement of *BR*. Taking the 2's complement of *BR* has the effect of changing a positive number to negative, and vice versa. An overflow must be checked during this operation because the two numbers added could have the same sign. The programmer must realize that if an overflow occurs, there will be an erroneous result in the AC register.

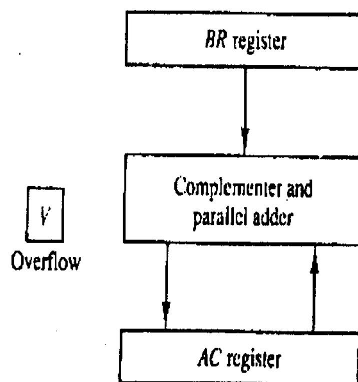



**Figure 8.3:Hardware For Signed-2's Complement Addition And Subtraction**

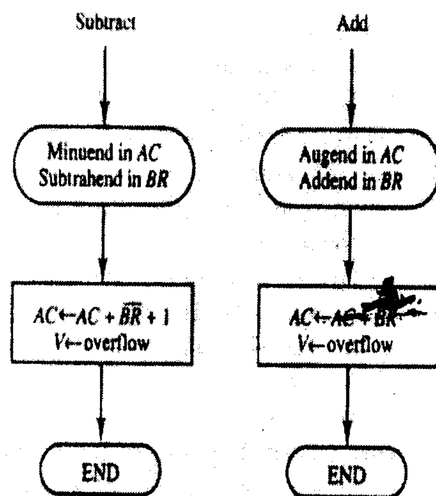



**Figure 8.4 : Algorithm For Adding And Subtracting Numbers In Signed-2's Complement Representation**

## 8.4 Multiplication Algorithms

Multiplication of two fixed-point binary numbers in signed -magnitude representation is.

```
23        10111      Multiplicand
19 X      10011      Multiplier
          10111
         10111
        00000   +
       00000
      10111              437      110110101
```

The process consists of looking at successive bits of the multiplier, least significant bit first. If the multiplier bit is a 1, the multiplicand is copied down; otherwise, zeros are copied down. The numbers copied down in successive lines are shifted one position to the left from the previous number. Finally, the numbers are added and their sum forms the product.

The sign of the product is determined from the signs of the multiplicand and multiplier. If they are alike, the sign of the product is positive. If they are not the same, the sign of the product is negative.

**Hardware Implementation For Signed-Magnitude Data:** When multiplication is implemented in a digital computer, it is convenient to change the process slightly.

First, instead of providing registers to store and add simultaneously as many binary numbers as there are bits in the multiplier, it is convenient to provide an adder for the summation of only two binary numbers and successively accumulate the partial products in a register.

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

The hardware for multiplication consists of the equipment shown in Fig. 8-1 plus two more registers. These registers together with registers A and B are shown in Fig. 8-5. The multiplier is stored in the Q register and its sign in $Q_s$.

The sequence counter SC is initially set to a number equal to the number of bits in the multiplier. The counter is decremented by 1 after forming each partial product. When the content of the counter reaches zero, the product is formed and the process stops.

Initially, the multiplicand is in register B and the multiplier in Q. The sum of A and B forms a partial product which is transferred to the EA register. Both partial product and multiplier are shifted to the right. This shift will be denoted by the statement shr EAQ to designate the right shift depicted in Fig. 8-5. The

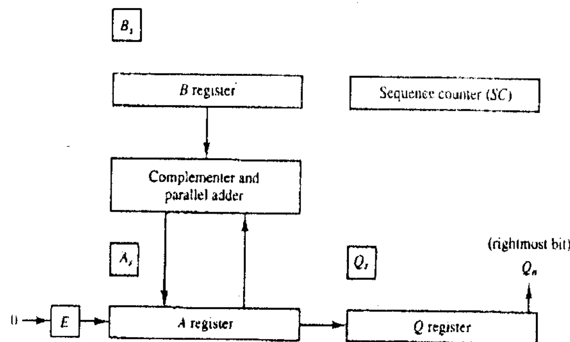



**Figure 8.5: Hardware For Multiplication Operation**

Least significant bit of A is shifted into the most significant position of Q, the bit from E is shifted into the most significant position of A, and 0 is shifted into E. After the shift, one bit of the partial product is shifted into Q, pushing the multiplier bits one position to the right. In this manner, the rightmost flip-flop in register Q, designated by $Q_n$, will hold the bit of the multiplier, which must be inspected next.

Hardware Algorithm: Figure 8-6 is a flowchart of the hardware multiply algorithm. Initially, the multiplicand is in B and the multiplier in Q. Their corresponding signs are in $B_S$ and $Q_s$, respectively. The signs are compared, and both A and Q are set to

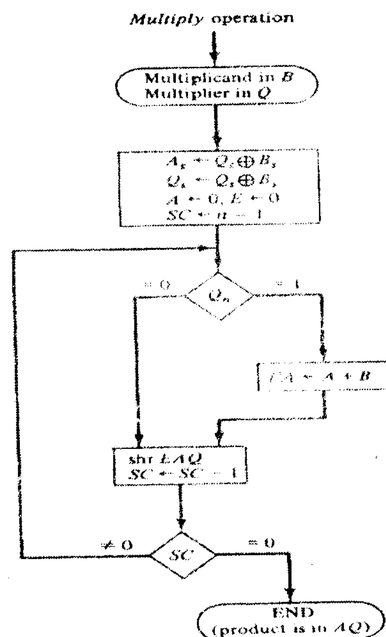



**Figure 8.6: Flowchart For Multiply Operation**

Correspond to the 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 a number equal to the number of bits of the multiplier. We are assuming here that operands are transferred to registers from a memory unit that has words of n bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of n - 1 bits. After the initialization, the low-order bit of the multiplier in $Q_n$ is tested. If it is a 1, the multiplicand in B is added to the present partial product in A.

If it is a 0, nothing is done. 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. The process stops when SC = 0. Note that the partial product formed in A is shifted into Q one bit at a time and eventually replaces the multiplier. The final product is available in both A and Q, with A holding the most significant bits and Q holding the least significant bits.

The previous numerical example is repeated in Table 8-2 to clarify the hardware multiplication process. The procedure follows the steps outlined in the flowchart.

### 8.4.1 Booth Multiplication 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 of 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.

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

**TABLE 8.2: Numerical Example For Binary Multiplier**

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

- The multiplicand is subtracted from the partial product upon encountering the first least significant 1 in a string of 1's in the multiplier.
- The multiplicand is added to the partial product upon encountering the first 0 (provided that there was a previous 1) in a string of 0's in the multiplier.
- The partial product does not change when the 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.

For example, a multiplier equal to —14 is represented in 2's complement as 110010 and is treated as $-2^4 + 2^2 - 2^1 = -14$. The hardware implementation of Booth algorithm requires the register configuration shown in Fig. 8-8. This is similar to Fig. 8-5 except that the sign bits are not separated from the rest of the registers.

To show this difference, we 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*. An extra flip-flop $Q_{n+1}$ is appended to *QR* to facilitate a double bit inspection of the multiplier. The flowchart for Booth algorithm is shown in Fig. 8-8. AC and the appended

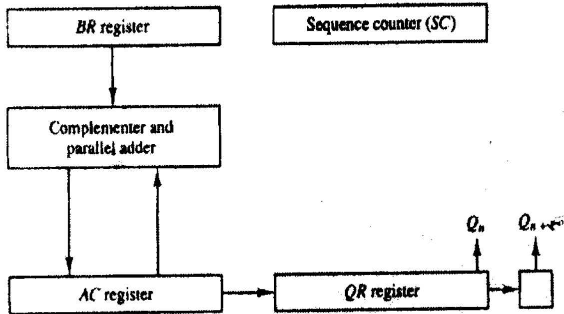



**Figure 8.7: Hardware For Booth Algorithm**

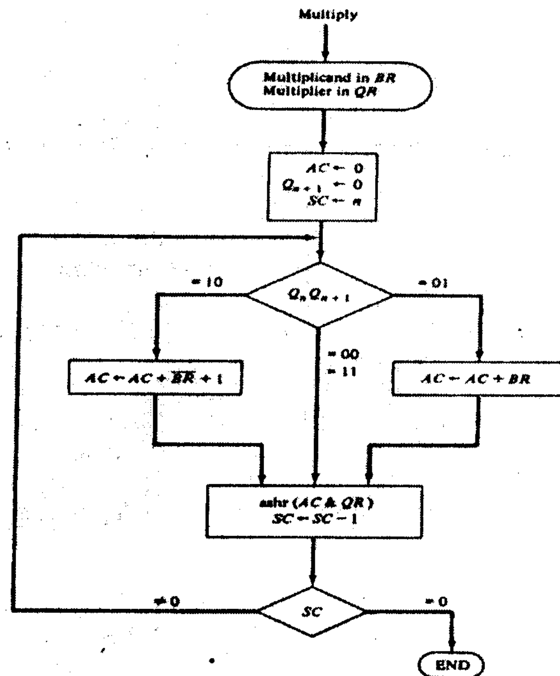



**Figure 8.8 : Booth Algorithm For Multiplication Of Signed-2's Complement Numbers**

Bit $Q_{n+1}$ are initially cleared to 0 and the sequence counter SC is set to a number *n* equal to the number of bits in the multiplier. The two bits of the multiplier in $Q_n$ and $Q_{n+1}$ 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. An overflow cannot occur because the addition and subtraction of the multiplicand follow each other. As a consequence, the two numbers that are added always have opposite signs, a condition that excludes an overflow.

The next step is to shift right the partial product and the multiplier (including bit $Q_{n+1}$). This is an arithmetic shift right (ashr) operation, which shifts AC and *QR* to the right and leaves the sign bit in AC unchanged.

The sequence counter is decremented and the computational loop is repeated *n* times.A numerical example of Booth algorithm is shown in Table 8-3 for *n* = 5: it shows the step-by-step multiplication of (-9) x (-13) = +118. Note that the multiplier in *QR* is negative and that the multiplicand in BR is also negative. The 10-bit product appears in AC and *QR* and is positive. The final value of $Q_{n+1}$ is the original sign bit of the multiplier and should not be taken as part of the product.

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

**Table 8.3: Example Of Multiplication With Booth Algorithm**

## 8.4.2 Array Multiplier

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

An array multiplier requires a large number of gates, and for this reason it was not economical until the development of integrated circuits.

To see how an array multiplier can be implemented with a combinational circuit, consider the multiplication of two 2-bit numbers as shown in Fig. 8-9. The multiplicand bits are b1 and *b0,* the multiplier bits are *a1* and a0, and the product is c3 c2 c1 c0.

The first partial product is formed by multiplying a0 by *b1 b0.* The multiplication of two bits such as a0 and *bo* produces a 1 if both bits are 1;otherwise, it produces a 0. This is identical to an AND operation and can be implemented with an AND gate.

As shown in the diagram, the first partial product is formed by means of two AND gates. The second partial product is formed by multiplying *a1* by b1 *b0* and is shifted one position to the left. The two partial products are added with two half-adder (HA) circuits.

Usually, there are more bits in the partial products and it will be necessary to use full-adders to produce the sum. Note that the least significant bit of the product does not have to go through an adder since it is formed by the output of the first AND gate. A combinational circuit binary multiplier with more bits can be constructed in a similar fashion.

A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier. The binary output in each level of AND gates are added in parallel with the partial product of the previous level to form a new partial product. The last level produces the product.

For j multiplier bits and *k* multiplicand bits we need *j x k* AND gates and (j-1) *k-bit* adders to produce a product of j+ *k* bits. As a second example, consider a multiplier circuit that multiplies a binary number of four bits with a number of three bits. Let the multiplicand be represented by $b_3b_2b_1b_0$ and the multiplier by $a_2a_1a_0$. Since k = 4 and j = 3, we need 12 AND gates and two 4-bit adders to produce a product of seven bits. The logic diagram of the multiplier is shown in Fig. 8-10.

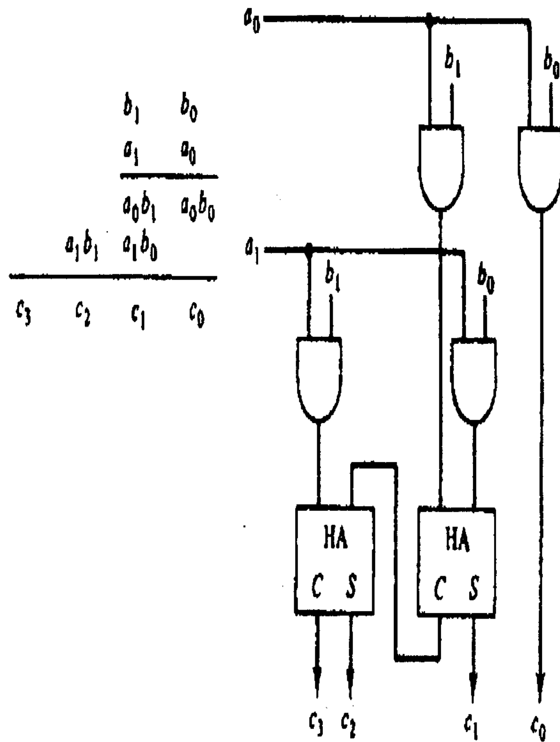



**Figure 8.9: 2-Bit by 2-Bit Array Multiplier**

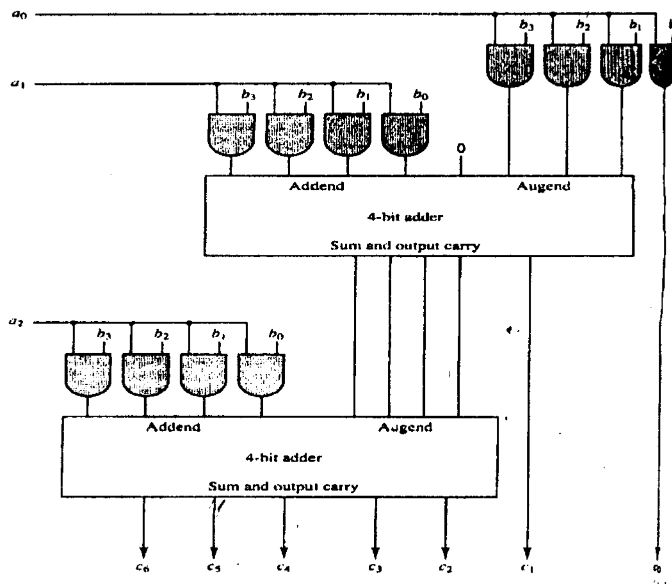



**Figure 8.10: 4-Bit By 3-Bit Array Multiplier**

## 8.5 Division Algorithms

Division of two fixed-point binary numbers in signed-magnitude representation is done with paper and pencil by a process of successive compare, shift, and subtract operations. Binary division is simpler than decimal division because 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 illustrated by a numerical example in Fig. 8-11. The divisor *B* consists of five bits and the dividend A, of ten bits. The five most significant bits of the dividend are compared with the divisor.

Since the 5-bit number is smaller than B, we try again by taking the six most significant bits of *A* and compare this number with B. The 6-bit number is greater than B, so we place a 1 for the quotient bit in the sixth position above the dividend.

The divisor is then shifted once to the right and subtracted from the dividend. The difference is called a *partial remainder* because the division could have stopped here to obtain a quotient of 1 and a remainder equal to the partial remainder. The process is continued by comparing a partial remainder with the divisor. 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. Note that the result gives both a quotient and a remainder.

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

**Figure 8-11: Example Of Binary Division**

**Hardware Implementation For Signed-Magnitude Data:** When the division is implemented in a digital computer, it is convenient to change the process slightly. Instead of shifting the divisor to the right, the dividend, or partial remainder, is shifted to the left, thus leaving the two numbers in the required relative position. Subtraction may be achieved by adding *A to* the 2's complement of B. The information about the relative magnitudes is then available from the end-carry.

The hardware for implementing the division operation is identical to that required for multiplication and consists of the components shown in Fig. 8-5. Register *EAQ* is now shifted to the left with 0 inserted into $Q_n$ and the previous value of E lost. The numerical example is repeated in Fig. 8-12 to clarify the proposed division process.

The divisor is stored in the B register and the double-length dividend is stored in registers A and Q. The dividend is shifted to the left and the divisor is subtracted by adding its 2's complement value. The information about the relative magnitude is available in E. If E = 1, it signifies that A >= B. A quotient bit 1 is inserted into *Qn* and the partial remainder is shifted to the left to repeat the process.

If E = 0, it signifies that A < B so the quotient in $Q_n$ remains a 0 (inserted during the shift). The value of B is then added to restore the partial remainder in A to its previous value. The partial remainder is shifted to the left and the process is repeated again until all five quotient bits are formed. Note that while the partial remainder is shifted left, the quotient bits are shifted also and after five shifts, the quotient is in Q and the final remainder is in A.

Before showing the algorithm in flowchart form, we have to consider the sign of the result and a possible overflow condition. The sign of the quotient is determined from the signs of the dividend and the divisor. If the two signs are alike, the sign of the quotient is plus. If they are unalike, the sign is minus. The sign of the remainder is the same as the sign of the dividend.

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 | |

**Figure 8.12: Example Of Binary Division With Digital Hardware**

**Divide Overflow:** The division operation may result in a quotient with an overflow. This is not a problem when working with paper and pencil but is critical when the operation is implemented with hardware. This is because the length of registers is finite and will not hold a number that exceeds the standard length. To see this, consider a system that has 5-bit registers. We use one register to hold the divisor and two registers to hold the dividend. From the example of Fig. 8-11 we note that the quotient will consist of six bits if the five most significant bits of the dividend constitute a number greater than the divisor.

The quotient is to be stored in a standard 5-bit register, so the overflow bit will require one more flip-flop for storing the sixth bit. This divide-overflow condition must be avoided in normal computer operations because the entire quotient will be too long for transfer into a memory unit that has words of standard length, that is, the same as the length of registers. Provisions to ensure that this condition is detected must be included in either the hardware or the software of the computer, or in a combination of the two.

When the dividend is twice as long as the divisor, the condition for overflow can be stated as follows: A divide-overflow condition occurs if the high-order half bits of the dividend constitute a number greater than or equal to the divisor.

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

**Hardware Algorithm:** The hardware divide algorithm is shown in the flowchart of Fig. 8-13. The dividend is in A and Q and the divisor in B. The sign of the result is transferred into Q, to be part of the quotient. A constant is set into the sequence counter SC to specify the number of bits in the quotient. As in multiplication, we assume that operands are transferred to registers from a memory unit that has words of *n* bits. Since an operand must be stored with its sign, one bit of the word will be occupied by the sign and the magnitude will consist of $n - 1$ bits.

A divide-overflow condition is tested by subtracting the divisor in B from half of the bits of the dividend stored in A.

If A >= B, the divide-overflow flip-flop *DVF* is set and the operation is terminated prematurely. If A < B, no divide overflow occurs so the value of the dividend is restored by adding B to A.

The division of the magnitudes starts by shifting the dividend in AQ to the left with the high-order bit shifted into E. If the bit shifted into E is 1, we know that EA > B because EA consists of a 1

followed by *n -1* bits while B consists of only *n-1* bits. In this case, B must be subtracted from EA and 1 inserted into $Q_n$ for the quotient bit. Since register A is missing the high-order bit of the dividend (which is in E), its value is $EA - 2^{n-1}$. Adding to this value the 2's complement of B results in

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

The carry from this addition is not transferred to E if we want E to remain a 1.
If the shift-left operation inserts a 0 into E, the divisor is subtracted by adding its 2's complement value and the carry is transferred into E. If E = *1,* it signifies that A >= B; therefore, $Q_n$ is set to 1.

If E = 0, it signifies that A < B and the original number is restored by adding B to A. In the latter case we leave a 0 in $Q_n$ (0 was inserted during the shift).

This process is repeated again with register A holding the partial remainder. After n-1 times, the quotient magnitude is formed in register Q and the remainder is found in register A. The quotient sign is in Q, and the sign of the remainder in A, is the same as the original sign of the dividend.

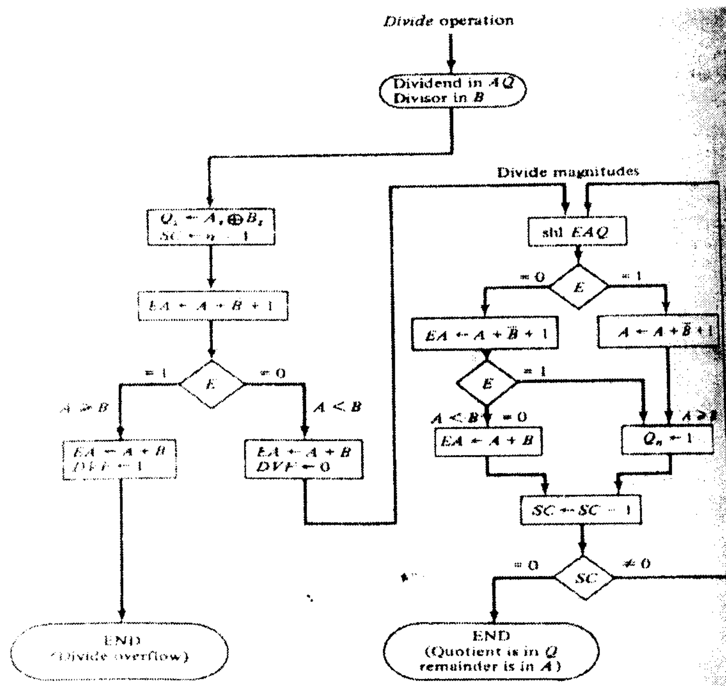



**Figure 8-13: Flowchart For Divide Operation**

## 8.6 Summary

In this chapter we develop the various arithmetic algorithms and show the procedure for implementing them with digital hardware. We consider addition, subtraction, multiplication, and division for the following types of data. Fixed - point binary data in signed-magnitude representation. Fixed–point binary data in signed-2's complement representation.

## 8.7 Model Questions

1. Derive an algorithm in flowchart form for adding and subtracting two fixed-point binary numbers when negative numbers are in signed-magnitude representation.
2. Show the hardware to be used for addition and subtraction of signed magnitude data.
3. Derive an algorithm in flowchart form for adding and subtracting two fixed-point binary numbers when negative numbers are in Signed-2's complement representation.
4. Show the hardware to be used for addition and subtraction of signed-2's complement data.
5. Show the hardware to be used for multiplication of signed magnitude data.
6. Explain booth Multiplication algorithm with example.
7. Design a 2-bit by 2-bit array multiplier.
8. Design a 4-bit by 3-bit array multiplier.
9. Show the hardware to be used for division of signed magnitude data.
10. Explain flowchart for divide operation in detail.
11. Show the hardware to be used for multiplication of signed magnitude data.

## 8.8 References

**Computer System Architecture**
**by M.MORRIS MANO**
Computer Organization **by HAMACHER**

**Ms. T. ANURADHA, M.Sc., M.S,**

When two normalized mantissas are added, *the* sum may contain an overflow digit. An overflow can be corrected easily by shifting the sum once to the right and incrementing the exponent.

When two numbers are subtracted, the result may contain most significant zeros as shown in the following example:

$$
\begin{array}{r}
.56780*10^5 \\
-.56430*10^5 \\
\hline
.00350*10^5 \\
\hline
\end{array}
$$

A floating point number that has a 0 in the most significant position of the mantissa is said to have an *underflow*. To normalize a number that contains an underflow, it is necessary to shift the mantissa to the left and decrement the exponent until a non-zero digit appears in the first position. In the example above, it is necessary to shift left twice to obtain $.35000 \times 10^3$. In most computers, a normalization procedure is performed after each operation to ensure that all results are in a normalized form.

Floating-point multiplication and division do not require an alignment of the mantissas. The product can be formed by multiplying the two mantissas and adding the exponents. Division is accomplished by dividing the mantissas and subtracting the exponents.

The operations performed 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 compare and increment (for aligning the mantissas), add and subtract (for multiplication and division), and decrement (to normalize the result). The exponent may be represented in any one of the three representations: signed-magnitude, signed-2's complement, or signed- 1's complement.

A fourth representation employed in many computers is known as a *biased* exponent. In this representation, the sign bit is removed from being a separate entity. The bias is a positive number that is added to each exponent as the floating-point number is formed, so that internally all exponents are positive. The following example may clarify this type of representation.

Consider an exponent that ranges from -50 to 49. Internally, it is represented by two digits (without a sign) by adding to it a bias of 50. The exponent register contains the number $e + 50$, where $e$ is the actual exponent. This way, the exponents are represented in registers as positive numbers in the range of 00 to 99. Positive exponents in registers have the range of numbers from 99 to 50. The subtraction of 50 gives the positive values from 49 to 0. Negative exponents are represented in registers in the range from 49 to 00. The subtraction of 50 gives the negative values in the range of —1 to —50.

The advantage of biased exponents is that they contain only positive numbers. It is then simpler to compare their relative magnitude without being concerned with their signs. As a consequence, a magnitude comparator can be used to compare their relative magnitude during the alignment of the mantissa. 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. In the examples above, we used decimal numbers to demonstrate some of the concepts that must be understood when dealing with floating-point numbers.

Obviously, the same concepts apply to binary numbers as well. The algorithms developed in this section are for binary numbers.

Decimal computer arithmetic is discussed in the next section.

### 9.2.1 Register Configuration

The register configuration for floating-point operations is quite similar to the layout for fixed-point operations. 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. 9-1. There are three registers, 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 uses the corresponding lowercase letter symbol.

It is assumed 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 $A_S$, and a magnitude that is in A. The exponent is in the part of the register denoted by the lowercase letter symbol *a*. The diagram shows explicitly the most significant bit of A, labeled by $A_1$. The bit in this position must be a 1 for the number to be normalized.

Note that the symbol AC represents the entire register, that is, the concatenation of $A_s$, A, and *a*. Similarly, register *BR* is subdivided into $B_s$, B, and *b,* and *QR* into $Q_s$, Q, and q. A parallel-adder adds the two mantissas and transfers the sum into A and the carry into E.

A separate parallel-adder is used for the exponents. Since the exponents are biased, they do not have a distinct sign bit but are represented as a biased positive quantity. It is assumed that the floating-point numbers are so large that the chance of an exponent overflow is very remote, and for this reason 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 the 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 are assumed to be initially normalized. After each arithmetic operation, the result will be normalized. Thus all floating-point operands coming from and going to the memory unit are always normalized.

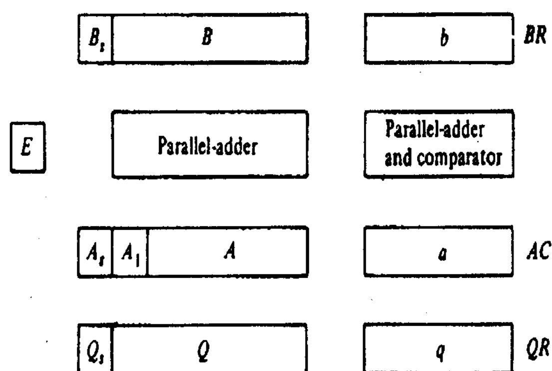


**Fig. 9-1: Registers For Floating-Point Arithmetic Operations**

### 9.2.2 Addition and Subtraction

During addition or subtraction, the two floating-point operands are 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 that is zero cannot be normalized. 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 unnormalized. The normalization procedure ensures that the result is normalized prior to its transfer to memory.

The flowchart for adding or subtracting two floating-point binary numbers is shown in Fig. 9-2. If *BR* is equal to zero, the operation is terminated, with the value in the AC being the result. If AC is equal to zero, we transfer the content of *BR* into *AC* and also complement its sign if the numbers are to be subtracted. If neither number is equal to zero, we proceed to align the mantissas. The magnitude comparator attached to exponents *a* and *b* provides three outputs that indicate their relative magnitude. If the two exponents are equal, we go to perform the arithmetic operation.

If the exponents are not equal, the mantissa having the smaller exponent is shifted to the right and its exponent incremented. This process is repeated until the two exponents are equal.

The addition and subtraction of the two mantissas is identical to the fixed-point addition and subtraction algorithm. The magnitude part is added or subtracted depending on the operation and the signs of the two mantissas.

If an overflow occurs when the magnitudes are added, it is transferred into flip-flop E. If E is equal to 1, the bit is transferred into $A_1$ and all other bits of A are shifted right. The exponent must be incremented to maintain the correct number. No underflow may occur in this case because the original mantissa that was not shifted during the alignment was already in a normalized position.

If the magnitudes were subtracted, the result may be zero or may have an underflow. If the mantissa is zero, the entire floating-point number in the *AC* is made 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 $A_1$ is 0.

In that case, the mantissa is shifted left and the exponent decremented. The bit in $A_1$ is checked again and the process is repeated until it is equal to 1. When $A_1 = 1$, the mantissa is normalized and the operation is completed.

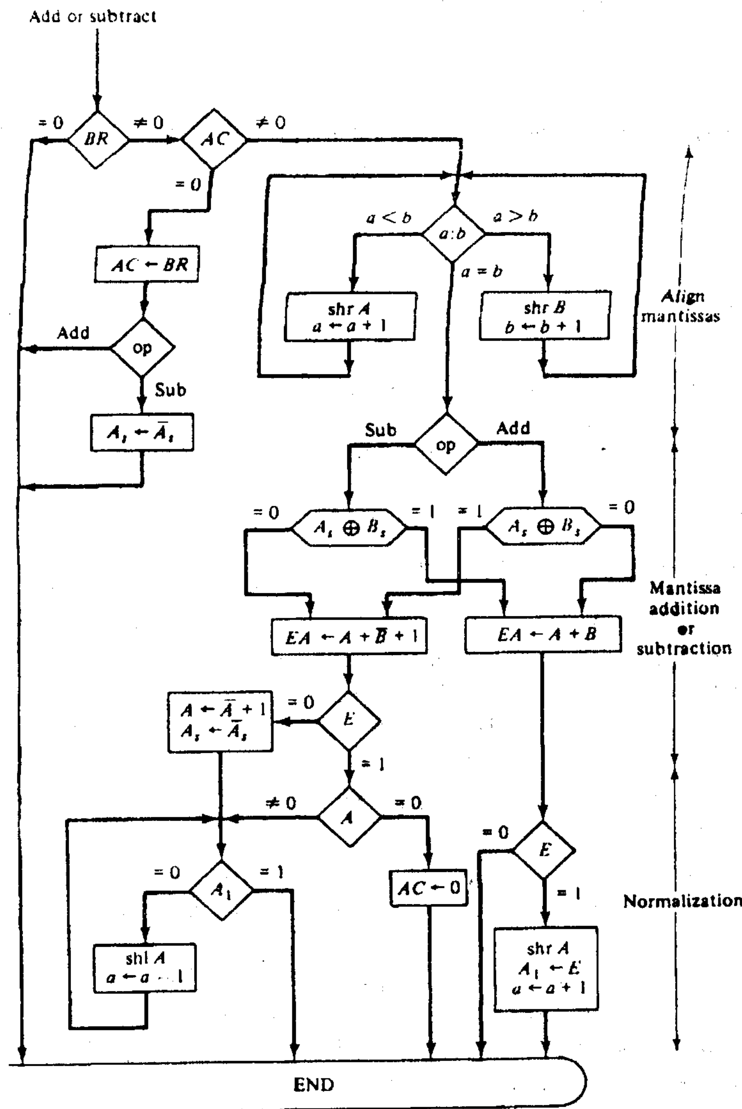



**Fig. 9-2: Addition And Subtraction Of Floating-Point Numbers**

### 9.2.3 Multiplication

The multiplication of two floating-point numbers requires that we multiply the mantissas and add the exponents. No comparison of exponents or alignment of mantissas is necessary. The multiplication of the mantissas is performed in the same way as in fixed-point to provide a double-precision product.

The double-precision answer is used in fixed-point numbers to increase the accuracy of the product. In floating-point, the range of a single-precision mantissa combined with the exponent is usually accurate enough so that only single-precision numbers are maintained. Thus the half most significant bits of the mantissa product and the exponent will be taken together to form a single-precision floating-point product. The multiplication algorithm can be subdivided into four parts:

1. Check for zeros.
2. Add the exponents.
3. Multiply the mantissas.
4. Normalize the product.

Steps 2 and 3 can be done simultaneously if separate adders are available for the mantissas and exponents. The flowchart for floating-point multiplication is shown in Fig. 9-3. The two operands are checked to determine if they contain a zero. If either operand is equal to zero, the product in the AC is set to zero and the operation is terminated.

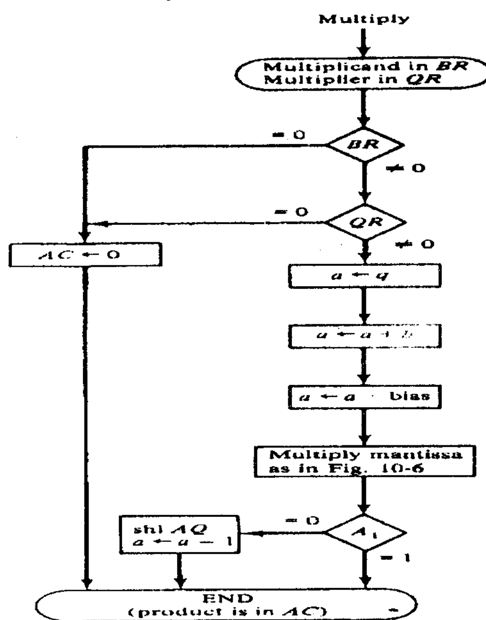



**Fig. 9-3: Multiplication Of Floating-Point Numbers**

If neither of the operands is equal to zero, the process continues with the exponent addition. The exponent of the multiplier is in *q* and the adder is between exponents *a* and *b*. It is necessary to transfer the exponents from *q* to a, add the two exponents, and transfer the sum into *a*. Since both exponents are biased by the addition of a constant, the exponent sum will have double this bias.

The correct biased exponent for the product is obtained by subtracting the bias number from the sum.

The multiplication of the mantissas is done as in the fixed-point case with the product residing in A and *Q*. Overflow cannot occur during multiplication, so there is no need to check for it.

The product may have an underflow, so the most significant bit in A is checked. If it is a 1, the product is already normalized. If it is a 0, the mantissa in AQ is shifted left and the exponent decremented.

Although the low-order half of the mantissa is in Q, we do not use it for the floating-point product. Only the value in the AC is taken as the product.

### 9.2.4 Division

Floating-point division requires that the exponents be subtracted and the mantissas divided. The mantissa division is done as in fixed-point except that the dividend has a single-precision mantissa that is placed in the AC. Remember that the mantissa dividend is a fraction and not an integer.

For integer representation, a single-precision dividend must be placed in register Q and register A must be cleared. The zeros in A are to the left of the binary point and have no significance. In fraction representation, a single-precision dividend is placed in register A and register Q is cleared. The zeros in Q are to the right of the binary point and have no significance. The check for divide-overflow is the same as in fixed-point representation.

However, with floating-point numbers the divide-overflow imposes no problems. If the dividend is greater than or equal to the divisor, the dividend fraction is shifted to the right and its exponent incremented by 1.

For normalized operands this is a sufficient operation to ensure that no mantissa divide-overflow will occur. The operation above is referred to as a *dividend alignment.*

The division of two normalized floating-point numbers will always result in a normalized quotient provided that a dividend alignment is carried out before the division. Therefore, unlike the other operations, the quotient obtained after the division does not require normalization.

The division algorithm can be subdivided into five parts:

1. Check for zeros.
2. Initialize registers and evaluate the sign.
3. Align the dividend.
4. Subtract the exponents.
5. Divide the mantissas.

The flowchart for floating-point division is shown in Fig. 9-4. The two operands are checked for zero. If the divisor is zero, it indicates an attempt to divide by zero, which is an illegal operation. The operation is terminated with an error, message. An alternative procedure would be to set the quotient in *QR* to the most positive number possible (if the dividend is positive) or to the most negative possible (if the dividend is negative). If the dividend in AC is zero, the quotient in *QR* is made zero and the operation terminates.

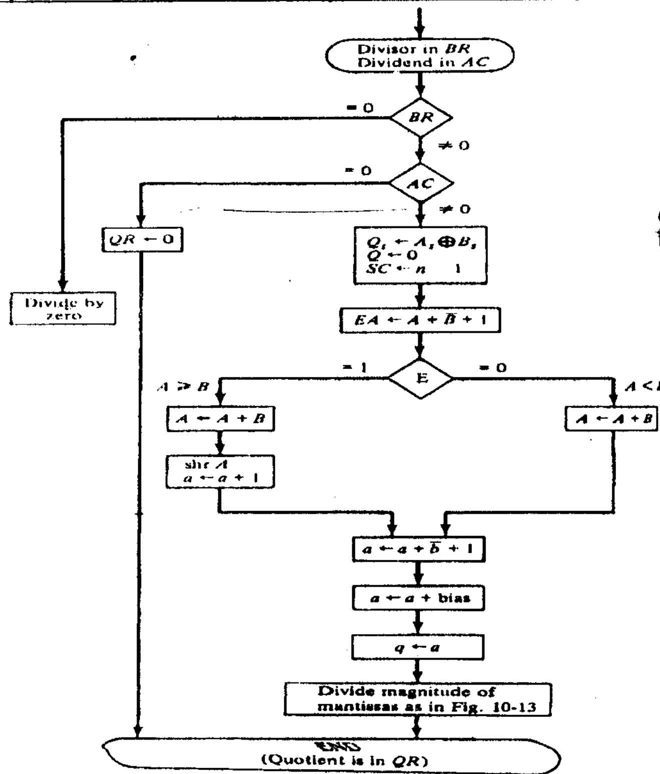



**Fig. 9-4: Division Of Floating-Point Numbers**

If the operands are not zero, we proceed to determine the sign of the quotient and store it in $Q_s$. The sign of the dividend in $A_s$, is left unchanged to be the sign of the remainder.

The Q register is cleared and the sequence counter SC is set to a number equal to the number of bits in the quotient.

The dividend alignment is similar to the divide-overflow check in the fixed-point operation. The proper alignment requires that the fraction dividend be smaller than the divisor.

The two fractions are compared by a subtraction test. The carry in E determines their relative magnitude. The dividend fraction is restored to its original value by adding the divisor.

If $A \geq B$, it is necessary to shift A once to the right and increment the dividend exponent. Since both operands are normalized, this alignment ensures that $A<B$. Next, the divisor exponent is subtracted from the dividend exponent.

Since both exponents were originally biased, the subtraction operation gives the difference without the bias. The bias is then added and the result transferred into $q$ because the quotient is formed in QR.

The magnitudes of the mantissas are divided as in the fixed-point case. After the operation, the mantissa quotient resides in Q and the remainder in A. The floating-point quotient is already normalized and resides in *QR*.

The exponent of the remainder should be the same as the exponent of the dividend. The binary point for the remainder mantissa lies, (n-1) positions to the left of $A_1$. The remainder can be converted to a normalized fraction by subtracting n-1 from the dividend exponent and by shift and decrement until the bit in $A_1$ is equal to 1.

## 9.3 Decimal Arithmetic Unit

The user of a computer prepares data with decimal numbers and receives results in decimal form. A CPU with an arithmetic logic unit can perform arithmetic micro-operations with binary data. To perform arithmetic operations with decimal data, it is necessary to convert the input decimal numbers to binary, to perform all calculations with binary numbers, and to convert the results into decimal.

This may be an efficient method in applications requiring a large number of calculations and a relatively smaller amount of input and output data. When the application calls for a large amount of input-output and a relatively smaller number of arithmetic calculations, it becomes convenient to do the internal arithmetic directly with the decimal numbers.

Computers capable of performing decimal arithmetic must store the decimal data in binary-coded form. The decimal numbers are then applied to a decimal arithmetic unit capable of executing decimal arithmetic micro-operations. A decimal arithmetic unit is a digital function that performs decimal micro-operations.

It can add or subtract decimal numbers, usually by forming the 9's or 10's complement of the subtrahend. The unit accepts coded decimal numbers and generates results in the same adopted binary code.

A single-stage decimal arithmetic unit consists of nine binary input variables and five binary output variables, since a minimum of four bits is required to represent each coded decimal digit. Each stage must have four inputs for the augend digit, four inputs for the addend digit, and an input-carry.

### 9.3.1 BCD Addition

Consider the arithmetic addition of two decimal digits in BCD, together with a possible carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than 9 + 9 + 1 = 19, the 1 in the sum being an input-carry.

Suppose that we apply two BCD digits to a 4-bit binary adder. The adder will form the sum in *binary* and produce a result that may range from 0 to 19.

These binary numbers are listed in Table 9-1 and are labeled by symbols $K$, $Z_8$, $Z_4$, $Z_2$, and $Z_1$. $K$ is the carry and the subscripts under the letter Z represent the weights 8, 4, 2, and 1 that can be assigned to the four bits in the BCD code. The first column in the table lists the binary sums as they appear in the outputs of a 4-bit *binary* adder.

The output, sum of two *decimal* numbers must be represented in BCD and should appear in the form listed in the second column of the table. The problem is to find a simple rule by which the binary number in the first column can be converted to the correct BCD digit representation of the number in the second column.

Table 9-1 Derivation Of BCD Adder

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

In examining the contents of the table, it is apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical and therefore no conversion is needed. When the binary sum is greater than 1001, we obtain a nonvalid BCD representation. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output-carry as required. One method of adding decimal numbers in BCD would be to employ one 4-bit binary adder and perform the arithmetic operation one digit at a time. The low-order pair of BCD digits is first added to produce a binary sum.

If the result is equal or greater than 1010, it is corrected by adding 0110 to the binary sum. This second operation will automatically produce an output-carry for the next pair of significant digits.

The next higher-order pair of digits, together with the input-carry, is then added to produce their binary sum. If this result is equal to or greater than 1010, it is corrected by adding 0110. The procedure is repeated until all decimal digits are added. The logic circuit that detects the necessary correction can be derived from the table entries.

It is obvious that a correction is needed when the binary sum has an output carry $K = 1$. The other six combinations from 1010 to 1111 that need a correction have a 1 in position $Z_8$.To distinguish them from binary 1000 and 1001 which also have a 1 in position $Z_8$, we specify further that either $Z_4$ or $z_2$ must have a 1. The condition for a correction and an output-carry can be expressed by the Boolean function

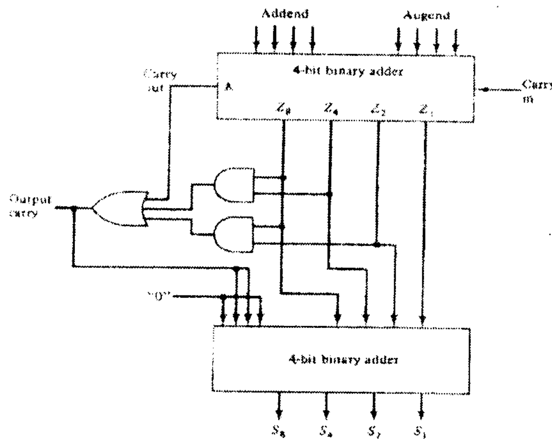
$$C=K+Z_8Z_4+Z_8Z_2$$

When C = 1, it is necessary to add 0110 to the binary sum and provide an output-carry for the next stage.

A BCD adder is a circuit that adds two BCD digits in parallel and produces a sum digit also in BCD. A BCD adder must include the correction logic in its internal construction. To add 0110 to the binary sum, we use a second 4-bit binary adder as shown in Fig. 9-5. The two decimal digits, together with the input-carry, are first added in the top 4-bit binary adder to produce the binary sum. When the output carry is equal to 0, nothing is added to the binary sum. When it is equal to 1, binary 0110 is added to the binary sum through the bottom 4-bit binary adder.

The output-carry generated from the bottom binary adder may be ignored, since it supplies information already available in the output-carry terminal.

**Fig 9-5 Block Diagram Of BCD Adder**




### 9.3.2 BCD Subtraction

A straight subtraction of two decimal numbers will require a subtractor circuit that will be somewhat different from a BCD adder. It is more economical to perform the subtraction by taking the 9's or 10's complement of the subtrahend and adding it to the minuend.

Since the BCD is not self-complementing code, the 9's compliment cannot be obtained by complementing each bit in the code, it must be formed by a circuit that subtracts each BCD digit from 9.

The 9's complement of a decimal digit represented in BCD may be obtained by complementing of the bits in the coded representation of the digit provided a correction included. These are two possible correction methods. In the first method, the binary 1010 (decimal 10) is added to each complemented digit and the carry discarded after each addition.

In the second method the binary 0110(decimal 6) is added before the digit is complemented. As a numerical illustration, the 9's complement of BCD 0111(decimal 7) is computed by first complementing each bit to obtain 1000. Adding binary 1010 and discarding the carry we obtain 0010(decimal 2).

By the second method, we add 0110 to 0111 to obtain 1101. Complementing each bit we obtain the required result of 0010. Complementing each bit of a 4 bit binary number N is identical to subtraction of a number from 1111(decimal 15). Adding the binary equivalent of decimal 10 gives 15-N+10=9-N+16. But 16, signifies the carry that is discarded, so the result is 9-N as required.

Adding the binary equivalent of decimal 6 and then complementing gives, 15-(N+6)=9-N as required. The 9's complement of a BCD digit can also be obtained through a combinational circuit. When the circuit is attached to a BCD adder, the result is a BCD adder/subtractor.

Let the subtrahend (or addend) digit be denoted by four variables $B_8$, $B_4$, $B_2$ and $B_1$. Let M be a mode bit that controls the add/subtract operation. When M=0, the two digits are added; when M=1, the digits are subtracted.

Let the binary variable $x_8$, x4, x2 and $x_1$ be the outputs of the 9's complement circuit. By an examination of the truth table for the circuit it may be observed that $B_1$ should be always be complemented; $B_2$ is always the same in the 9's complement as in original digit; $x_4$ is 1 when the exclusive –OR of $B_2$ and $B_4$ is 1; and $x_8$ is 1 when $B_8$ $B_4$ $B_2$ =000. The Boolean functions for 9's complementer circuit are

$$x_1 = B_1 M' + B_1' M$$

$$x_2 = B_2$$

$$x_4 = B_4 M' + (B_4' B_2 + B_4 B_2')M$$

$$x_8 = B_8 M' + B_8' B_4' B_2' M$$

From these equations we see that x=B when M=0. When M=1, the x outputs produce the 9's complement of B.

One stage of a decimal arithmetic unit that can add or subtract two BCD digits is shown in fig 9.5. It consists of a BCD adder and a 9's complementer. The mode M controls the operation of the unit. With M=0, the S outputs from the sum of A and B. With M=1 the S outputs from the sum of A plus the 9's complement of B. For numbers with n decimal digits we need n such stages.

The output carry $C_{i+1}$ from one stage must be connected to the input carry $C_i$ of the next higher order stage. The best way to subtract two decimal numbers is to let M=1 and apply a 1 to the input carry $C_1$ of the first stage.

The outputs will form sum of A plus the 10's complement of B, which is equivalent to the subtraction operation if the carry–out the last stage is discarded.

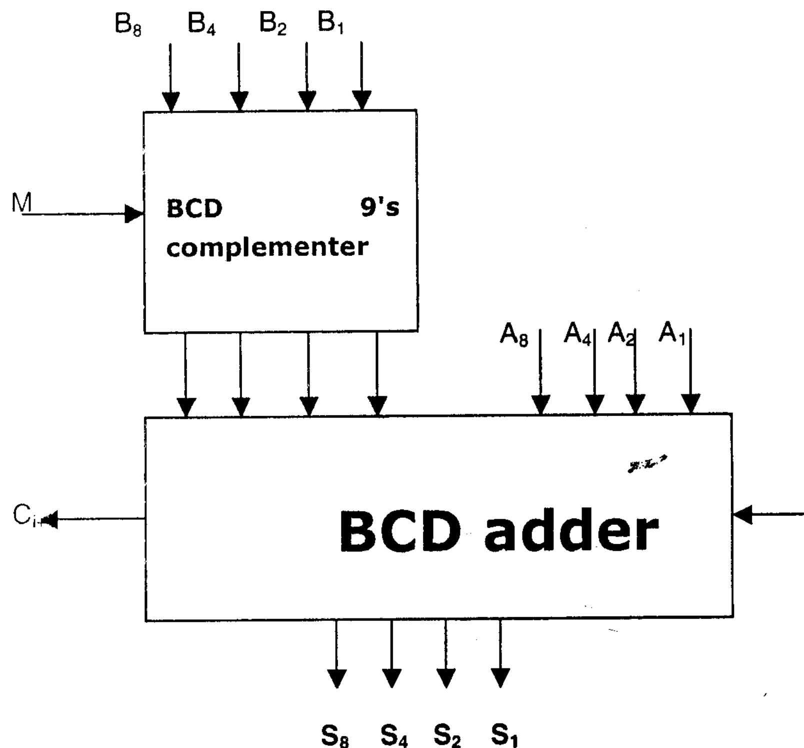



**Fig 9-5: One Stage of a Decimal Arithmetic Unit**

## 9.4 Summary

In this chapter we covered the various arithmetic algorithms and show the procedure for implementing them with digital hardware. We covered addition, subtraction, multiplication and division for the floating-point binary data. We also covered BCD addition and subtraction.

## 9.5 Model Questions

1. Derive an algorithm in flowchart form for addition and subtraction of floating-point numbers.
2. Derive an algorithm in flowchart form for multiplication of floating-point numbers.
3. Derive an algorithm in flowchart form for division of floating-point numbers.
4. Design a 4-bit BCD adder.
5. Show the registers to be used for floating-point arithmetic operations.
6. Explain BCD addition and subtraction in detail.

## 9.6 Reference

1. Computer System Architecture

**By M.MORRIS MANO**

2. Computer Engineering: Hardware Design

**By M.MORRIS MANO**

**Ms. T. ANURADHA, M.Sc., M.S.,**

Lesson 10

# Input – Output Organization

## 10.0 Objective:

In this chapter it discusses the techniques that computers use to communicate with input and output devices. Interface units are presented to show the way that the processor interacts with external peripherals. The procedure for asynchronous transfer of either parallel or serial data is explained. Four modes of transfer are discussed: Programmed I/O, interrupt initiated transfer, direct memory access, and the use of input-output processors.

## Structure of the Lesson


**10.1. Peripheral Devices**

**10.2. Input-Output Interface**

**10.3. Asynchronous Data Transfer**

**10.4.Modes of Transfer**

**10.5.Input-Output Processor (IOP)**

**10.6. Summary**

**10.7. Model Qu-‑stions**

**10.8. References**


## 10.1 PERIPHERAL DEVICES

The input-output (I/O) subsystem of a computer provides an efficient mode of communication between the central system and the outside environment. Programs and data must be entered into computer memory for processing and results obtained from computations must be recorded or displayed for the user. The most familiar input-output peripherals used in Computer system are keyboard, magnetic tape or disks, printer, monitor etc., I

**Peripherals**

Input or output devices attached to the computer.

**Monitor**

Video monitors are the display unit as the output device. There are different types of video monitors, but the most popular use a cathode ray tube **(CRT).** CRT contains an electronic gun that sends an electronic beam to a phosphorescent screen in front of the tube. The beam can be deflected horizontally and vertically. To produce a pattern on the screen, a grid inside the CRT receives a variable voltage that causes the beam to hit the screen and make it glow at the selected spots.

**Keyboard**

Keyboard is used to enter the information into a computer. It allows alphanumeric information directly. Every time a key is depressed, it sends a binary coded character to the computer. The processor will be idle most of the time while waiting for the information to arrive due to slow speed of the keyboard.

**Printer**

Printers provide a permanent record on paper of computer output data or text. There are three basic types of character printers: daisywheel, dot matrix, and laser printers. The daisywheel printer contains a wheel with the characters placed along the circumference. To print a character, the wheel rotates to the proper position and energized magnet then presses the letter against the ribbon. The dot matrix printer contains a set of dots along the printing mechanism. Each dot can be printed or not, depending on the specific characters that are printed on the line. The laser printer uses a rotating photographic drum that is used to imprint the character images. The pattern is then transferred onto paper in the same manner as a copying machine.

**Magnetic tape/disks**

Magnetic tapes are used mostly for storing files of data. The access mechanism is sequential. It is cheapest and slowest methods for storage. The advantage existed in tapes is that can be removed when not in use.

Magnetic disks have high-speed rotational surfaces coated with magnetic material. Access is achieved by moving a read-write mechanism to a track in the magnetized surface. Disks are used mostly for storing huge data.

**ASCII Alphanumeric Characters:**

Input and Output devices can communicate information in between people and the computer is usually in alphanumeric information. The standard binary code for the alphanumeric character is ASCII (**American Standard Code for Information Interchange).** It uses seven bits to code (b1 through b7). The ASCII code contains 94 characters that can be printed (26 uppercase letters A through Z, the lowercase letters, the 10 numerals 0 through 9, and 32 special characters such as %, *, and $) and 34 characters that can be non printed used for various control functions.

34 control characters are designated in the ASCII table with abbreviated names.

| | $b_7 b_6 b_5$ | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $b_4 b_3 b_2 b_1$ | 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 | DE |

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 b |
| 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 |

## 10.2 INPUT-OUTPUT INTERFACE

Input-Output Interface provides a method for transferring information between internal storage and external storage devices. Special communication links are needed to connect with Central Processing Unit. Communication links is to resolve the differences that exist between central computer and peripheral. The differences are

- Peripherals are electromechanical and electromagnetic devices. The mode of operation is different from the operation of CPU. and memory. Therefore a conversion of signal values may be required.

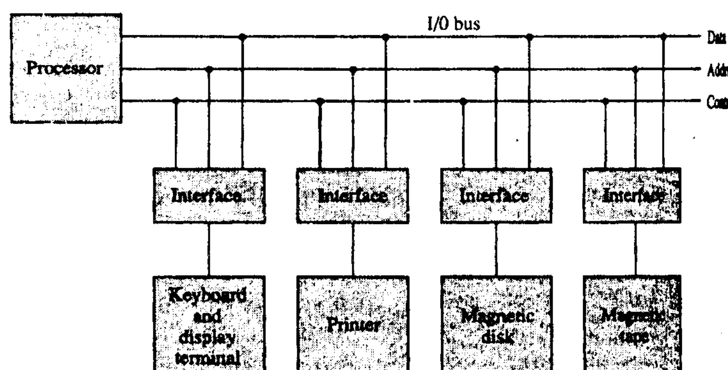
- The data transfer rate of peripherals is usually slow than the transfer rate of CPU. A synchronization mechanism is needed.
- Data codes and formats in peripherals differ from the word format in the CPU and memory.
- The operating modes of peripherals are different from each other and each must be controlled so as not to disturb the operation of other peripherals connected to the CPU.

To overcome the above differences, computer systems are having special hardware components between the CPU and peripherals. The special hardware components are to supervise and synchronize all input and output transfers. The Components are called interface units.

**I/O Bus and Interface Modules:**

The I/O Bus communicates between processor and several peripherals. It consists of data lines, address lines, and control lines. Each peripheral device has associated with it an interface unit. Each interface decodes the address and control received from the I/O bus, interprets them for the peripheral, and provides signals for the peripheral controller. Each I/O bus synchronizes the data flow and supervises the transfer between peripheral and processor. Each peripheral has its own controller that operates the particular electromechanical device. Controller may be separately or may be integrated with the peripheral.

**Connecting 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 lines and the device that it controls. All peripherals whose address does not correspond to the address in the bus are disabled by their interface. If the address is available in the address line, the processor provides a function code in the control lines. The function code is referred to as an I/O command and is in essence an instruction that is executed in the interface and its attached peripheral unit. There are four types of commands that an interface may receive. They are classified as control, status, data output and data input.

Control Command: A control command is issued to activate the peripheral and to inform it what to do.

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

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

Data input: The data input command is the opposite of the data output.

**I/O versus Memory Bus:**

The I/O bus, the memory bus contains data, address, and read/write control lines. There are three ways that computer buses can be used to communicate with memory and I/O.

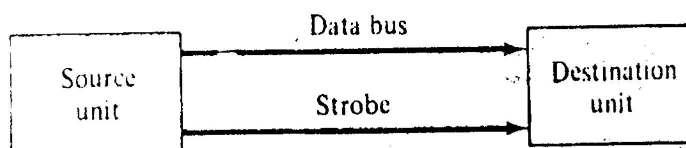
1. Use two separate buses, one for memory and the other for I/O.
2. Use one common bus for both memory and I/O but have separate control lines for each.
3. Use one common bus for memory and I/O with common control lines.

## 10.3 ASYNCHRONOUS DATA TRANSFER

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. In asynchronous data transfer source and destination do not share a common clock. It needs a communication methodology. They are strobe and control signal handshaking.

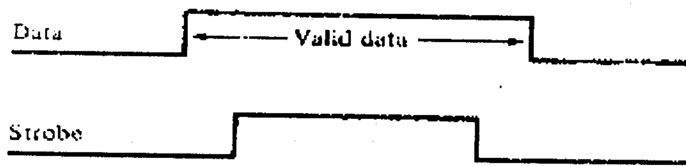
**Strobe Control**

The strobe control method of asynchronous data transfer employs a single control line to time each transfer. The strobe may be activated either the source or destination unit.



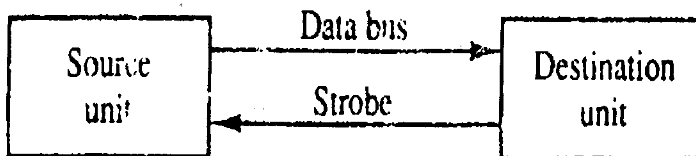

(a) Block diagram

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

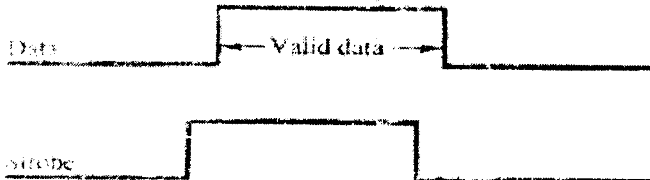



(b) Timing diagram

In the above diagram the source unit first places the data on the data bus. After a brief delay to ensure that the data settle to a steady value, the source activates the strobe pulse. The information on 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. Often, the destination unit uses the falling edge of the strobe pulse to transfer the contents of the data bus into one of its internal registers. The source removes the data from the bus a brief period after it disables its strobe pulse. The source does not have to change the information in the data bus because that the strobe signal is disabled indicates that the data bus does not contain valid data.




(a) Block diagram



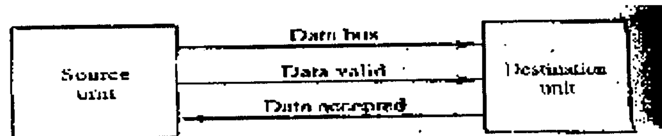

(b) Timing diagram

The above shows a data transfer initiated by the destination unit. In this case 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 data bus. 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 unit removes the data from the bus after a predetermined period.

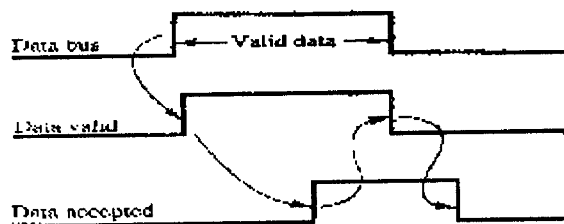
**Handshaking:**

The disadvantage of the strobe method is that the source unit that initiates the transfer has no way of knowing whether the destination unit has actually received the data item that was placed in the bus. Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit has actually placed the data on the bus. The Handshaking method solves the above problems. It introduces a second control signal that provides a reply to the unit that

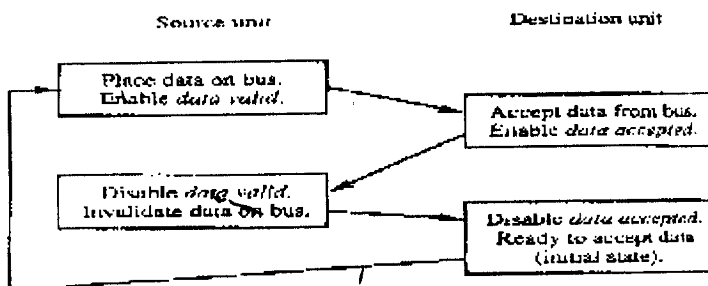
initiates the transfer. One control line is in the same direction as the data flow in the bus from the source to the destination. The basic principle of the two-wire handshaking method of data transfer is as follows. One control line is in the same direction as the data flow in the bus from the source to the destination. It is used by the source unit to inform the destination unit whether there are valid data in the bus uses it. The second control line is in the other direction from the destination to the source. The destination unit to inform the source whether it can accept data uses it. The sequence of control during the transfer depends on the unit that initiates the transfer.




(a) Block diagram




(b) Timing diagram




(c) Sequence of events

The above diagram shows the data transfer procedure initiated by the source. The two-handshaking lines are data valid and data accepted. The source unit generates data valid and the destination unit generates data accepted. The timing diagram shows the exchange of signals between two units. The sequence of events listed in part © shows the four possible states that the system can be at any given time.

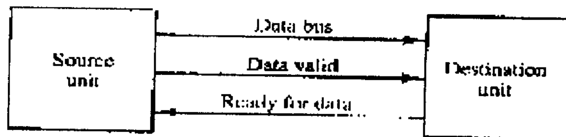
1.The source unit indicates the transfer by placing the data on the bus and enabling its data valid signal.

2. The data accepted the destination unit activates signal after it accepts the data from the bus.

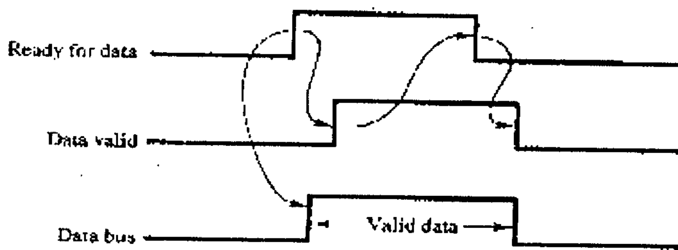
3.The-source unit then disables its data valid signal, which invalidates the data on the bus.

4. The destination unit then disables its data accepted signal and the system goes into its initial state. The source does not send the next data item until after the destination unit shows its readiness to accept new data by disabling its data accepted signal.

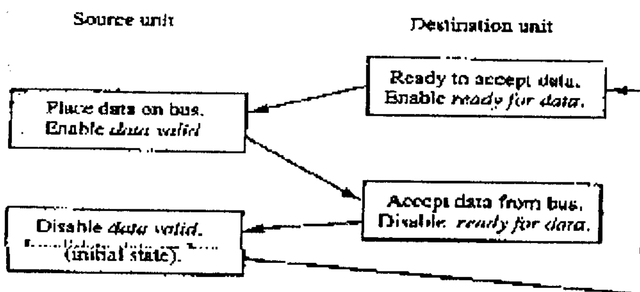
The destination-initiated transfer using handshaking lines is shown in the below diagram.




(a) Block diagram




(b) Timing diagram




The signal generated by the destination unit has been changed to ready for data to reflect its new meaning. The source unit in this case does not place data on the bus until after it receives the ready for data signal from the destination unit. From there on, the handshaking procedure follows the same method as in the source-initiated case. The sequence of events in both cases would be identical if we consider the ready data signal as the complement of data accepted. The only difference between the source-initiated and the destination –initiated transfer is in the choice of initial state.

## 10.4. MODES OF TRANSFER

The Input/Output operations can be performed by three basic techniques. These are

- Programmed Input/Output

- Interrupt driven Input/Output
- Direct Memory Access

Overview of the three Input/Output Techniques

| | Interrupt Required | I/O module to/from Memory transfer |
|---|---|---|
| Programmed I/O | NO | Through CPU |
| Interrupt driven I/O | Yes | Through CPU |
| DMA | Yes | Direct to Memory |

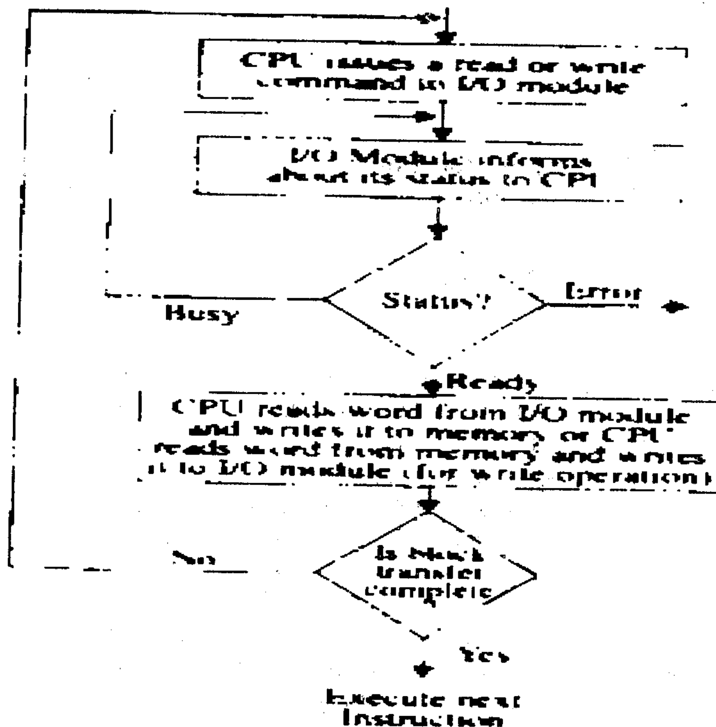
In Programmed I/O, the I/O operations are completely controlled by the CPU. The CPU executes programs that initiate, directs and terminate an I/O operation. It required a little special I/O hardware, but is quite time consuming for the CPU, since CPU has to wait for slower I/O operations to complete. Another technique suggests to reduce the waiting by CPU is interrupt driven I/O. The CPU issues the I/O command to I/O module and starts doing other work, which may be execution of a separate program. When the I/O operation is complete, I/O module interrupts CPU by informing the CPU that I/O has finished. CPU, then, may proceed execution of this program. In both programmed I/O and interrupt driven I/O CPU is responsible for extracting data from the memory for Output and storing data in memory for input. Such a requirement does not exist in DMA where the memory can be accessed directly by I/O module. Thus the I/O module can store or extract data in/from the memory.

**Programmed input/output**

Programmed input/output is useful I/O method for computers where hardware costs need to be minimized. The Input or Output operation in such cases may involve:

- Transfer of data from I/O device to the CPU registers.
- Transfer of data from CPU registers to memory.

In programmed I/O method the responsibility of CPU is to constantly check the status of the I/O device to check whether it has become free or it has finished inputting the current series of data. Thus, Programmed I/O is a very time consuming method where CPU wastes lot of time for checking and verifying the status of an I/O device.




(a) Programmed I/O

I/O Instructions

To carry out input/output CPU issues I/O related instruction. These instructions consist of two components:

- The address of the Input/Output device specifying the I/O device and I/O module and
- An input/output command

There are four types of I/O Commands, which can be classified as:

CONTROL, TEST, READ and WRITE

CONTROL commands are device specific and are used to control the specific instructions to the device e.g., a magnetic tape requires rewinding or moving forward by a block. TEST command checks the status such as, if a device is ready or not or is in error condition. The READ command is used for input of data from input device and WRITE command is used for output of data to output device.

The other part of I/O instruction is the address of the I/O device. In systems with programmed I/O the I/O module, the main memory and the CPU normally share the system bus. Thus, each I/O module should interpret the address lines to determine if the command is for itself. Or in other words: How does CPU specify which device to access? There are two methods of doing so. These are called memory mapped I/O and I/O-mapped I/O.

If we use the single address space for memory locations and I/O devices, i.e., the CPU treats the status and data registers of I/O module as memory locations, and then memory and I/O devices can be accessed using the same instructions. This is referred to as memory mapped I/O. For a memory mapped I/O only a single READ and a single WRITE line are needed for memory or I/O module read or write operations. CPU activates these lines for either memory access or I/O device access.

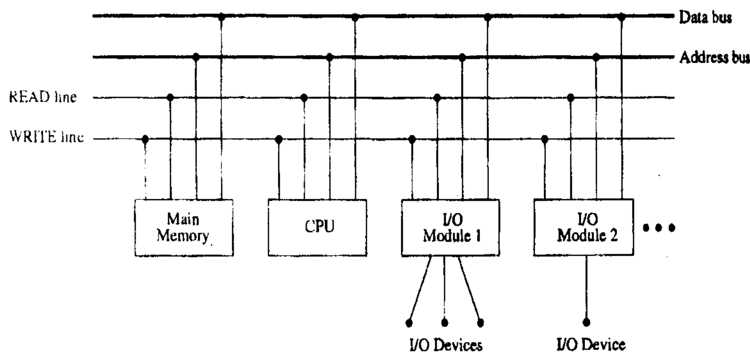



**Figure 4: Structure of Memory Mapped I/O**

In I/O-mapped I/O devices and memory are addressed separately. There are separate control lines for memory and I/O device read or write operations, thus, a memory reference instruction does not affect an I/O device. Here separate Input/ Output instructions are needed which cause data transfer between addressed I/O module and CPU.

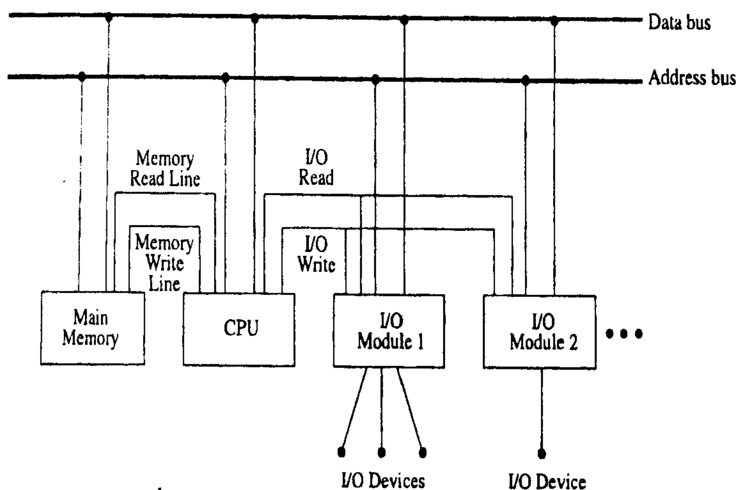


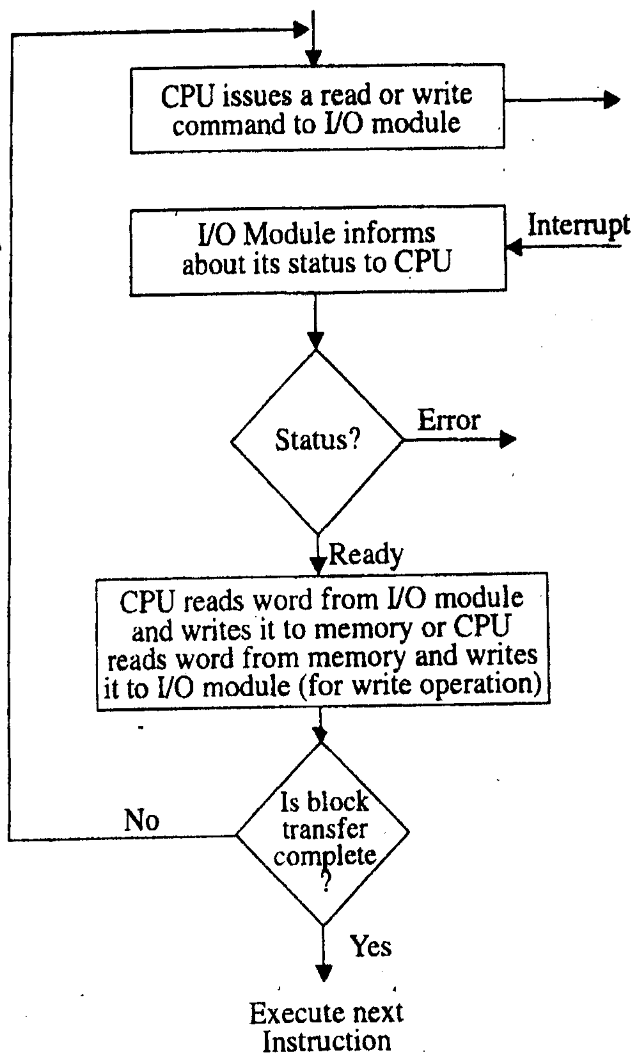

**Figure 5: Structure of I/O-mapped I/O**

In the case of memory mapped I/O the READ instruction may bring data to or from memory or I/O module, while in I/O-mapped I/O we need to have separate instruction for Input / Output.

**Interrupt Driven Input / Output:**

The drawback in programmed I/O is the speed of I/O devices is much slower in comparison to that of CPU and because the CPU has to repeatedly check whether a device is free; or wait till the completion of I/O, therefore, the performance of CPU in programmed I/O goes down tremendously. For solving the above problem a well-designed mechanism was conceived, which is referred to as Interrupt driven I/O. In this mechanism, provisions of interruption of CPU work, once I/O device has finished the I/O or when it is ready for the I/O, has been provided.

The Interrupt driven I/O mechanism for transferring a block of data is shown in




(b) Interrupt driven I/O

After issuing a READ command (for input the CPU goes off to do other useful work (it may be execution of a different program) while I/O module proceeds for reading of data from associated device At the completion of an instruction cycle the CPU checks for interrupts (which will occur when data is in data register of I/O module and it now needs CPU's attention. An interrupt loosely is used for any exceptional event that causes temporary transfer of control of CPU from one program to the other which is causing the interrupt. Interrupts are primarily issued on:

- Initiation of input/output operation
- Completion of an Input / Output operation
- Occurrence of hardware or software errors.

Interrupts can be generated by various sources internal external to the CPU. An interrupt generated internally by CPU is sometimes termed as Traps. The traps are normally results of programming errors such as division by zero while execution of a program. The two key issues in Interrupt driven input/output are:

- To determine the device which has issued an interrupt
- In case of occurrence of multiple interrupts which one to be processed first

There are several solutions to these problems. The simplest of them is to provide multiple interrupt lines, which will result in immediate recognition of the interrupting device. The priorities can be assigned to various interrupts and the interrupt with highest priority should be selected for service in case multiple interrupt occurs. But providing multiple interrupt lines is an impractical approach because only a few lines of the system bus can be devoted for the interrupt. Other methods are software poll, daisy chaining etc.,

**Software Poll:**
In this scheme on occurrence of an interrupt CPU starts executing a software routine termed as interrupt service program or routine which poll to each I/O module to determine which I/O module has caused the interrupt. This may be achieved by reading the status register of the I/O modules. The priority here can be implemented easily by defining the polling sequence, since the device polled first will have higher priority. Please note that after identifying the device the next set of instructions to be executed will be the device service routines of that device, resulting in the desired input or output.

## Direct Memory Access (DMA)

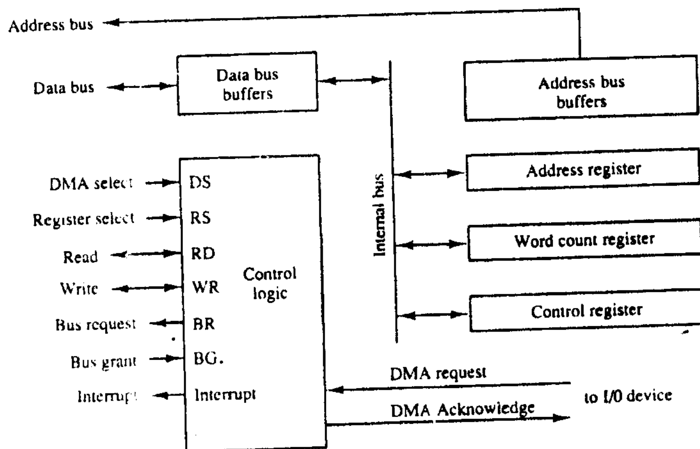
The transfer of data between a fast storage device such as magnetic disk and memory is often limited by the speed of the CPU. Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of the transfer. This transfer technique is called Direct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

The CPU may be placed in an idle state in a variety of ways. One method extensively used in microprocessors is to disable the buses through special control signals. In the below diagram shows two control signals in the CPU that facilitate the DMA transfer. The bus request (BR)

input is used by the DMA controller to request the CPU to relinquish control of the buses. When this input is active, the CPU terminates the execution of the current instruction and places the address bus, the data bus, and the read and write lines into high-impedance state. The high-impedance state behaves like an open circuit, which means that the output is disconnected and does not have logic significance. The CPU activates the bus grant (BG) output to inform the external DMA that the buses are in the high-impedance state. The DMA that originated the bus request can now take control of the buses to conduct memory transfers without processor intervention. When the DMA terminates the transfer, it disables the bus request line. When DMA takes control of the system, it communicates directly with the memory. The transfer can be made in several ways. In DMA burst transfer, a block sequence consisting of a number of memory words is transferred in a continuous burst while the DMA controller is master of the memory buses. This mode of transfer is needed for fast devices such as magnetic disks where data transmission cannot be stopped or slowed down until an entire block is transferred. An alternative technique called cycle stealing allows the DMA controller to transfer one data word at a time, after which it must return control of the buses to the CPU. The CPU merely delays its operation for one memory cycle to allow the direct memory I/O transfer to steal one memory cycle.

**DMA C. ntroller**

The DM controller needs the circuits of an interface to communicate with the CPU and I/O device. addition, it needs an address register, a word count register, and a set of address lines. The address register and address lines are used for direct communication with the memory. The word count specifies the number of words that must be transferred. The data transfer may be done directly between the device and memory under control of the DMA.




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

register, a word count register, and a control register. The address register contains an address to specify the desired location in memory. The address register contains an address to specify the desired location in memory. The address bits go through bus buffers into the address bus. The address register is incremented after each word that is transferred to memory. The word count register holds the number of word to be transferred. This register is decremented by one after each word transfer and internally tested for zero. The control register specifies the mode of transfer. All registers in the DMA appear to the CPU as I/O interface registers. Thus the CPU can read from or write into the DMA registers under program control via the data bus. The CPU first initializes the DMA. After that, the DMA starts and continues to transfer data between memory and peripheral unit until an entire block is transmitted. The initialization process is essentially a program consisting of I/O instructions that include the address for selecting particular DMA registers. The CPU initializes the DMA by sending the following information through the data bus.

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

The starting address is stored in the address register. The word count is stored in the word count register, and the control information in the control register. Once the DMA is initialized, the CPU stops communicating with the DMA unless it receives an interrupt signal or if it wants to check how many words have been transferred.

**DMA Transfer**

The CPU Communicates with the DMA through the address and data buses as with any interface unit. The DMA has its own address, which activates the DS and RS lines. The C PU initializes the DMA through the data bus. Once the DMA receives the start control command, it can start the transfer between the peripheral device and the memory. When the peripheral device sends a DMA request, the DMA controller activates the BR line, informing the DMA that its buses are disabled. The CPU responds with its BG line, informing the DMA that its buses are disabled. The DMA then puts the current value of its address register into the address bus, initiates the RD or WR signal, and sends a DMA acknowledge to the peripheral device. The RD and WR lines in the DMA controller are bi-directional. The direction of transfer depends on the status of the BG line. When BG=0, the RD and WR are input lines allowing the CPU to communicate with the internal DMA registers. When BG=1, the RD and WR are output lines from the DMA controller to the random-access memory to specify the read or write operation for the data.

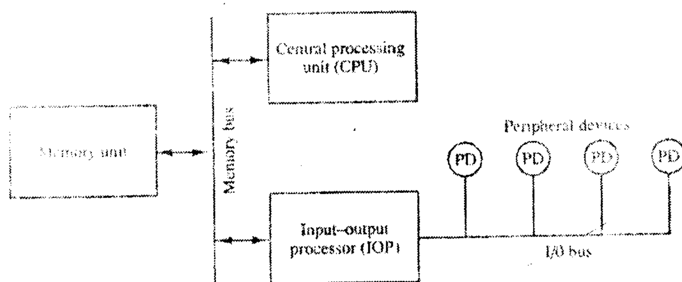
When the peripheral device receives a DMA acknowledge, it puts a word in the data bus (for write) or receives a word from the data bus (for read). Thus the DMA controls the read or write operations and supplies the address for the memory. The peripheral unit can then communicate with memory through the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

momentarily disabled. For each word that is transferred, the DMA increments its address registers and decrements its word count registers. 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 disables 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 registers 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. The CPU can read this register at any time to check the number of words already transferred. A DMA controller may have more than one channel. In this case, each channel has a request and acknowledges pair of control signals, which are connected to separate peripheral devices. Each channel also has its own address register and word count register within the DMA controller. A priority among the channels may be established so that channels with high priority are serviced before channels with lower priority. DMA transfer is very useful in many applications. It is used for fast transfer of information between magnetic disks and memory. It is also useful for updating the display in an interactive terminal. Typically, an image of the screen display of the terminal is kept in memory, which can be updated under program control. The contents of the memory can be transferred to the screen periodically by means of the DMA transfer.

## 10.5. INPUT-OUTPUT PROCESSOR (IOP)

An input-output processor (IOP) may be classified as a processor with direct memory access capability that communicates with I/O devices. The IOP is similar to a CPU except that it is designed to handle the details of I/O processing. Unlike the DMA controller that must be set up entirely by the CPU, the IOP can fetch and execute its own instructions. IOP instructions are specifically designed to facilitate I/O transfers. In addition, IOP can perform other processing tasks, such as arithmetic, logic, branching, and code translation. The block diagram of a computer with two processors is shown in figure:

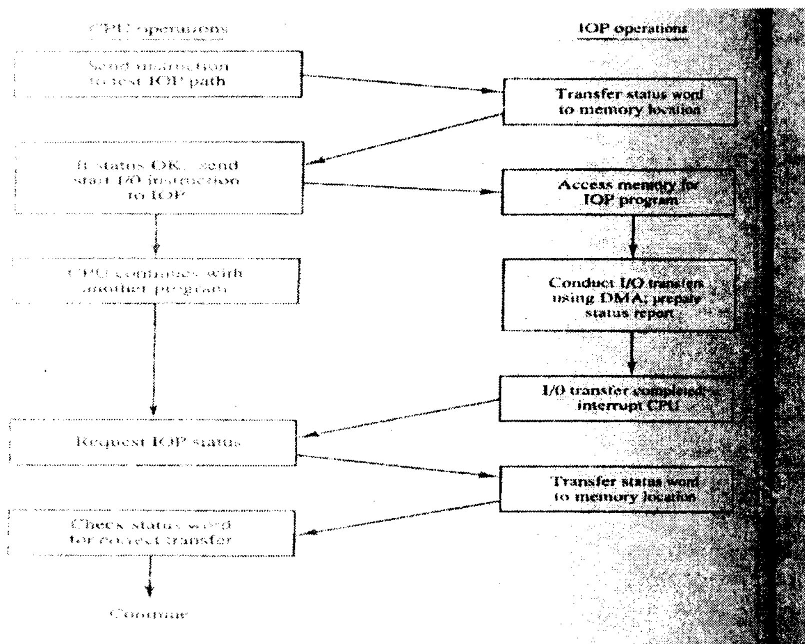



The memory unit occupies a central position and can communicate with each processor by means of direct memory access. The CPU is responsible for processing data needed in the solution of computational tasks. The IOP provides a path for transfer of data between various peripheral devices and the memory unit. The CPU is usually assigned the task of initiating the I/O program. From then on the IOP operates independent of the CPU and continues to transfer data from external devices and memory.

The data formats of peripheral devices differ from memory and CPU data formats. The IOP must structure data words from many different sources. For example, it may be necessary to take four bytes from an input device and pack them into one 32-bit word before the transfer to memory. Data are gathered in the IOP at the device rate and bit capacity while the CPU is executing its own program. After the input data are assembled into a memory word, they are transferred from IOP directly into memory by "stealing" one memory cycle from the CPU. Similarly, an output word transferred from memory to the IOP is directed from the IOP to the output device at the device rate and bit capacity. The communication between the IOP and the devices attached to it is similar to the program control method of transfer. Communication with the memory is similar to the direct memory access method. The way by which the CPU and IOP communicate depends on the level of sophistication included in the system. In very large-scale computers, each processor is independent of all others and any one processor can initiate an operation. In most computer systems, the CPU is the master while the IOP is a slave processor. The CPU is assigned the task of initiating all operations to start an I/O instructions are executed in the IOP. CPU instructions provide operations to start an I/O transfer and also to test I/O status conditions needed for making decisions on various I/O activities. The IOP, in turn, typically asks for CPU attention by means of an interrupt. It also responds to CPU requests by placing a status word in a prescribed location in memory to be examined later by a CPU program. An I/O operations is desired, the CPU informs the IOP where to find the I/O program and then leaves the transfer details to the IOP.

**CPU - IOP Communication**

The communication between CPU and IOP may take different forms, depending on the particular computer considered. The memory unit acts as a message center where each processor leaves information for the other. The sequence of operations may be carried out as shown in the flowchart.



The CPU sends an instruction to test the IOP path. The IOP responds by inserting a status word in memory for the CPU to check. The bits of the status word indicate the condition of the IOP and I/O device, such as IOP overload condition, device busy with another transfer, or device ready for I/O transfer. The CPU refers to the status word in memory to decide what to do next. If all is in order, the CPU sends the instruction to start I/O transfer. The memory address received with this instruction tells the IOP where to find its program. The CPU can now continue with another program while the IOP is busy with the I/O program. Both programs refer to memory by means of DMA transfer. When the IOP terminates the execution of its program, it sends an interrupt request to the CPU. The CPU responds to the interrupt by issuing an instruction to read the status from the IOP. The IOP responds by placing the contents of its status report into a specified memory location. The status word indicates whether the transfer has been completed or if any errors occurred during the transfer. From inspection of the bits in the status word, the CPU determines if the I/O operation was completed satisfactorily without errors.

The IOP takes care of all data transfers between several I/O units and the memory while the CPU is processing another program. The IOP and CPU are competing for the use of memory, so the number of devices that can be in operation is limited by the access time of the memory. It is not possible to saturate the memory by I/O devices in most systems, as the speed of most devices is much slower than the CPU. However, some very fast units, such as magnetic disks, can use an appreciable number of available memory cycles. In that case, the speed of the CPU may deteriorate because it will often have to wait for the IOP to conduct memory transfers.

## 10.6. SUMMARY

In this lesson you learnt the concept of peripheral devices. In this chapter we introduced the concept of Input–Output Interface and I/O versus Memory Bus. In this lesson we introduced the concept of Asynchronous Data Transfer. In Asynchronous Data Transfer we introduced the concept of strobe control, Handshaking mechanism. In this lesson we learned the concept of Modes of Transfer.

## 10.7. MODEL QUESTIONS

1. List four peripheral Devices and explain them?
2. Explain the concept of Handshaking?
3. What is DMA? Explain Block Diagram of DMA Controller?
4. Explain Programmed Input/ Output?
5. Explain in detail about CPU-IOP Communication?

## 10.8. REFERENCES

1. Hays, J.F., Computer Architecture and Organization, 2nd edition, New York: McGraw-Hill.
2. M. Morris Mano., Computer System Architecture – Eastern Economy Edition

**Sri. C.V.P.R.PRASAD,** M.C.A.,

Lesson 11

# Memory Organization

## 11.0 Objective:

In this unit we will discuss about the main memory, cache memory, magnetic secondary memory, optical memory. At the end of the unit, you will be able to describes memory hierarchy, distinguish among various types of memories, describes the importance of cache and virtual memories.

## Structure of the Lesson:


**11.1. Memory Hierarchy**

**11.2. Main Memory**

**11.3. Auxiliary Memory**

**11.4. Associative Memory**

**11.5. Cache Memory**

**11.6. Virtual Memory**

**11.7. Summary**

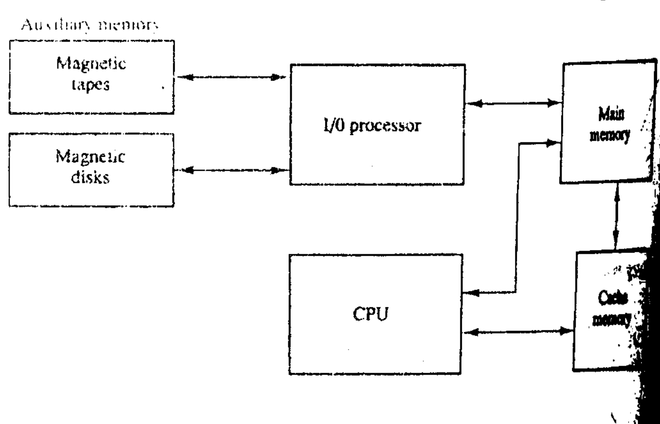
**11.8. Model Questions**

**11.9. References**


## 11.1. MEMORY HIERARCHY

The memory unit is an essential component in any digital computer because it is needed for storing programs and data. A very small computer with a limited application may be able to fulfill its intended task without the need of additional storage capacity. The memory unit that communicates directly with the CPU is called the main memory. Devices that provide backup storage are called auxiliary memory. The most common auxiliary memory devices used in computer systems are magnetic disks and tapes. The total memory capacity of a computer can be visualized as being a hierarchy of components. 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.

**Figure 11.1. Memory Hierarchy in a computer system**




The above diagram illustrates the components in a typical memory hierarchy.

At the bottom of the hierarchy are the relatively slow magnetic tapes used to store removable files. Next are the magnetic disks used as backup storage. The main memory occupies a central position by being able to communicate directly with the CPU and with auxiliary memory devices through an I/O Processor. When the CPU needs programs not residing in main memory, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space for currently used programs and data. A special very-high-speed memory called a cache is sometimes used to increase the speed of processing by making current programs and data available to the CPU at a rapid rate. The cache memory is employed in computer systems to compensate for the speed differential between main memory access time and processor logic. 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 programs and data available at a rapid rate, it is possible to increase the performance rate of the computer.

## 11.2. MAIN MEMORY

The main memory is the central storage unit in a computer system. It is a relatively large and fast memory used to store programs and data during the computer operation. The principal technology used for the main memory is based on semiconductor integrated circuits. Integrated circuit RAM chips are available in two possible operating modes, **static** and **dynamic**.

The static RAM consists essentially of internal flip-flops that store the binary information. The stored information remains valid as long as power is applied to the unit. The dynamic RAM stores the binary information in the form of electric charges that are applied to capacitors. The capacitors are provided inside the chip by MOS transistors. The stored charge on the capacitors tends to discharge with time and the capacitors must be periodically recharged by refreshing the dynamic memory. Refreshing is done by cycling through the words every few milliseconds to restore the decaying charge. The dynamic RAM offers reduced power consumption and larger storage capacity in a single memory chip. The static RAM is easier to use and has shorter read and write cycles. Originally RAM was used to refer to a random-access memory, but now it is used to designate a read/write memory. ROM means read only memory is used for storing

programs that are permanently resident in the computer and for tables of constants that do not change in value once the production of the computer is completed. The ROM portion of main memory is needed for storing an initial program called a **bootstrap loader**.

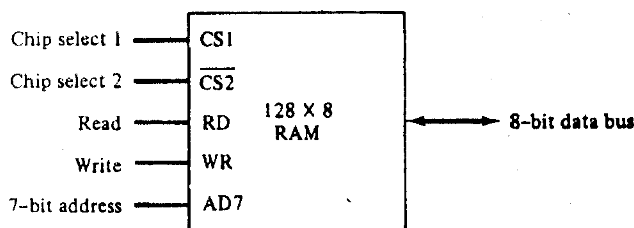
The **bootstrap loader** is a program whose function is to start the computer software operating when power is turned on. Since RAM is volatile, its contents are destroyed when power is turned off. The contents of ROM remain unchanged after power is turned off and on again. The startup of a computer consists of turning the power on and starting the execution of an initial program. When power is turned on, the hardware of the computer sets the program counter to the first address of the bootstrap loader. The bootstrap program loads a portion of the operating system from disk to main memory and control is then transferred to the operating system, which prepares the computer for general use. RAM and ROM chips are available in a variety of sizes. If the memory needed for the computer is larger than the capacity of one chip, it is necessary to combine a number of chips to form the required memory size.

**RAM and ROM chips**

A RAM chip is better suited for communication with the CPU if it has one or more control inputs that selected the chip only when needed. Another common feature is a bi-directional 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 bi-directional bus can be constructed with three-state buffers. A three-state buffer output could be placed in one of three possible states: a signal equivalent to logic 1, a signal equivalent to logic 0, or a high-impedance state. The logic 1 and 0 are normal digital signals. The high-impedance state behaves like an open circuit, which means that the output does not carry a signal and has no logic significance.

The block diagram of a RAM chip is as follows:

**Figure 11.2 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

**Function Table**

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 bi-directional data bus. The read and write inputs specify the memory operation and the two chips select (CS) control inputs are for enabling the chip only when it is selected by the microprocessor. The availability of more than one control input to select the chip facilities the decoding of the address lines when multiple chips are used in the microcomputer. The read and write inputs are sometimes combined into one line labeled R/W. When chip is selected, the two binary states in this line specify the two operations of read or write. The function table listed in the above diagram specifies the operation of the RAM chip. The unit is in operation only

When CS1=1 and $\overline{CS2}$ = 0. The bar on top of the second select variable indicates that this input is enabled when it is equal to 0. If the chip select inputs are not enabled, or if they are enabled but the read or write inputs are not enabled, the memory is inhibited and its data bus is in a high-impedance state. When CS1=1 and CS2 = 0, the memory can be placed in a write or read mode. When the WR input is enabled, the memory stores a byte from the data bus into a location specified by the address input lines. When the RD input is enabled, the content of the selected byte is placed into the data bus. The RD and WR signals control the memory operation as well as the bus buffers associated with the bi-directional data bus.

A ROM chip is organized externally in a similar manner, However, since a ROM can only read, the data bus can only be in an output mode. The block diagram of a ROM chip is shown in fig, 11.2. for the same-size chip, it is possible to have more bits of ROM than RAM, because the internal binary cells in Rom occupy less space than in RAM. For this reason, the diagram specifies a 512-byte ROM, while the RAM has only 128 bytes.

The nine address lines in the ROM chip specify of any one of the 512 bytes stored in it. The two chip select inputs must be CS1= 1 and cs2=0 for the unit to operate. Otherwise, the data bus is in a high-impedance state. There is no need for a read or write control because the unit can only read. Thus when the chip is enabled by the two inputs, the byte selected by the address lines appears on the data bus.

**Memory Address Map**

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

To demonstrate with a particular example, assume that a computer system needs 512 bytes of RAM and 512 bytes of ROM. The RAM and ROM chips to be used are specified in Fig. 11.2 and 11.3.

Figure 11.3

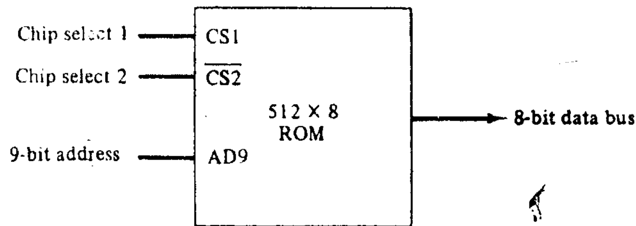



**Table 11.1 Memory Address Map for Microprocessor**

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

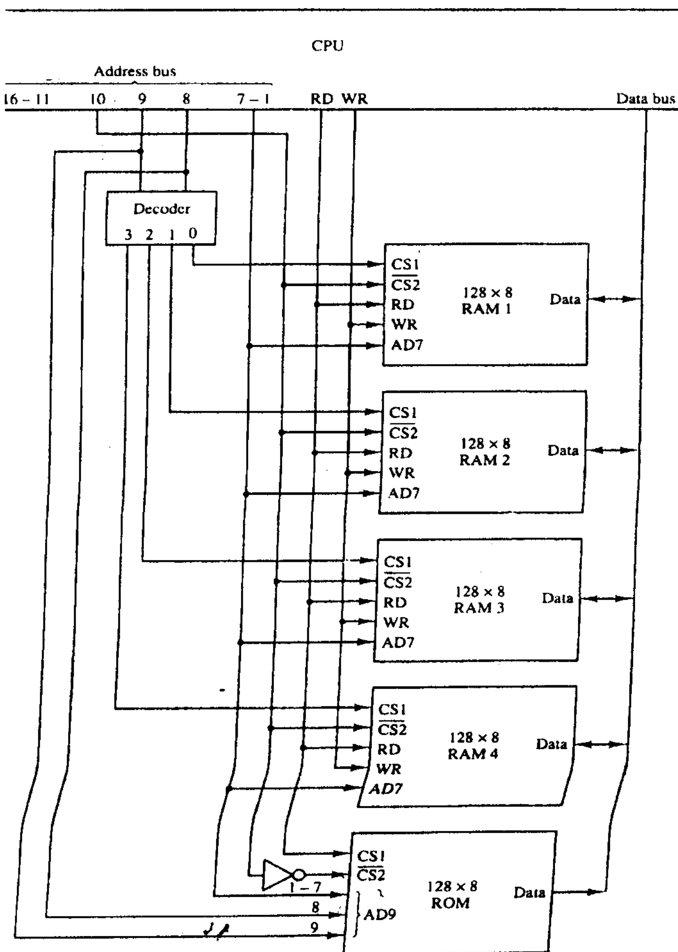
The memory address map for this configuration is shown in Table 11.1. The component column specifies whether a RAM or a ROM chip is used. The hexadecimal address column assigns a range of hexadecimal equivalent addresses for each chip. The address bus lines are listed in the third column. Although there are 16 lines on the address bus, the table shows only 10 lines because the other 6 are not used in this example and are assumed to be zero. The small x's under the address bus lines designate those lines that must be connected to the address inputs in each chip. The RAM chips have 128 bytes and need seven addresses lines. The ROM chip has 512 bytes and needs 9 address lines. The x's are always assigned to the low-order bus lines: lines 1 though 7 for the RAM and lines 1 through 9 for the ROM. It is then necessary to distinguish between four RAM chips by assigning to each a different address. For this particular example we choose bus lines 8 and 9 to represent four distinct binary combinations. Note that any other pair of unused bus lines can be chosen for this purpose. The table clearly shows that the nine low-order bus lines constitute a memory space for RAM equal to 29-512 bytes. The distinction between a RAM and ROM address is done with another bus line. Here we choose line 10 for this purpose. When line 10 is 0, the CPU selects a RAM , and when this line is equal to 1,it selects the ROM.

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

**Memory Connection to CPU**

RAM and ROM chips are connected to a CPU through the data and address buses. The low-order lines in the address bus select the byte within the chips and other lines in the address bus select a particular chip through its chip select inputs. The connection of memory chips to the CPU is shown in Figure. 11.4. This configuration gives a memory capacity of 512 bytes of RAM and 512 bytes of ROM. It implements the memory map of table 11.1. 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. This is done through a 2 x 4 decoder whose outputs go to the CSI inputs in each RAM chip. Thus, when address lines 8 and 9 are equal to 00, the first RAM chip is selected. When 01, the second RAM chip is selected, and so on. The RD and WR outputs from the microprocessor are applied to the inputs of each RAM chip.

**Figure 11.4 Memory connection to the CPU**




The selection between RAM and ROM is achieved through bus line 10. The RAM's are selected when the bit in this line is 0 and the ROM when the bit is 1. The other chip select input in the ROM is connected to the Rd control line for the ROM chip to be enabled only during a read

operation. Address bus lines 1 to 9 are applied to the input address of ROM without going through the decoder. This assigns addresses 0 to 511 to RAM and 512 to 1023 to ROM. The data bus of the ROM has only an output capability, whereas the data bus connected to the RAM's can be transfer information in both directions.

The example just shown gives an indication of the interconnection complexity that can exist between memory chips and the CPU. The more chips that are connected, the more external decoders are required for selection among the chips. The designer must establish a memory map that assigns addresses to the various chips from which the required connections are determined.

## 11.3. AUXILIARY MEMORY

The auxiliary memory devices used in computer systems are magnetic disks, tapes, drums, magnetic bubble memory and optical disks. The most common auxiliary memory devices are magnetic disks and tapes. The physical properties of these storage devices can be quite complex; their logical properties can be characterized and compared by a few parameters. The important characteristics of any device are its access mode, access time, transfer rate, capacity and cost. The average time required to reach a storage location in memory and obtain its contents is called the access time. In electromechanical devices with moving parts such as disks and tapes, the access time consists of a seek time required to position the read-write head to a location and a transfer time required to transfer data to or from the device.

**Magnetic Disks**

A Magnetic disk is a circular plate constructed of metal or plastic coated with magnetized material. 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 the magnetized surface in spots along concentric circles called tracks. The tracks are commonly divided into sections called sectors. The minimum quantity of information, which can be transferred, is a sector. The subdivision of one disk surface into tracks and sectors.

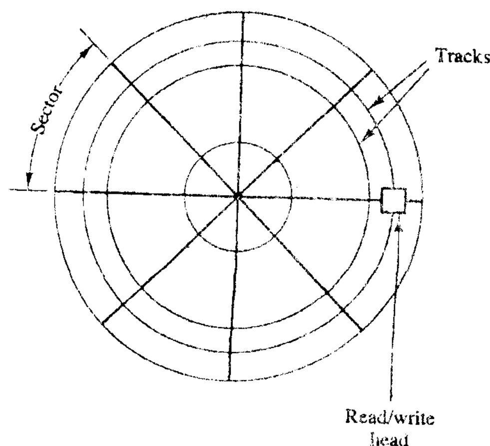



**Figure 11.5 Magnetic disk**

Some units use a single read/write head for each disk surface. In this type of unit, the track address bits are used by a mechanical assembly to move the head into the specified track position before reading or writing. In other disk systems, separate read/write heads are provided for each track in each surface. The address bits can then select a particular track electronically through a decoder circuit. The type of unit is more expensive and is found only in very large computer systems.

Permanent timing tracks are used in disks to synchronize the bits and recognize the sectors. Address bits that specify the disk number, the disk surface, the sector number and the track within the sector address a disk system. After the read/write heads are positioned in the specified track, the system has to wait until the rotating disk reaches the specified sector under the read/write head. Information transfer is very fast once the beginning of a sector has been reached. Disks may have multiple heads and simultaneous transfer of bits from several tracks at the same time. A track in a given sector near the circumference is longer than a track near the center of the disk. If bits are recorded with equal density, some tracks will contain more recorded bits than others. To make all the records in a sector of equal length, some disks use a variable recording density with higher density on tracks near the center than on tracks near the circumference.

Disks that are permanently attached to the unit assembly and cannot be removed by the occasional user are called hard disks. A disk drive with removable disks is called a floppy disk.

**Magnetic Tape**

A magnetic tape consists of the electrical, mechanical, and electronic components to provide the parts and control mechanism for a magnetic tape unit. The tape itself is a strip of plastic coated with a magnetic recording medium. Bits are recorded as magnetic spots on the tape along several tracks. Usually, seven or nine bits are recorded simultaneously to form a character together with a parity bit. Read/write heads are mounted one in each track 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. They cannot be stopped or started fast enough between individual characters. For this reason, the information is recorded in blocks referred to as records. Gaps of unrecorded tape are inserted between records where the tape can be stopped. The tape starts moving while in a gap and attains its constant speed by the time it reaches its next record. Each record on tape has an identification bit pattern at the beginning and end. By reading the bit pattern at the beginning, the tape control identifies the record number. By reading the bit pattern at the end of the record, the control recognizes the beginning of a gap. A tape unit is addressed by specifying the record number and the number of characters in the record.

## 11.4. ASSOCIATIVE MEMORY

Many data processing applications require the search of items in a table stored in memory. An assembler program searches the symbol address table in order to extract the symbol's binary equivalent. An account number may be searched in a file to determine the holder's name and account status. The established way to search a table is to store all items where they can be addressed in sequence. The search procedure is a strategy for choosing a sequence of addresses, reading the content of memory at each address, and comparing the information read with the item being searched until a match occurs. The number of access to memory depends on the location of the item and the efficiency of the search algorithm. Many search algorithms

have been developed to minimize the number of access while searching for an item in a random or sequential access memory.

The time required to find an item stored in memory can be reduced considerably if stored data can be identified for access by the content of the data itself rather than by an address. A memory unit accessed by content is called an associative memory or content addressable memory (CAM). This type of memory is accessed simultaneously and in parallel on the basis of data content rather than by specific address or location. When a word is written in an associative memory, no address is given. The memory is capable of finding an empty unused location to store the word. When a word is to be read from an associative memory, the content of the word is specified. The memory locates all words, which match the specified content, and marks them for reading.

The associative memory is uniquely suited to do parallel searches by data association. Searches can be done on an entire word or on a specific field with in a word. 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. Associative memories are used in application where the search time is very critical and must be very short.

Hardware Organization

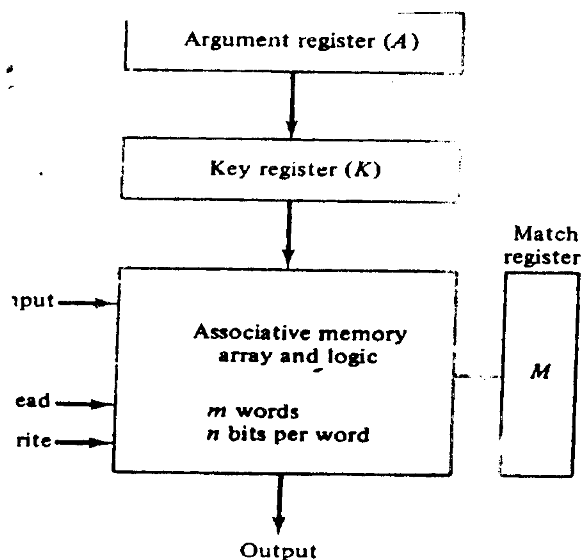



**Figure 11.6 Block diagram of associative Memory**

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

| | | | |
|---|---|---|---|
| A | 101 | 111100 | |
| K | 111 | 000000 | |
| Word1 | 100 | 111100 | no match |
| Word2 | 101 | 000001 | match |

Word2 matches the unmatched argument field because the three leftmost bits of the argument and the word are equal.

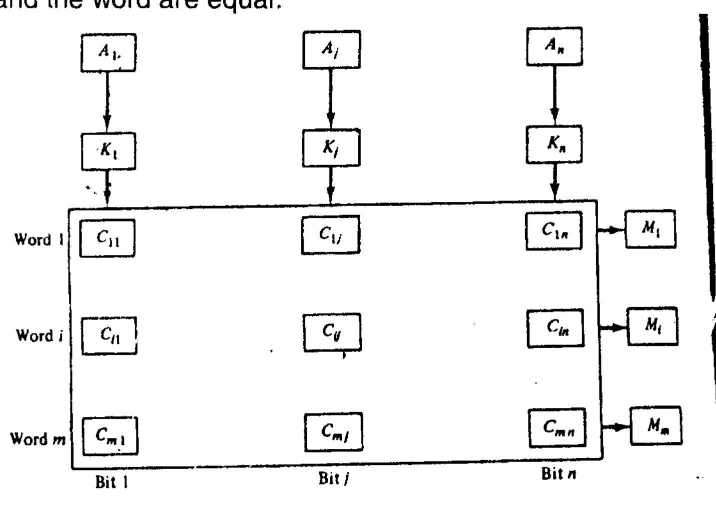



**Figure 11.7 Associative Memory of m word, n cells, per word**

The letter C with two subscripts marks the cells in the array. The first subscript gives the word number and the second specifies the bit position in the word. The cell $C_{ij}$ is the cell for bit j in word i. A bit $A_{ij}$ in the argument register is compared with all the bits in the column j of the array provided that $k_j$ =1. This is done for all columns j = 1,2,..n. If a match occurs between all the unmasked bits of the argument and the bits in word i, the corresponding bit $M_i$ in the match register is set to 1. If one or more unmasked bits of the argument and the word do not match, $M_i$ is cleared to 0.

## 11.5. CACHE MEMORY

Cache memory that can dramatically increase the performance of a computer system at relatively little cost.

Cache memory provides system designers with a way of exploiting high-speed processors without incurring the cost of large high-speed memory systems. The word cache is pronounced

"*cash*" or "*cash-ay*" and is derived from the French word meaning *hidden.* Cache memory is hidden from the programmer and appears as part of the system's memory space. There's nothing mysterious about cache memory—it's simply a quantity of very high-speed memory that can be accessed rapidly by the processor. The element of magic stems from the ability of systems with a tiny cache memory (e.g., 512M bytes of cache memory in a system with 2 Gbytes of DRAM) and expects the processor to make over 95% of its accesses to the cache rather than the slower DRAM.

Cache memory can be understood in everyday terms by its analogy with a diary or notebook used to jot down telephone numbers. A telephone directory contains hundreds of thousands of telephone numbers and nobody carries a telephone directory around with them. However lots of people have a notebook with a hundred or so telephone numbers that they keep with them. Although the fraction of all possible telephone numbers in someone's notebook might be less than 0.01%. the probability that their next call will be to a number in the notebook is high because they frequently call the same people. Cache memory operates on exactly the same principle by locating frequently accessed information in the cache memory rather than in the much slower main memory. Unfortunately, unlike the personal notebook, the computer cannot know, in advance, what data is most likely to be accessed? You could say that computer caches operate on a learning principle. By experience they learn what data is most frequently used and then transfer it to the cache.

A block of cache memory sits on the processor's address and data buses in parallel with the much larger main memory. Note that the implication of parallel in the previous sentence is that data in the cache is also maintained in the main memory. To return to the analogy with the telephone notebook, writing a friend's number in the notebook does not delete their number in the directory.

**Structure of a cache memory**

Cache memory relies on the same principle as the notebook with telephone numbers. The probability of accessing the next item of data in memory isn't a random function. Because of the nature of programs and their attendant data structures, the data required by a processor is often highly clustered. This aspect of memories is called the locality of reference and makes the use of cache memory possible.

A cache memory requires a cache controller to determine whether the data currently being accessed by the CPU resides in the cache or whether it must be obtained from the main memory. When the current address is applied to the cache controller, the controller returns a signal called hit that is asserted if the data is currently in the cache. Before we look at how cache memories are organized, we will demonstrate their effect on a system's performance.

**Effect of Cache Memory on Computer Performance**

The principal parameter of a cache system is its hit ratio, h that defines the ratio of hits to all accesses. The hit ratio is determined by statistical observations of the operation of a real system and cannot readily be calculated. Furthermore, the hit ratio is dependent on the specific nature of the programs being executed. It is possible to have some programs with very high hit ratios and others with very low hit ratios. Fortunately, the effect of locality of reference usually means that the hit ratio is very high—often in the region of 95%. Before calculating the effect of a cache memory on a processor's performance, we need to introduce some terms.

| | |
|---|---|
| Access time of main store | $t_m$ |
| Access time of cache memory | $t_c$ |
| Hit ratio | h |
| Miss ratio | m |
| Speedup ratio | S |

The merit of a computer with cache is called the speedup ratio that indicates how much the cache accelerates the memory's access time. The speedup ratio is defined as the ratio of the memory system's access time without cache to its access time with cache.

The effect of cache memory on the performance of a computer depends on many factors including the way in which the cache is organized and the way in which data is written to main memory when a write access takes place. We will return to some of these considerations when we have described how cache systems are organized.

**Cache Organization**

There are at least three ways of organizing a cache memory—direct-mapped, associative mapped and set associative mapped cache. Each of these systems has its own performance: cost tradeoff.

**Direct-Mapped Cache**

The easiest way of organizing a cache memory employs direct mapping that relies on a simple algorithm to map data block i from the main memory into data block j in the cache. For the purpose of this section we will regard the smallest unit of data held in a cache as a line that is made up of typically two or four consecutive words. The line is the basic unit of data that is transferred between the cache and main store and varies between 4 and 32 bytes.

The direct mapped cache diagram illustrates the structure of a highly simplified direct-mapped cache. As you can see, the memory space is divided into sets and the sets into lines. This memory is composed of 32 words and accessed by a 5-bit address bus from the CPU. For the purpose of this discussion we need only consider the set and line (as it doesn't matter how many words there are in a line). The address in this example has a 2-bit set field, a 2-bit line field and a 1-bit word field. The cache memory holds $2^2$ = 4 lines of two words. When the processor generates an address, the appropriate line in the cache is accessed. For example, if the processor generates the 5-bit address $10100_2$, line 2 in set 2 is accessed.

There are four possible lines numbered two—a line 2 in set 0, a line 2 in set 1, a line 2 in set 2 and a line 2 in set 3. In this example the processor accessed line 2 in set 2. The obvious question is, "How does the system know whether the line 2 accessed in the cache is the line 2 from set 2 in the main memory?"

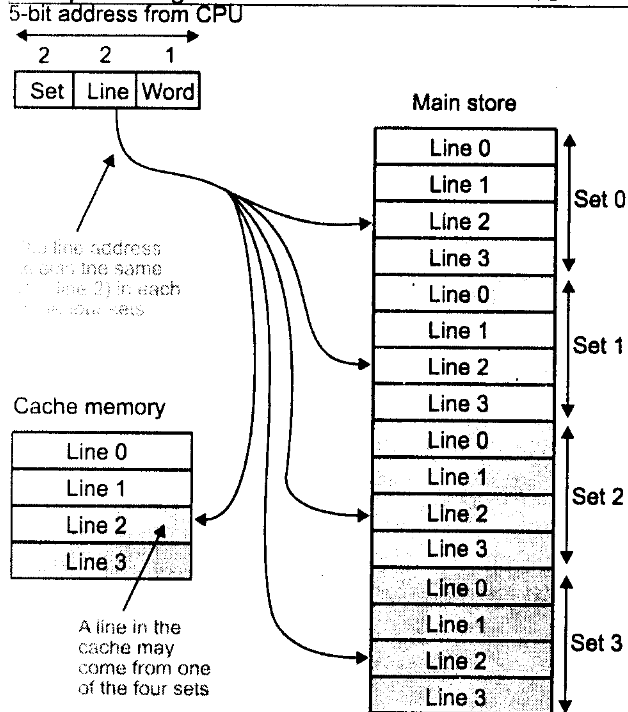



**Figure 11.8 Direct Mapped Cache**

In the above figure shows how a direct mapped cache resolves the contention between lines. Each line in the cache memory has a tag or label that identifies which set this particular line belongs to. When the processor accesses line 2, the tag belonging to line 2 in the cache is sent to a comparator. At the same time the set field from the processor is also sent to the comparator. If they are the same, the line in the cache is the desired line and a hit occurs.

If they are not the same, a miss occurs and the cache must be updated. The old line 2 from set 1 is either simply discarded or rewritten back to main memory depending on how the updating of main memory is organized.

The cache memory itself is nothing more than a block of very high speed random accesses read/write memory. The cache tag RAM is a fast combined memory and comparator that receives both its address and data inputs from the processor's address bus. The cache tag RAM's address input is the line address from the processor that is used to access a unique location (one for each of the possible lines). The data in the cache tag RAM at this location is the tag associated with that location. The cache tag RAM also has a data input that receives the tag field from the processor's address bus. If the tag field from the processor matches the contents of the tag (i.e., set) field being accessed, the cache tag RAM returns a hit signal.

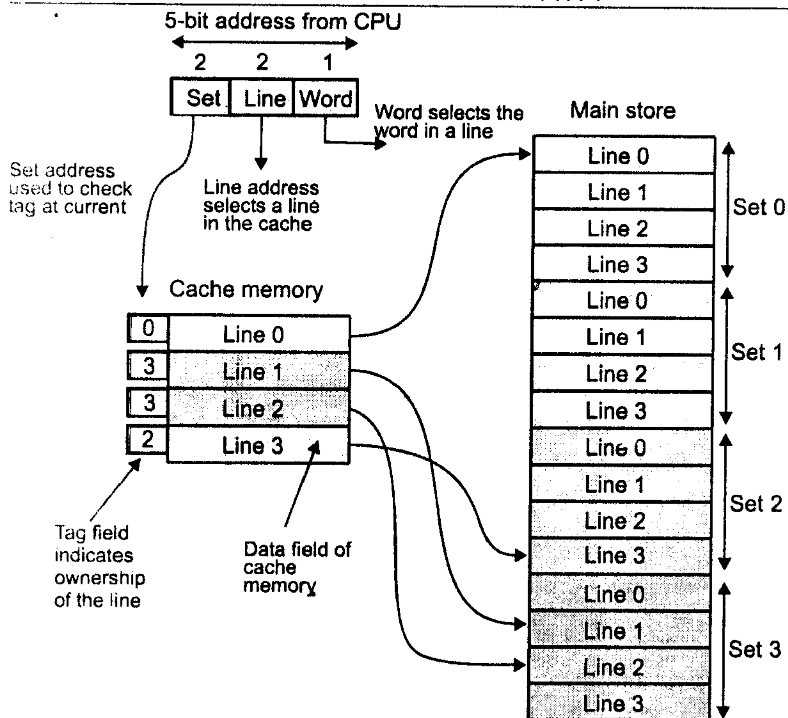



**Figure 11.9 Resolving contention between lines in a direct-mapped cache**

The advantage of the directly mapped cache is almost self-evident. Both the cache memory and the cache tag RAM are widely available devices that, apart from their speed, are no more complex than other mainstream devices. Moreover, the direct mapped cache requires no complex line replacement algorithm. If line *x* in set *y* is accessed and a miss takes place, line *x* from set *y* in the main store is loaded into the frame for line *x* in the cache memory and the tag set to *y*. That is, there is no decision concerning which line has to be rejected when a new line is to be loaded.

Another important advantage of direct mapped cache is its inherent parallelism. Since the cache memory holding the data and the cache tag RAM are entirely independent, they can both be accessed simultaneously. Once the tag has been matched and a hit has occurred, the data from the cache will also be valid (assuming the two cache data and cache tag memories have approximately equal access times).

The disadvantage of direct mapped cache is almost a corollary of its advantage. A cache with n lines has one restriction—at any instant it can hold only one line numbered *x*. What it cannot do is hold a line *x* from set *p* and a line *x* from set *q*. This restriction exists because there is one page frame in the cache for each of the possible lines. Consider the following fragment of code:

```
REPEAT
        Get_data
        Compare
UNTIL match OR end_of_data
```

This innocuous fragment of code reads a string of data from a buffer and then compares it with another string until a match is found. Suppose that in the compiled version of this code part of the Get_data routine is in set *x*, line *y* and that part of the Compare routine is in set *z*, line *y*. Because a direct-mapped cache can hold only one line *y* at a time, the frame corresponding to line *y* must be reloaded twice for each path through the loop. Consequently, the performance of a direct-mapped cache can be very poor under certain circumstances. However, statistical measurements on real programs indicate that the very poor worst-case behavior of direct-mapped caches has no significant impact on their average behavior.

Suppose a cache is almost empty and most of its lines have not yet been loaded with active data. Certain lines may have to be swapped out of the cache frequently because data in the main store just happens to share the same line numbers. In spite of this objection to direct mapped cache, it is very popular because of its low cost of implementation and high speed.

**Associative Mapped Cache**

One way of organizing a cache memory that overcomes the limitations of direct mapped cache is described in below figure 11.10. Ideally, we would like a cache that places no restrictions on what data it can contain. The associative-cache is such a memory.

An address from the processor is divided into three fields: the tag, the line, and the word. Like the direct mapped cache, the smallest unit of data transferred into and out of the cache is the line. Unlike the direct-mapped cache, there's no predetermined relationship between the location of lines in the cache and lines in the main memory. Line *p* in the memory can be put in line *q* in the cache with no restrictions on the values of *p* and *q*. Consider a system with 1 Mbytes of main store and 64 Kbytes of associatively mapped cache. If the size of a line is four 32-bit words (i.e., 16 bytes), the main memory is composed of $2^{20}/16$ = 64K lines and the cache is composed of 216/16 = 4096 lines. Because an associative cache permits any line in the main store to be loaded into one of its lines, line *i* in the associative cache can be loaded with any one of the 64K possible lines in the main store. Therefore, line *i* requires a 16-bit tag to uniquely label it as being associated with line *i* from the main store.

When the processor generates an address, the word bits select a word location in both the main memory and the cache. Unlike the direct-mapped cache memory, the line address from the processor can't be used to address a line in the associative cache. Why? Because each line in the direct-mapped cache can come only from one of *n* lines in the main store (were *n* is the number of sets). The tag resolves which of the lines is actually present. In an associative cache any of the 64K lines in the main store can be located in any of the lines in the cache. Consequently, the associative cache requires a 16-bit tag to identify one of the $2^{16}$ lines from the main memory. Because the cache's lines are not ordered, the tags are not ordered and cannot be stored in a simple look up table like the direct mapped cache. In other words, when the CPU accesses line *i*, it may be anywhere in the cache or it may not be in the cache.

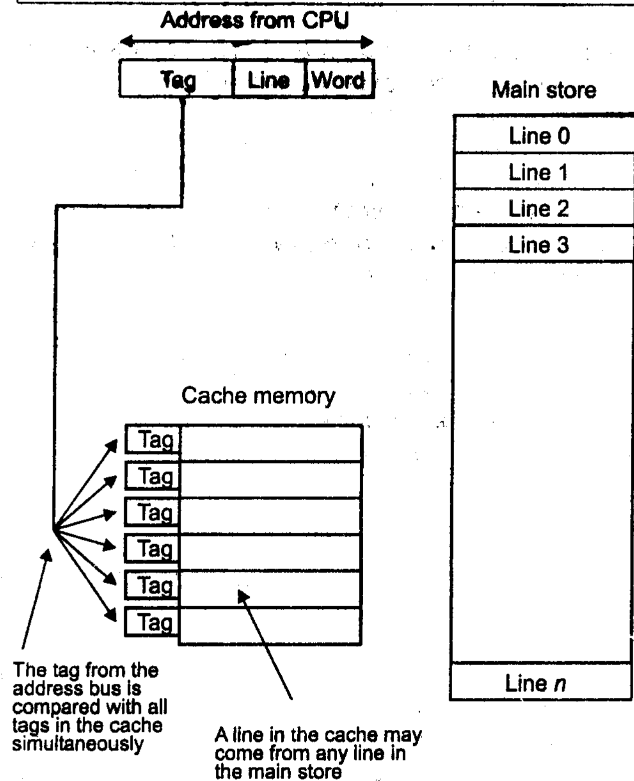



**Figure11.10** Associative-mapped cache

Associative cache systems employ a special type of memory called associative memory. An associative memory has an $n$-bit input but not necessarily $2^n$ unique internal locations. The $n$-bit address input is a tag that is compared with a tag field in each of its locations simultaneously. If the input tag matches a stored tag, the data associated with that location is output. Otherwise the associative memory produces a miss output. An associative memory is not addressed in the same way that a computer's main store is addressed. Conventional computer memory requires the explicit address of a location, whereas an associative memory is accessed by asking, "Do you have this item stored somewhere?"

Associative cache memories are efficient because they place no restriction on the data they hold. In figure 8.20 the tag that specifies the line currently being accessed is compared with the tag of each entry in the cache simultaneously. In other words, all locations are accessed at once. Unfortunately, large associative memories are not yet cost-effective. Once the associative cache is full, a new line can be brought in only by overwriting an existing line that requires a suitable line replacement policy.

**Set Associative Mapped Cache**

Most computers employ an arrangement that is a compromise between the direct-mapped cache and the fully associative cache. This compromise system is called a set associative cache.

A set associative cache memory is nothing more than several direct-mapped caches operated in parallel. The simplest arrangement is called a 2-way set associative cache and consists of

two direct-mapped cache memories. Each line in the cache system is duplicated; for example there are two line 5s in the cache. Consequently, it is now possible to store two line 5s, one line 5 from set x and one line 5 from set y. A set associative cache memory is therefore associative within a set. If the cache has n parallel sets, an *n*-way comparison is performed in parallel against all members of the set. Because n in small (typically 2 to 16), the logic required to perform the comparison is not complex.

When the processor accesses memory, the appropriate line in each of four direct-mapped caches is accessed simultaneously. Since there are four lines, a simple associative match can be used to determine which (if any) of the lines in cache are to supply the data. In below figure the hit output from each direct-mapped cache is fed to an OR gate which generates a hit if any of the caches generate a hit.

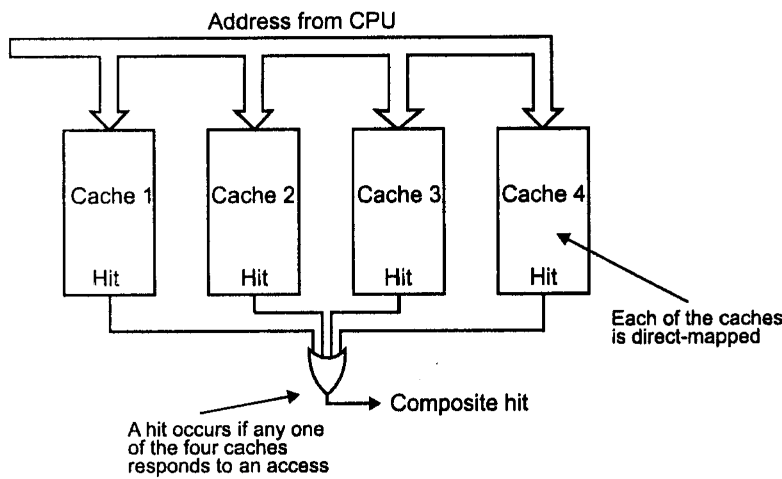



**Figure11.11** Set associative-mapped cache

## 11.6. VIRTUAL MEMORY

In a memory hierarchy system, programs and data are first stored in auxiliary memory, portions of a program or data are bought into main memory as they are needed by the CPU. Virtual memory is a concept used in some large computer systems that permit the user to construct programs as though a large memory space were available, equal to the totality of auxiliary memory. Each address that is referenced by the CPU goes through an address mapping from the so-called virtual address to a physical address in main memory. Virtual memory is used to give programmers the illusion that they have a very large memory at their disposal, even though the computer actually has a relatively small main memory. A virtual memory system provides a mechanism for translating program-generated addresses into correct main memory locations. This is done dynamically, while programs are being executed in the CPU. The translation or mapping is handled automatically by the hardware by means of a mapping table.

### 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 an illustration, consider a computer with a main-memory capacity of *32K* words (K = 1024). Fifteen bits are needed to specify a physical address in memory since $32K = 2^{15}$. Suppose that the computer has available auxiliary memory for storing $2^{20}$ = 1O24K words. Thus auxiliary memory has a capacity for storing information equivalent to the capacity of 32 main memories. Denoting the address space by N and the memory space by M, we then have for this example N = *1024K* and M = *32K.*

In a multiprogram computer system, programs and data are transferred to and from auxiliary memory and main memory based on demands imposed by the CPU. Suppose that program 1 is currently being executed in the CPO. Program 1 and a portion of its associated data are moved from auxiliary memory into main memory as shown in Fig. 12-16. Portions of programs and data need not be in contiguous locations in memory since information is being moved in and out, and empty spaces may be available in scattered locations in memory.

In a virtual memory system, programmers are told that they have the total address space at their disposal. Moreover, the address field of the instruction code has a sufficient number of bits to specify all virtual addresses. In our example, the address field of an instruction code will consist of 20 bits but physical memory addresses must be specified with only 15 bits. Thus CPU will reference instructions and data with a 20-bit address, but the information at this address must be taken from physical memory because access to auxiliary storage for individual words will be prohibitively long.

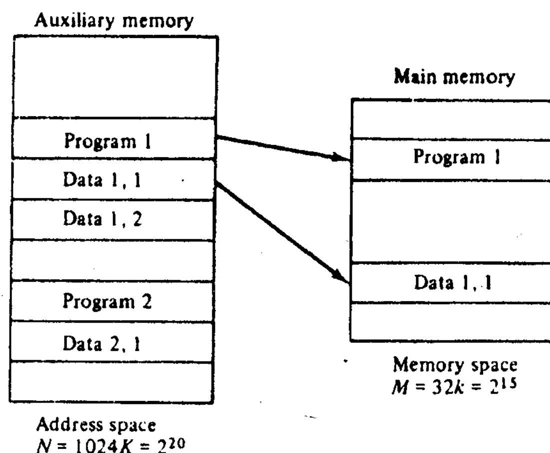



***Figure11.12 Relation between address and memory space in a virtual memory system***

(Remember that for efficient transfers auxiliary storage mOoves an entire record to the main memory). A table is then needed, as shown in figure 11-13, 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 in fig 11.13 or in main memory. In the first case, an additional memory unit is required as well as one extra memory access time. In this second case the table

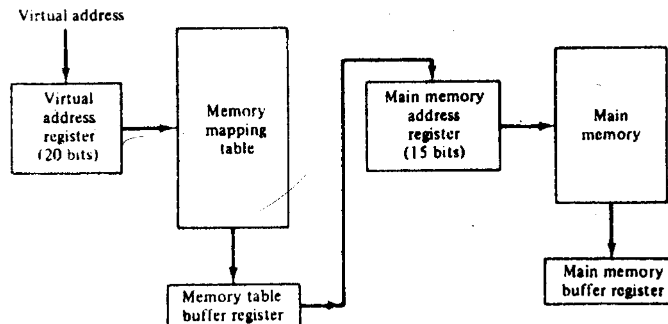



**Figure11.13 Memory table for mapping a virtual address**

takes space from main memory and two accesses to memory are required with the program running at half speed. A third alternative is to use an associative memory as explained below.

**Address Mapping Using Pages**

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

Consider a computer with an address space of 8K and a memory space of 4K. If we split each into groups of 1K words we obtain eight pages and four blocks as shown in Fig. 11.14. At any given time, up to four pages of address space may reside in main memory in anyone of the four blocks.

The mapping from address space to memory space is facilitated if each virtual address is considered to be represented by two numbers: a page number address and a line within the page. In a computer with $2^p$ words per page, $p$ bits are used to specify a line address and the remaining high-order bits of the virtual address specify the page number. In the example of Fig. 11.14, a virtual address has 13 bits. Since each page consists of $2^{10}$ = 1024 words, the high--order three bits of a virtual address will specify one of the eight pages and the low-order 10 bits give the line address within the page. Note that the line address in address space and memory space is the same; the only mapping required is from a page number to a block number.

The organization of the memory-mapping table in a paged system is shown in Fig. 11.15. The memory-page table consists of eight words, one for each page. The address in the page table de notes 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. A 0 in the presence bit indicates that this page is not available in main memory. The CPU references a word in memory with a virtual address of 13 bits. The three high-order bits of the virtual address specify a page number and also an address for the memory-page table. The content of the

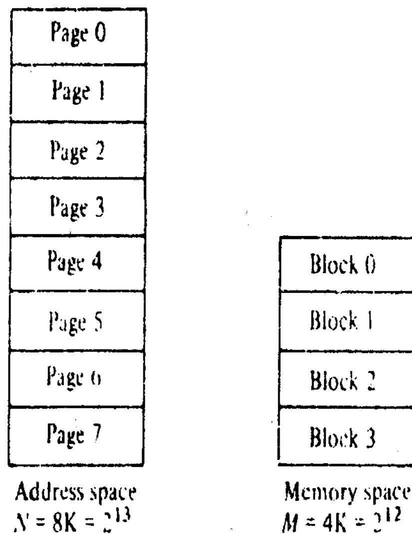



**Figure 11.14 Address space and memory space split into groups of 1k words**

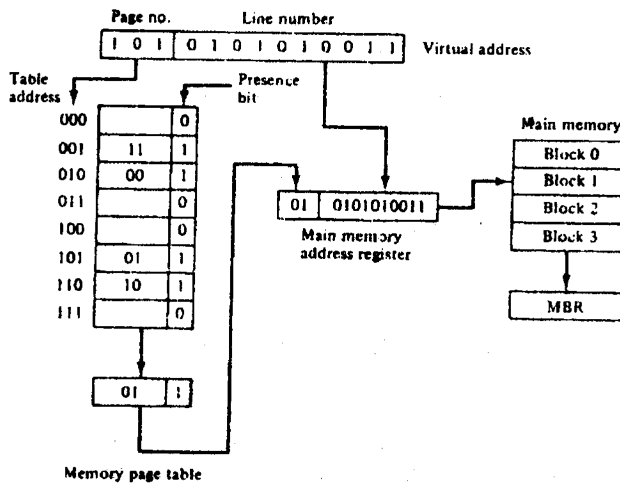



**Figure 11.15 Memory table in a paged system**

word in the memory page table at the page number addresses read out in to the memory table buffer register . If the presence bit is 1, the block number thus read is transferred to the two high order bits of the main memory address register. The line number from the virtual address is transferred in to the 10 low-order bits of the memory address register. A read signal to main memory transfers the content of the word to the main memory buffer register ready to be used by the CPU.If the presence bit in the word read from the page table is 0, it signifies that the content of the word referenced by the virtual address does not reside in main memory. A call to the operating system is then generated to fetch the required page from auxiliary memory and place it in to main memory before resuming computation.

## ASSOCIATIVE MEMORY PAGE TABLE

A random access memory page table is inefficient with respect to storage utilization. In the example of fig,11.15 we observe that eight words of memory are needed one for each page ,but the least four words will always be marked empty because main memory cannot accommodate more than four blocks In general a system with n-pages and m-blocks would require a memory page table of n-locations of which up to m-blocks will be marked with block numbers and all others will be empty. As a second numerical example, consider an address space of 1024k and memory space of 32k words. If each page or block contains 1k words, the number of pages is 1024 and the number of blocks is 32. The capacity of memory page table must be 1024 words and only 32 locations may have a presence bit equal to 1. At any given time, at least 992 will be empty and not in use.

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

Consider again the case of eight pages and four blocks as in the example of Fig. 11.15. We replace the random access memory-page table with an associative memory of four words as shown in Fig. 11.16. 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.

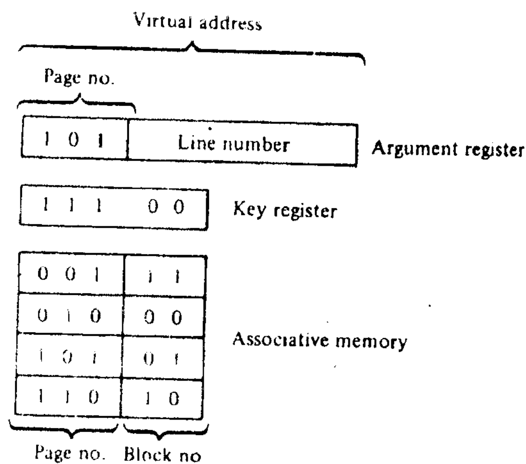



**Figure 11.16 An associative memory page table**

***Page Replacement***

A virtual memory system is a combination of hardware and software techniques. The memory management software system handles all the software operations for the efficient utilization of memory space. It must decide (1) which page in main memory ought to be removed to make room for a new page, (2) when a new page is to be transferred from auxiliary memory to n:_in memory, and (3) where the page is to be placed in main memory. The hardware mapping mechanism and the memory management software together constitute the architecture of a virtual memory.

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

When a page fault occurs in a virtual memory system, it signifies that the page referenced by the CPU is not in main memory. A new page is then transferred from auxiliary memory to main memory. If main memory is full, it would be necessary to remove a page from a memory block to make room for the new page. The policy for choosing pages to remove is determined from the replacement algorithm that is used. The goal of a replacement policy is to try to remove the page least likely to be referenced in the immediate future. Two of the most common replacement algorithms used are the *first-in first-out* and least recently used(LRU). The FIFO algorithm selects for replacement the page that has been in memory the longest time. Each time a page is loaded into memory, its identification number is pushed in to a FIFO stack. FIFO will be full whenever the memory has no more empty blocks. When a new page must be loaded the page recently bought is removed. The page to be removed is easily determined its identification number is at the top of the FIFO stack. The replacement policy has the advantage

of being easy to implement. It has the advantage that under certain circumstances pages are removed and loaded in to memory too frequently.

The LRU policy is more difficult to implement but has been more attractive on the assumption that least recently used page is better candidate for removal than least recently loaded page as in FIFO. The LRU algorithm can be implemented by associating a counter with every page that is in main memory. When a page is referenced, its associated counter is set to zero. At fixed intervals of time, the counter associated with all pages presently in memory is incremented by 1. The least recently used page is the page with the highest count. The counters are often called aging registers, as their count indicates their age, that is, how long ago their associated pages have been referenced.

## 11.7. SUMMARY

In this lesson we learnt the concept of main memory. In this lesson we introduced the concept of Auxiliary Memory, Cache Memory and Virtual Memory, In this lesson we have learnt the different types of memories like RAM, ROM.

## 11.8. MODEL QUESTIONS

1. Explain about Main memory?
2. Explain the differences between RAM and ROM?
3. Define Cache Memory? What are the advantages of Cache Memory?
4. Explain the features in Virtual Memory?

## 11.9. REFERENCES

1. Computer System Architecture – M. Morris Mano – Eastern Economy Edition.
2. Digital Design - M.Morris Mano – Englewood Cliffs.

**Sri. C.V.P.R.PRASAD,** M.C.A.,

(కత్తిరించి పంపవలెను)

## అధ్యాపకుల, విద్యార్థుల సలహాలు, సూచనలు :

అధ్యాపకులు, విద్యార్థులు ఈ స్టడీ మెటీరియల్‌కు సంబంధించిన సలహాలు, సూచనలు, ముద్రణ దోషాలు తెలియపరచినచో, పునర్ముద్రణలో తగు చర్యలు తీసుకొనగలము. తెలియపరచవలసిన చిరునామా : **డిప్యూటీ డైరెక్టర్, దూరవిద్యా కేంద్రం, ఆచార్య నాగార్జున విశ్వవిద్యాలయం, నాగార్జున నగర్ - 522 510.**

**Title**

**Computer Organization**

(కత్తిరించి పంపవలెను)

(కత్తిరించి పంపవలెను)

(కత్తిరించి పంపవలెను)