Also available as e Book
www.technicalpublications.org

Computer Organization

As per New Syllabus of
Gujarat Technological University
Semester - IV (CE / CSE / IT / I&CT)



Includes
- Solved University Questions Dec.-2002 to May-2014

Include
Computer Organization Tutorial

TECHNICAL PUBLICATIONS
An Up-Thrust for Knowledge

A. P. Godse
Dr. D. A. Godse

As per New Syllabus of

GUJARAT TECHNOLOGICAL UNIVERSITY

COMPUTER ORGANIZATION

Atul P. Godse

M.S. Software Systems (BITS Pilani)
B.E. Industrial Electronics
Formerly Lecturer in Department of Electronics Engg.
Vishwakarma Institute of Technology
Pune

Dr. Mrs. Deepali A. Godse

M.E., Ph.D. (Computer Engg.)
Head of Information Technology Department
Sinhgad Vidyaprasarini's College of Engineering for Women
Pune

STUDENT BOOK STALL
Purohit Arcade Nr Iskcon Temple,
V V Nagar M 98244 46491
Please check the Book Your Requirement
No Exchange No Replacement

TECHNICAL PUBLICATIONS
An Up-Thrust for Knowledge

STUDENT BOOK STALL
Purohit Arcade Nr Iskcon Temple,
V V Nagar M 98244 46491
Please check the Book Your Requirement
No Exchange No Replacement

COMPUTER ORGANIZATION

Semester : IV (CE / CSE / IT / I & C)

First Printed in India : May 2018
First Edition : January 2018 (As per revised syllabus of Gujarat Technological University)

This edition is for sale in India only. Sale and Purchase of this book outside of India is unauthorized by the publisher.

© Copyright with Authors

All publishing rights (printed and ebook version) reserved with Technical Publications. No part of this book should be reproduced in any form, Electronic, Mechanical, Photocopy or any information storage and retrieval system without prior permission in writing, from Technical Publications, Pune.

Published by :
TECHNICAL PUBLICATIONS
Amit Residency, Office No.1, 412, Shaniwar Peth, Pune - 411030, M.S. INDIA
Ph. : +91-020-24495496/97, Telefax : +91-020-24495497
Email : technical@vtubooks.com, Website : www.technicalpublications.org

Printer :
Raosaheb Offset
S.no.15, Sukhsagar Nagar,
Shaghar Chowk, Ambegaon Khurd, Pune
Pune - 411 046

Price : ₹ 395/-

ISBN 978-93-333-0067-7

ISBN 978933330067 (Printed book)

ISBN 9789333300684 (E-Book)

eBook available on www.technicalpublications.org

Course 13

The importance of **Computer Organization** is well known in various engineering fields. Overwhelming response to our books on various subjects inspired us to write this book. The book is structured to cover the key aspects of the subject **Computer Organization.**

The book uses plain, lucid language to explain fundamentals of this subject. The book provides logical method of explaining various complicated concepts and stepwise methods to explain the important topics. Each chapter is well supported with necessary illustrations, practical examples and solved problems. All the chapters in the book are arranged in a proper sequence that permits each topic to build upon earlier studies. All care has been taken to make students comfortable in understanding the basic concepts of the subject.

Representative questions have been added at the end of each section to help the students in picking important points from that section.

The book not only covers the entire scope of the subject but explains the philosophy of the subject. This makes the understanding of this subject more clear and makes it more interesting. The book will be very useful not only to the students but also to the subject teachers. The students have to omit nothing and possibly have to cover nothing more.

We wish to express our profound thanks to all those who helped in making this book a reality. Much needed moral support and encouragement is provided on numerous occasions by our whole family. We wish to thank the **Publisher** and the entire team of **Technical Publications** who have taken immense pain to get this book in time with quality printing.

Any suggestion for the improvement of the book will be acknowledged and well appreciated.

Authors
A. P. Godse
Dr. D. A. Godse

Dedicated to God.

Syllabus (Computer Organization)

1. Computer Data Representation (Chapters - 1, 2)

Basic computer data types, Complements, Fixed point representation, Register transfer and micro-operations : Floating point representation, Register transfer language, Register transfer, Bus and memory transfers (Tree-state bus buffers; Memory transfer), Arithmetic micro-operations, Logic micro-operations, Shift micro-operations, Arithmetic logical shift unit.

2. Basic Computer Organization and Design (Chapter - 3)

Instruction codes, Computer registers, Computer instructions, Timing and control, Instruction cycle, Memory-reference instructions, Input-output and interrupt, Complete computer description, Design of basic computer, Design of accumulator unit.

3. Programming the Basic Computer (Chapter - 4)

Introduction, Machine language, Assembly language, Assembler, Program loops, Programming arithmetic and logic operations, Subroutines, I-O programming.

4. Microprogrammed Control (Chapter - 5)

Control memory, Address sequencing, Micro program example, Design of control unit.

5. Central Processing Unit (Chapter - 6)

Introduction, General register organization, Stack organization, Instruction format, Addressing modes, Data transfer and manipulation, Program control, Reduced Instruction Set Computer (RISC).

6. Pipeline and Vector Processing (Chapter - 7)

Flynn's taxonomy, Parallel processing, Pipelining, Arithmetic pipeline, Instruction, Pipeline, RISC pipeline, Vector processing, Array processors.

7. Computer Ar

Introduction, Ad
algorithm), Divis
unit.

8. Input-Outpu

Input-Output In
DMA, Input-Out

9. Memory Or

Memory hierar
Virtual memor

10. Multiproc

Characteristic
Inter-process
multiprocess

7. Computer Arithmetic (Chapter - 8)

Introduction, Addition and subtraction, Multiplication algorithms (Booth multiplication algorithm), Division algorithms, Floating point arithmetic operations, Decimal arithmetic unit.

8. Input-Output Organization (Chapter - 9)

Input-Output interface, Asynchronous data transfer, Modes of transfer, Priority interrupt, DMA, Input-Output Processor (IOP), CPU|OP communication, Serial communication.

9. Memory Organization (Chapter - 10)

Memory hierarchy, Main memory, Auxiliary memory, Associative memory, Cache memory, Virtual memory.

10. Multiprocessors (Chapter - 11)

Characteristics of multiprocessors, Interconnection structures, Inter-processor arbitration, Inter-processor communication and synchronization, Cache coherence, Shared memory multiprocessors.

Table of Contents

Chapter - 1 Computer Data Representation (1 - 1) to (1 - 20)

1.1 Basic Computer Data Types ..... 1 - 2

1.2 Fixed Point Numbers ..... 1 - 2

1.2.1 Signed Magnitude Representation ..... 1 - 2

1.2.2 1's Complement Representation ..... 1 - 3

1.2.3 2's Complement Representation ..... 1 - 3

1.2.4 Sign Extension ..... 1 - 4

1.3 Floating Point Representation ..... 1 - 6

1.3.1 IEEE Standard for Floating-Point Numbers ..... 1 - 6

Chapter - 2 Register Transfer and Micro-operations (2 - 1) to (2 - 52)

2.1 Register Transfer Language ..... 2 - 2

2.1.1 Features ..... 2 - 2

2.1.2 Basic Components of Register Transfer Logic ..... 2 - 2

2.1.3 Definition of Register Transfer Language ..... 2 - 3

2.2 Register Transfer ..... 2 - 4

2.3 Bus and Memory Transfers ..... 2 - 9

2.3.1 Bus Transfer ..... 2 - 9

2.3.2 Memory Transfer ..... 2 - 15

2.4 Arithmetic Micro-operations ..... 2 - 23

2.4.1 Binary Adder ..... 2 - 24

2.4.2 Parallel Adder ..... 2 - 26

2.4.3 Parallel Subtractor ..... 2 - 26

2.4.4 Addition / Subtraction Logic Unit ..... 2 - 27

2.4.5 Binary Incrementer ..... 2 - 27

2.4.6 Arithmetic Circuit ..... 2 - 28

2.5 Logic Micro-operations ..... 2 - 39
2.6 Shift Micro-operations ..... 2 - 45
2.6.1 Logic Shift ..... 2 - 46
2.6.2 Arithmetic Shift ..... 2 - 46
2.6.3 Rotate or Circular Shift Micro-operations ..... 2 - 47
2.7 Arithmetic Logic Shift Unit ..... 2 - 49

Chapter - 3 Basic Computer Organization and Design (3 - 1) to (3 - 38)

3.1 Instruction Codes ..... 3 - 2
3.2 Computer Registers ..... 3 - 6
3.3 Computer Instructions ..... 3 - 10
3.3.1 Memory Reference Instructions ..... 3 - 10
3.3.2 Register Reference Instructions ..... 3 - 11
3.3.3 Input-Output Instructions ..... 3 - 12
3.4 Timing and Control ..... 3 - 13
3.5 Instruction Cycle ..... 3 - 15
3.6 Input-Output and Interrupt ..... 3 - 22
3.6.1 Input-Output Configuration ..... 3 - 22
3.6.2 Input-Output Instructions ..... 3 - 22
3.6.3 Program Interrupt ..... 3 - 23
3.7 Complete Computer Description ..... 3 - 27
3.8 Design of Basic Computer ..... 3 - 29
3.9 Design of Accumulator Logic ..... 3 - 34
3.9.1 Control of AC Register ..... 3 - 35
3.9.2 Control of ALU ..... 3 - 36

Chapter - 4 Programming the Basic Computer (4 - 1) to (4 - 38)

4.1 Introduction ..... 4 - 2
4.2 Machine Language ..... 4 - 4
4.3 Assembly Language ..... 4 - 7
4.4 The Assembler ..... 4 - 12

4.4 Assembler ..... 4 - 15
4.4.1 First Pass ..... 4 - 16
4.4.2 Second Pass ..... 4 - 19
4.4.3 Error Diagnostics ..... 4 - 20
4.5 Program Loops ..... 4 - 22
4.6 Programming Arithmetic and Logic Operations ..... 4 - 23
4.6.1 Multiplication ..... 4 - 26
4.6.2 Double-precision Addition ..... 4 - 27
4.6.3 Logic Operations ..... 4 - 28
4.6.4 Shift Operations ..... 4 - 29
4.7 Subroutines ..... 4 - 32
4.7.1 Subroutine Parameters and Data Linkage ..... 4 - 33
4.8 Input-Output Programming ..... 4 - 34
4.8.1 Character Manipulation ..... 4 - 35
4.8.2 Program Interrupt ..... (5 - 1) to (5 - 28)

Chapter - 5 Microprogrammed Control

5.1 Introduction ..... 5 - 2
5.2 Control Memory (CM) ..... 5 - 2
5.3 Address Sequencing ..... 5 - 5
5.3.1 Conditional Branching ..... 5 - 8
5.3.2 Mapping of Instruction ..... 5 - 9
5.3.3 Subroutines ..... 5 - 11
5.4 Microprogram Example ..... 5 - 12
5.4.1 Computer Configuration ..... 5 - 12
5.4.2 Microinstruction Format ..... 5 - 14
5.4.3 Symbolic Microinstructions and Microprogram ..... 5 - 18
5.5 Design of Control Unit ..... 5 - 20
5.5.1 Decoding of Micro-operation Fields and their Further Processing ..... 5 - 20
5.5.2 Microprogram Sequencer ..... 5 - 21
5.6 Key Definitions Related to Microprogrammed Control Unit ..... 5 - 24
5.7 Applications of Microprogramming ..... 5 - 25

5.8 Hardwired Control ..... 5 - 25
5.8.1 Advantages of Hardwired Control ..... 5 - 25
5.8.2 Disadvantages of Hardwired Control ..... 5 - 26
5.9 Comparison between Hardwired and Microprogrammed Control ..... 5 - 26

Chapter - 6 Central Processing Unit (6 - 1) to (6 - 40)

6.1 Introduction ..... 6 - 2
6.2 General Register Organization ..... 6 - 5
6.3 Stack Organization ..... 6 - 8
6.3.1 Register Stack ..... 6 - 8
6.3.2 Memory Stack ..... 6 - 9
6.3.3 Notations ..... 6 - 11
6.4 Instruction Format ..... 6 - 13
6.5 Addressing Modes ..... 6 - 20
6.5.1 Implementation of Variables and Constants ..... 6 - 21
6.5.2 Indirection and Pointers ..... 6 - 21
6.5.3 Indexing and Arrays ..... 6 - 22
6.5.4 Relative Addressing ..... 6 - 23
6.5.5 Additional Modes ..... 6 - 23
6.6 Data Transfer and Manipulation ..... 6 - 27
6.6.1 Data Transfer Instructions ..... 6 - 27
6.6.2 Data Manipulation Instructions ..... 6 - 28
6.7 Program Control ..... 6 - 30
6.8 Reduced Instruction Set Computer (RISC) ..... 6 - 33
6.8.1 RISC Versus CISC ..... 6 - 33
6.8.2 RISC Properties ..... 6 - 35
6.8.3 RISC Addressing Modes ..... 6 - 36
6.8.4 CISC and RISC Application Areas ..... 6 - 37

Chapter - 7 Pipeline and Vector Processing (7 - 1) to (7 - 40)

7.1 Parallel Processing ..... 7 - 2
7.1.1 Flynn's Taxonomy ..... 7 - 2

7.2 Pipeline Processing ..... 7 - 5
7.2.1 Principles of Pipelining ..... 7 - 5
7.2.1.1 Asynchronous Pipeline Model ..... 7 - 7
7.2.1.2 Synchronous Pipeline Model ..... 7 - 8
7.2.2 Performance Parameters of Pipelining ..... 7 - 8
7.3 Arithmetic Pipeline ..... 7 - 13
7.4 Instruction Pipeline ..... 7 - 17
7.4.1 Pipeline Conflicts ..... 7 - 18
7.4.2 Techniques to Avoid Data Conflicts ..... 7 - 22
7.5 RISC Pipeline ..... 7 - 26
7.6 Vector Processing ..... 7 - 28
7.6.1 Characteristics of Vector Processing ..... 7 - 28
7.6.2 Vector Processing Approach ..... 7 - 29
7.6.3 Memory Interleaving ..... 7 - 34
7.6.4 Supercomputers ..... 7 - 35
7.6.4.1 Supercomputers Examples ..... 7 - 35
7.6.4.2 Practical Considerations in Supercomputer Design ..... 7 - 37
7.7 Array Processors ..... 7 - 38
7.7.1 SIMD Array Processor ..... 7 - 38
7.7.2 Attached Array Processor ..... 7 - 39

Chapter - 8 Computer Arithmetic (8 - 1) to (8 - 80)

8.1 Introduction ..... 8 - 2
8.2 Addition and Subtraction of Signed Numbers ..... 8 - 2
8.2.1 1's Complement Representation ..... 8 - 2
8.2.2 2's Complement Representation ..... 8 - 3
8.2.3 Subtraction of Binary Numbers using 2's Complement Method ..... 8 - 3
8.2.4 Adders ..... 8 - 5
8.2.4.1 Half Adders ..... 8 - 5
8.2.4.2 Full Adders ..... 8 - 6
8.2.5 Parallel Adder ..... 8 - 8
8.2.6 Parallel Subtractor ..... 8 - 9
8.2.7 Addition / Subtraction Logic Unit ..... 8 - 9

8.2.8 Overflow in Integer Arithmetic ..... 8 - 10
8.2.9 Addition and Subtraction of Signed-magnitude Data ..... 8 - 11
8.3 Multiplication of Positive Numbers ..... 8 - 14
8.4 Signed Operand Multiplication ..... 8 - 16
8.4.1 Booth's Algorithm ..... 8 - 17
8.4.2 Hardware Implementation of Booth's Algorithm ..... 8 - 21
8.4.3 Flowchart of Booth's Multiplication Algorithm ..... 8 - 22
8.5 Fast Multiplication ..... 8 - 25
8.5.1 Bit-Pair Recoding of Multipliers ..... 8 - 25
8.5.2 Array Multiplier ..... 8 - 29
8.6 Division Algorithms ..... 8 - 35
8.6.1 Restoring Division Algorithm ..... 8 - 36
8.6.2 Non-Restoring Division ..... 8 - 44
8.6.3 Comparison between Restoring and Non-Restoring Division Algorithm ..... 8 - 51
8.7 Floating Point Arithmetic Operations ..... 8 - 52
8.7.1 Addition and Subtraction ..... 8 - 52
8.7.1.1 Overflows and Underflows in Mantissa and Exponent ..... 8 - 54
8.7.1.2 Flowchart for Floating Point Addition and Subtraction ..... 8 - 54
8.7.1.3 Hardware Implementation of Floating Point Addition/Subtraction ..... 8 - 57
8.7.2 Multiplication and Division ..... 8 - 59
8.8 Decimal Arithmetic Unit ..... 8 - 62
8.8.1 BCD Number ..... 8 - 62
8.8.2 BCD Addition ..... 8 - 63
8.8.3 BCD (Decimal) Adder ..... 8 - 66
8.8.4 BCD Subtraction using 9's Complement Method ..... 8 - 68
8.8.5 BCD Subtraction using 10's Complement Method ..... 8 - 71
8.9 Decimal Arithmetic Operations ..... 8 - 74
8.9.1 BCD Multiplication ..... 8 - 74
8.9.2 Decimal Division ..... 8 - 75

Chapter - 9 Input - Output Organization (9 - 1) to (9 - 46)

9.1 Input/Output Interface ..... 9 - 2

9.2 I/O Bus and Interface Modules ..... 9 - 3
9.2.1 Connection of I/O Bus to I/O Devices ..... 9 - 5
9.2.2 I/O Versus Memory Bus and I/O Interfacing Techniques ..... 9 - 7
9.2.2.1 Memory Mapped I/O ..... 9 - 7
9.2.2.2 I/O Mapped I/O ..... 9 - 8
9.2.2.3 I/O Versus Memory Bus ..... 9 - 8
9.2.2.4 Isolated versus Memory-Mapped ..... 9 - 9
9.2.2.5 Example of I/O Interface ..... 9 - 10
9.3 Asynchronous Data Transfer ..... 9 - 12
9.3.1 Strobe Control ..... 9 - 12
9.3.2 Handshaking ..... 9 - 13
9.4 Asynchronous Serial Transfer ..... 9 - 15
9.4.1 Asynchronous Serial Communication ..... 9 - 16
9.5 Asynchronous Communication Interface ..... 9 - 19
9.5.1 Transmitter Operation ..... 9 - 20
9.5.2 Receiver Operation ..... 9 - 21
9.5.3 First - In First - Out (FIFO) Buffer ..... 9 - 21
9.6 Modes of Transfer ..... 9 - 23
9.6.1 Programmed I/O ..... 9 - 23
9.6.2 Interrupt-Initiated I/O ..... 9 - 24
9.6.2.1 Enabling and Disabling Interrupts ..... 9 - 25
9.6.2.2 Handling Multiple Devices ..... 9 - 26
9.6.3 Interrupt Priority ..... 9 - 28
9.6.3.1 Daisy-Chaining Priority ..... 9 - 28
9.6.3.2 Arrangement of Priority Groups ..... 9 - 28
9.6.3.3 Vectored Interrupts Scheme / Priority Interrupt Hardware ..... 9 - 29
9.6.3.4 Priority Encoder ..... 9 - 30
9.6.4 Direct Memory Access (DMA) ..... 9 - 31
9.6.4.1 DMA Operation ..... 9 - 32
9.6.4.2 Comparison of I/O Program Controlled Transfer and DMA Transfer ..... 9 - 33
9.6.4.3 DMA Block Diagram ..... 9 - 33
9.6.4.4 Data Transfer Modes ..... 9 - 34
9.6.4.5 Use of DMA in a Computer System ..... 9 - 38

9.6.5 I/O Processor (IOP) . . . . . 9 - 38
9.6.5.1 Features and Functions of IOP . . . . . 9 - 38
9.6.5.2 Block Diagram of IOP . . . . . 9 - 39
9.6.5.3 CPU and IOP Communication . . . . . 9 - 39
9.7 Serial Communication . . . . . 9 - 40
9.7.1 Classification . . . . . 9 - 41
9.7.1.1 Simplex . . . . . 9 - 41
9.7.1.2 Half Duplex . . . . . 9 - 41
9.7.1.3 Full Duplex. . . . . 9 - 41
9.7.2 Communication Protocols . . . . . 9 - 41
9.7.2.1 Standard Asynchronous Serial Communication Protocol . . . . . 9 - 41
9.7.2.2 BISYNC : Character/Byte Oriented Protocol . . . . . 9 - 41
9.7.2.3 HDLC and SDLC Protocols : Bit Oriented Protocols . . . . . 9 - 42
9.7.2.3 HDLC and SDLC Protocols : Bit Oriented Protocols . . . . . 9 - 43

Chapter - 10 Memory Organization (10 - 1) to (10 - 54)

10.1 Memory Hierarchy . . . . . 10 - 2
10.1.1 Multiprogramming . . . . . 10 - 4
10.2 Main Memory . . . . . 10 - 5
10.2.1 Semiconductor Read/Write Memories . . . . . 10 - 5
10.2.2 Static RAM . . . . . 10 - 6
10.2.2.1 Static RAM Cell . . . . . 10 - 6
10.2.2.2 CMOS Cell . . . . . 10 - 6
10.2.3 Dynamic RAMs . . . . . 10 - 7
10.2.4 Comparison between SRAM and DRAM . . . . . 10 - 8
10.2.5 Semiconductor Read Only Memories (ROM) . . . . . 10 - 8
10.2.5.1 PROM (Programmable Read Only Memory) . . . . . 10 - 9
10.2.5.2 EPROM (Erasable Programmable Read Only Memory). . . . . 10 - 9
10.2.5.3 EEPROM (Electrically Erasable Programmable Read Only Memory). . . . 10 - 10
10.2.5.4 Bootstrap Loader . . . . . 10 - 11
10.2.6 RAM and ROM Chips . . . . . 10 - 11
10.2.6.1 RAM Chips. . . . . 10 - 11
10.2.6.2 ROM Chips . . . . . 10 - 13
10.2.7 Memory Address Map . . . . . 10 - 13
10.2.8 Expanding Word Size and Memory Capacity . . . . . 10 - 15

10 - 20
10.3 Auxiliary Memory ..... 10 - 20
10.3.1 Magnetic Disk Memory ..... 10 - 20
10.3.1.1 Magnetic Surface Recording ..... 10 - 22
10.3.1.2 Data Organization and Formatting ..... 10 - 23
10.3.1.3 Characteristics ..... 10 - 24
10.3.1.4 Typical Disks ..... 10 - 24
10.3.1.5 Access Time ..... 10 - 24
10.3.2 Floppy Disk Memory ..... 10 - 25
10.3.2.1 Specifications of Floppy Disks ..... 10 - 25
10.3.2.2 Disk Format ..... 10 - 26
10.3.2.3 Storage Density ..... 10 - 28
10.3.3 Magnetic Tape ..... 10 - 30
10.4 Associative Memory ..... 10 - 30
10.4.1 Hardware Organization ..... 10 - 33
10.4.2 Read Operation ..... 10 - 33
10.4.3 Write Operation ..... 10 - 34
10.5 Cache Memories ..... 10 - 35
10.5.1 Mapping ..... 10 - 36
10.5.1.1 Direct-Mapping ..... 10 - 37
10.5.1.2 Associative-Mapping (Fully-Associative Mapping) ..... 10 - 38
10.5.1.3 Set-Associative Mapping ..... 10 - 43
10.5.1.4 Comparison between Mapping Techniques ..... 10 - 44
10.5.2 Cache Coherency ..... 10 - 45
10.5.3 Writing into Cache ..... 10 - 45
10.5.3.1 Write through System ..... 10 - 46
10.5.3.2 Buffered Write through System ..... 10 - 46
10.5.3.3 Write-Back System ..... 10 - 46
10.5.3.4 Replacement Algorithms ..... 10 - 46
10.6 Virtual Memory ..... 10 - 48
10.6.1 Address Mapping using Pages ..... 10 - 49
10.6.2 Page Replacement ..... 10 - 50
10.6.3 Page Replacement Algorithms ..... 10 - 50

Chapter - 11
11.1 Charac
11.1.1 C
11.
11.
11.1.2 L
11.2 Interco
11.2.1 T
11.2.2 C
11.2.3 M
11.2.4 M
11.2.5 H
11.3 Interp
11.3.1 S
11.3.2 A
11.3.3 E
11.3.4 S
11.3.5 E
11.3.6 E
11.4 Interp
11.4.1 I
11.4.2 I
11.4.3 I
11.5 Cache
11.6 Share
11.6.1 (
11.6.2 I
Computer Or

Chapter - 11 Multiprocessors (11 - 1) to (11 - 36)

11.1 Characteristics of Multiprocessors ..... 11 - 2
11.1.1 Closely Coupled Multiprocessors ..... 11 - 3
11.1.1.1 CCS without Private Cache ..... 11 - 4
11.1.1.2 CCS with Private Cache ..... 11 - 5
11.1.2 Loosely Coupled Multiprocessors ..... 11 - 5
11.2 Interconnection Structures ..... 11 - 8
11.2.1 Time Shared Bus or Common Bus ..... 11 - 9
11.2.2 Cross Bar Switch ..... 11 - 10
11.2.3 Multiport Memory ..... 11 - 12
11.2.4 Multistage Switching (Binary Switching) Networks ..... 11 - 13
11.2.5 Hypercube Interconnection ..... 11 - 15
11.3 Interprocessor Arbitration ..... 11 - 17
11.3.1 Synchronous Bus ..... 11 - 18
11.3.2 Asynchronous Bus ..... 11 - 19
11.3.3 Bus Arbitration ..... 11 - 21
11.3.4 Serial Arbitration ..... 11 - 21
11.3.5 Parallel Arbitration ..... 11 - 22
11.3.6 Dynamic Arbitration ..... 11 - 23
11.4 Interprocessor Communication and Synchronization ..... 11 - 24
11.4.1 IPC using Shared Variables ..... 11 - 25
11.4.2 IPC using Message Passing ..... 11 - 25
11.4.3 Interprocessor Synchronization ..... 11 - 26
11.5 Cache Coherence ..... 11 - 29
11.6 Shared Memory Multiprocessors ..... 11 - 32
11.6.1 Centralized Shared Memory Architectures ..... 11 - 33
11.6.2 Distributed-Shared Memory Architectures ..... 11 - 34

1 Computer Data Representation

Syllabus

Basic computer data types. Complements. Fixed point representation. Register transfer and micro-operations. Floating point representation.

Contents

1.1 Basic Computer Data Types

1.2 Fixed Point Numbers

1.3 Floating Point Representation

1.1 Basic Computer Data Types

- Based on the number system two basic data types are implemented in the computer system : **fixed point numbers** and **floating point numbers**. Representing numbers in such data types is commonly known as **fixed point representation** and **floating point representation**.
- In integer numbers, radix point is fixed and assumed to be to the right of the right most digit. As radix point is fixed, the number system is referred to as **fixed point number system**.
- With fixed point number system we can represent positive or negative integer numbers.
- However, **floating point number system** allows the representation of numbers having both **integer part** and **fractional part**.

Review Questions

1. *State two basic data types implemented in the computer system.*
2. *Write a note on data types*

1.2 Fixed Point Numbers

- The fixed point integer numbers are represented in two forms : **signed integer** and **unsigned integer**.
- Unsigned integer numbers represent positive numbers.
- To represent negative numbers various techniques are used because computer does not have provision to represent negative sign. Techniques to represent signed integer numbers are :
  - Sign-magnitude representation
  - 1's complement
  - 2's complement.

1.2.1 Signed Magnitude Representation

- The leftmost bit (sign bit) in the binary number represents sign of the number.
- Fig. 1.2.1 shows the sign magnitude format for 8-bit signed number.
- Here the most significant bit (MSB) represents sign of the number. If MSB is 1, number is negative and if MSB is 0, number is positive. The remaining bits represent magnitude of the number.

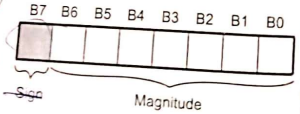


Fig. 1.2.1

- Here are some examples of sign-magnitude numbers.
  1. + 6 = 0000 0110
  2. − 14 = 1000 1110
  3. + 24 = 0001 1000
  4. − 64 = 1100 0000
- In case of unsigned 8-bit binary numbers the decimal range is 0 to 255. For signed magnitude 8-bit binary numbers the largest magnitude is reduced from 255 to 127 because we need to represent both positive and negative numbers.

Maximum positive number :	0	1	1	1	1	1	1	1	= +127
Maximum negative number :	1	1	1	1	1	1	1	1	= −127

1.2.2 1's Complement Representation

The 1's complement of a binary number is the number that results when we change all 1's to zeros and the zeros to ones.

Example 1.2.1 *Find 1's complement of* $(11010100)_2$.

Solution :

1	1	0	1	0	1	0	0	Number
↓	↓	↓	↓	↓	↓	↓	↓	NOT operation
0	0	1	0	1	0	1	1	1's complement of number

1.2.3 2's Complement Representation

- The 2's complement is the binary number that results when we add 1 to the 1's complement. It is given as

  2's complement = 1's complement + 1
- The 2's complement form is used to represent negative numbers.

Example 1.2.2 *Find 2's complement of* $(11000100)_2$.

Solution :

1	1	0	0	0	1	0	0	Number
					1	1		Carry
0	0	1	1	1	0	1	1	1's complement of number
+							1	Add 1
0	0	1	1	1	1	0	0	2's complement of number

- Table 1.2.1 shows representation of 4-bit integer numbers in sign-magnitude, complement and 2's complement form.

Decimal	Representations		
	Sign-magnitude	1's complement	2's complement
+ 7	0 1 1 1	0 1 1 1	0 1 1 1
+ 6	0 1 1 0	0 1 1 0	0 1 1 0
+ 5	0 1 0 1	0 1 0 1	0 1 0 1
+ 4	0 1 0 0	0 1 0 0	0 1 0 0
+ 3	0 0 1 1	0 0 1 1	0 0 1 1
+ 2	0 0 1 0	0 0 1 0	0 0 1 0
+ 1	0 0 0 1	0 0 0 1	0 0 0 1
+ 0	0 0 0 0	0 0 0 0	0 0 0 0
– 0	1 0 0 0	1 1 1 1	–
– 1	1 0 0 1	1 1 1 0	1 1 1 1
– 2	1 0 1 0	1 1 0 1	1 1 1 0
– 3	1 0 1 1	1 1 0 0	1 1 0 1
– 4	1 1 0 0	1 0 1 1	1 1 0 0
– 5	1 1 0 1	1 0 1 0	1 0 1 1
– 6	1 1 1 0	1 0 0 1	1 0 1 0
– 7	1 1 1 1	1 0 0 0	1 0 0 1
– 8	–	–	1 0 0 0

Table 1.2.1 Representation of 4-bit integer numbers in different forms

Advantages of 2's complement representation

- From Table 1.2.1 we can note that there are distinct + 0 and – 0 representations both the sign-magnitude and 1's complement systems, but the 2's compleme system has only a + 0 representation.
- For 4-bit numbers, the value – 8 is representable only in the 2's compleme system and not in other systems.
- It is more efficient for logic circuit implementation, and one often used computers, for addition and subtraction operations.

1.2.4 Sign Extension

- When we add two binary numbers the result may extend by 1-bit i.e. the result binary number may need one more bit if there is a carry after addition of mo significant bit. To represent such result in correct format we have to allocate t

additional bit for representing the magnitude of the numbers which are to be added. This is illustrated in Fig. 1.2.2.

0	0	1	0	0	0	(+ 8)
0	0	1	0	0	1	(+ 9)
0	1	0	0	0	1	(+ 17)

Fig. 1.2.2

- In case of positive binary number we extend the bit by appending bit-0 (sign-bit for positive number) to the left of the most-significant bit of the number. In case of negative binary number can extend the bit by appending bit-1 (sign-bit for negative number) to the left of the most-significant bit of the number. This is illustrated in the Fig. 1.2.3.

1. Original number represented using 6 bits : 0 1 0 1 0 0 (+ 20) — Sign-bit

   Same number represented using 7 bits : 0 0 1 0 1 0 0 (+ 20) — Sign-extension

   Same number represented using 8 bits : 0 0 0 1 0 1 0 0 (+ 20) — Sign-extension

2. Original number represented using 6 bits in 2's complement form : 1 0 1 1 0 0 (– 20) — Sign-bit

   Same number represented using 7 bits in 2's complement form : 1 1 0 1 1 0 0 (– 20) — Sign-extension

   Same number represented using 8 bits in 2's complement form : 1 1 1 0 1 1 0 0 (– 20) — Sign-extension

Fig. 1.2.3

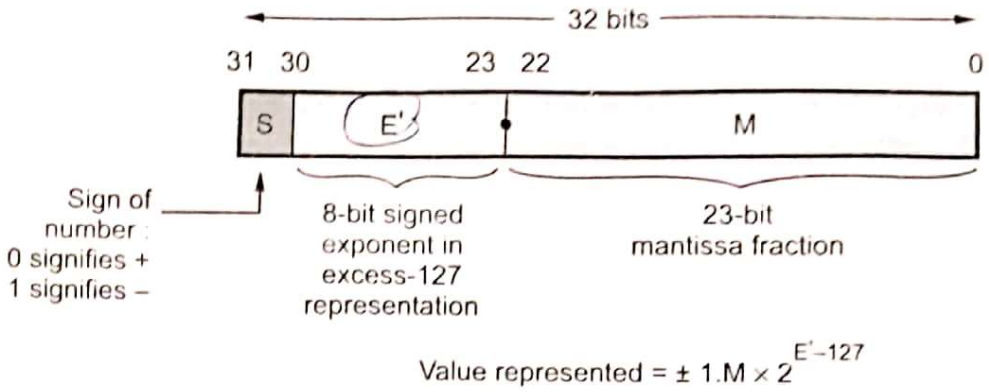
- In both the case sign-bit of the original number is extended and hence such extension is known as sign-extension of the number.

Review Questions

1. *Write a note on fixed point number representation.*
2. *Find 1's complement of* $(10101100)_2$ **[Ans. : $(01010011)_2$]**
3. *Find 2's complement of* $(01011011)_2$ **[Ans. : $(10100101)_2$]**

(field 2) Exponent ← 8 - bits

(field 3) Mantissa ← 23 - bits



$$\text{Value represented} = \pm 1.M \times 2^{E'-127}$$

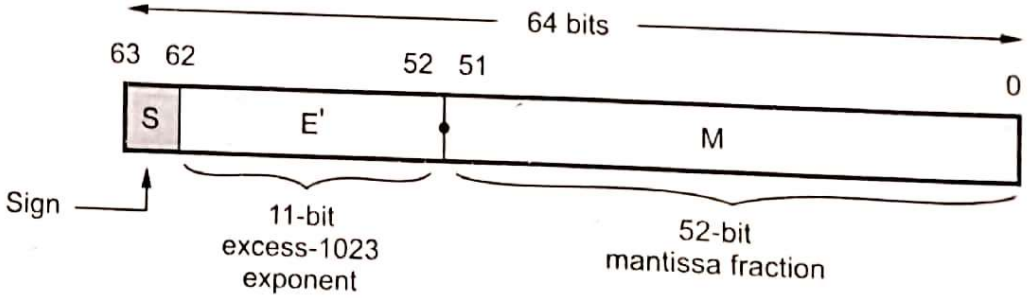
Fig. 1.3.2 (a) Single-precision

- Instead of the signed exponent, E, the value actually stored in the exponent field is E′ = E (scaling factor) + bias.
- In the 32-bit floating point system (single precision), bias is 127. Hence E′ = E (scaling factor) + 127. This representation of exponent is also called as the *excess-127 format*.
- The end values of E′, namely, 0 and 255, are used to indicate the floating point values of exact zero and infinity, respectively in single precision.
- Thus range of E′ for normal values in single precision is $0 < E' < 255$. This means that for 32-bit representation the actual exponent E is in the range $-126 \leq E \leq 127$.
- The 64-bit standard representation shown in Fig. 1.3.2 (b) is called a **double-precision representation** because it occupies two 32-bit words. The 64-bits are divided into three fields as shown below :

(field 1) Sign ← 1 - bit

(field 2) Exponent ← 11 - bit

(field 3) Mantissa ← 52 - bits



$$\text{Value represented} = \pm 1.M \times 2^{E'-1023}$$

Fig. 1.3.2 (b) Double precision

- In the double precision format value actually stored in the exponent field is given as

$$E' = E + 1023$$

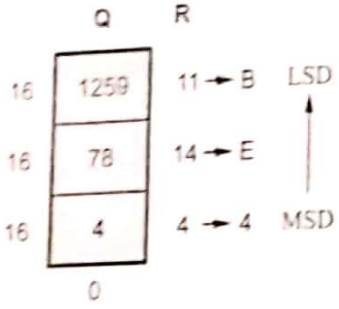
- Here, bias value is 1023 and hence it is also called **excess-1023 format.**
- The end values of E', namely, 0 and 2047, are used to indicate the floating point exact values of exact zero and infinity, respectively.
- Thus the range of E' for normal values in double precision is $0 < E' < 2047$. This means that for 64-bit representation the actual exponent E is in the range

$$-1022 \leq E \leq 1023.$$

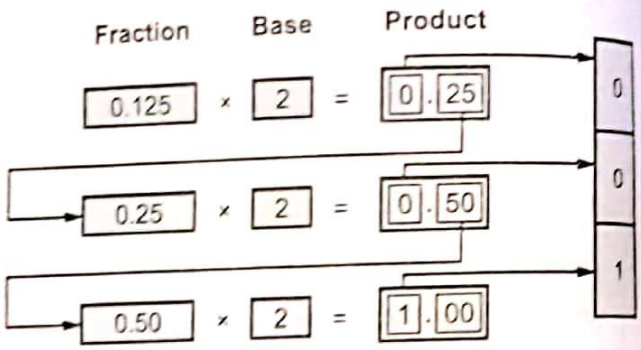
Example 1.3.1 *Represent 1259.125_{10} in single precision and double precision formats.*

Solution : **Step 1 : Convert decimal number in binary format**

Integer part :



Fractional part :



$\therefore (1259)_{10} = (4EB)_{16}$

4				E				B			
0	1	0	0	1	1	1	0	1	0	1	1

Hex number

Binary number

$\therefore \quad (1259)_{10} = (100\ 1110\ 1011)_2$

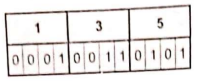
$\therefore \quad (0.125)_{10} = (0.001)_2$

$\therefore$ Binary number = 1 0 0 1 1 1 0 1 0 1 1 + 0. 0 0 1 = 1, 0 0 1 1 1 0 1 0 1 1 . 0 0 1

Step 2 : Normalize the number

$$1\ 0\ 0\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ 1.\ 0\ 0\ 1 = 1.0\ 0\ 1\ 1\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ 0\ 1 \times 2^{10}$$

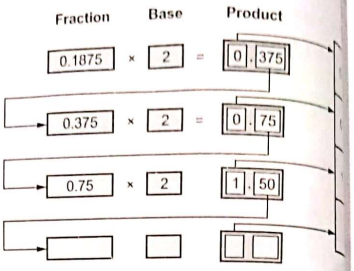
Integer part :



$\therefore \quad (309)_{10} = (135)_{16}$

$\therefore \quad (309)_{10} = (100110101)_2$

Fractional part :



$\therefore \quad (0.1875)_{10} = (0.0011)_2$

$\therefore$ Binary number = – 100110101 + .0011

= – 100110101 . 0011

Step 2 : Normalize the number

$$-100110101.0011 = -1.00110101001 1 \times 2^8$$

Step 3 : Single precision representation

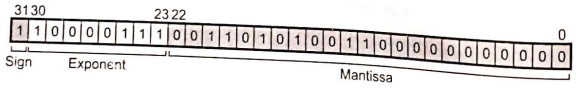
For a given number

S = 1, E = 8, and M = 0011 0101 0011

Bias for single precision format = 127

$\therefore \quad E' = E + 127 = 8 + 127 = 135_{10} = 10000111_2$

$\therefore$ Number in single precision format is given as



Step 4 : Double precision representation

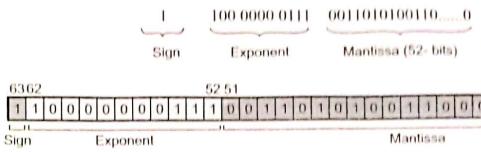
For a given number

S = 1, E = 8 and M = 0011 0101 0011

Bias for double precision format = 1023

$\therefore \quad E' = E + 1023 = 8 + 1023 = 1031_{10} = 10000000111_2$

$\therefore$ Number in double precision format is given as

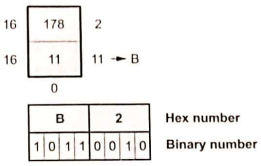


Example 1.3.3 *Represent $(178.1875)_{10}$ in single and double precision floating point format.*

PU : May-11, Marks 8

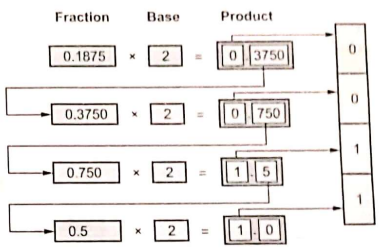
Solution : Step 1 : Convert decimal number in binary format

Integer part :



$\therefore \quad (178)_{10} = (B2)_{16} = (10110010)_2$

Fractional part :



= 0.0011

$\therefore$ Binary number = 10110010 + 0.0011 = 10110010.0011

Step 2 : Normalize the number

1011 0010.0011 = 1.01100100011×2^7

Step 3 : Single precision representation

S = 0, E = 7 and M = 011 001 000 11

Bias = 127

$\therefore \quad E' = E + 127 = 7 + 127 = 134_{10} = 10000110_2$

Number in single precision format is given as

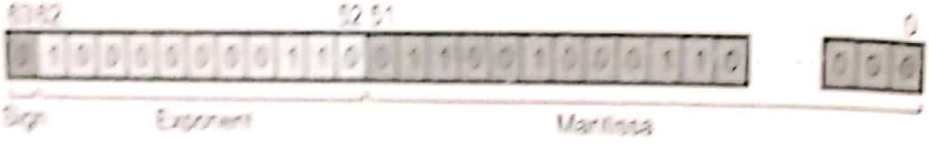


Step 4 : Double precision representation

$$\text{Bias} = 1023$$

$$E' = E + 1023 = 7 + 1023 = 1030_{10} = 1000\ 0000\ 110_2$$

Number in double precision format is given as



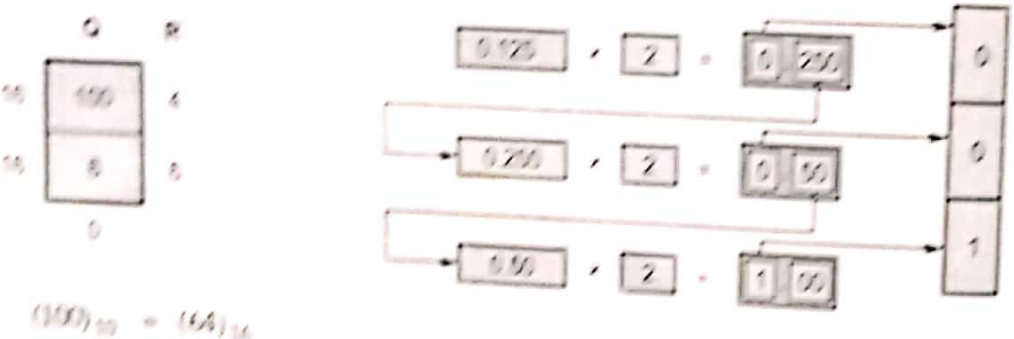
Example 1.3.4 *Represent the following numbers in single precision floating point format.*

a) 100.125 b) 42.625

MU : May 06

Solution :

a) Step 1 : Decimal to Binary conversion



$$\therefore \quad (100)_{10} = (64)_{16}$$

$$= (01100100)_2$$

$$\therefore \quad (100.125)_{10} = (1100100.001)_2$$

Step 2 : Normalization

$$1100100.001 = 1.100100001 \times 2^6$$

Step 3 : Single precision representation

$$S = 0, \quad E = 6 \text{ and } M = 10010000\ 1$$

$$E' = 127 + 6 = (133)_{10} = (10000101)_2$$

Step 2 : Normalize the number

$$10100.1111.001111 = 1.0100111100111111 \times 2^{8}$$

Step 3 : Single precision format

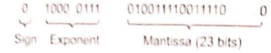
For a given number

S = 0, E = 8, M = 0100 1111 001111

Bias for single precision format = 127

$$E = E + 127 = 8 + 127 = (135)_{10} = (1000\ 0111)_2$$

Number in single precision format is given as

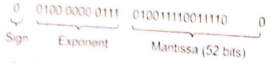


Step 4 : Double precision format

Bias for double precision format = 1023

$$E = E + 1023 = 8 + 1023 = (1031)_{10} = (0100\ 0000\ 0111)_2$$

Number in double precision format is given as



Example 1.3.6 *Represent the following in single precision floating point format as well as double precision format.*

i) 17.125 ii) 12.5

Solution : i) 17.125 **PU : Dec.-08, May-12**

Step 1 : Decimal to Binary conversion

$$\therefore \quad (17)_{10} = (10001)_2$$

$$\therefore \quad (17.125)_{10} = (10001.001)_2$$

$$0.125 \times 2 = 0.25 \quad 0$$

$$0.25 \times 2 = 0.5 \quad 0$$

$$0.5 \times 2 = 1.0 \quad 1$$

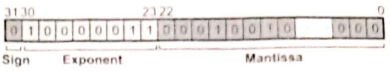
Step 2 : Normalization

$$10001.001 = 1.0001001 \times 2^{4}$$

Step 3 : Single precision representation

$$S = 0, \quad E = 4 \quad \text{and} \quad M = 0001001$$

$$E' = 127 + 4 = (131)_{10} = (10000011)_2$$

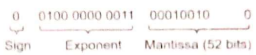


Step 4 : Double precision format

Bias for double precision format = 1023

$$\therefore \quad E' = E + 1023 = 4 + 1023 = (1027)_{10} = (0100\ 0000\ 0011)_2$$

∴ Number in double precision format is given as



ii) 12.5

Step 1 : Decimal to Binary conversion

$$\therefore \quad (12)_{10} = (1100)_2 \qquad 0.5 \times 2 = 1.0 = 1$$

$$\therefore \quad 12.5 = 1100.1$$

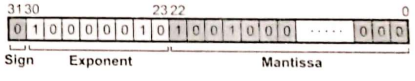
Step 2 : Normalization

$$1100.1 = 1.1001 \times 2^{3}$$

Step 3 : Single precision representation

$$S = 0, \quad E = 3 \quad \text{and} \quad M = 1001$$

$$E' = 127 + 3 = (130)_{10} = (10000010)_2$$



Step 4 : Double precision format

Bias for double precision format = 1023

$$\therefore \quad E' = E + 1023 = 3 + 1023 = (1026)_{10} = (0100\ 0000\ 0010)_2$$

∴ Number in double precision format is given as



Example 1.3.7 *Represent* $-(127.1075)_{10}$ *in single precision format.* **PU : Dec.-07**

Solution : Step 1 : Change the decimal number into binary format

Integer part :

	Q	R
16	127	5
16	7	7
	0	

LSD ↑ MSD

$(127)_{10} = (57)_{16}$

$= (01010111)_2$

Fractional part :

$0.1075 \times 2 = 0.2150$ 0 MSD

$0.215 \times 2 = 0.53$ 0

$0.53 \times 2 = 1.06$ 1

$0.06 \times 2 = 0.12$ 0 LSD

$\therefore \quad (0.1075)_{10} = (0.0010)_2$

$\therefore \quad (127.1075)_{10} = (01010111.0010)_2$

Step 2 : Normalize the number

$01010111.0010 = 1.0101110010 \times 2^6$

Step 3 : Single precision format

For the given number

S = 1, E = 6, M = 0101110010

Bias for single precision format = 127

$\therefore \quad E' = E + 127 = 6 + 127 = (133)_{10} = (100\ 00101)_2$

$\therefore$ Number in single precision format is given as

1 | 10000101 | 0101110010 . . . 0

Sign | Exponent | Mantissa(23 bits)

Example 1.3.8 *Represent the following number in single precision floating point format : i) 101.25 ii) 41.625.*

Solution : i) 101.25

Step 1 : Convert decimal number into binary format

Integer part :

	Q	R
16	101	5
16	6	6
	0	

LSD ↑ MSD

$\therefore (101)_{10} = (65)_{16}$

$= (0110\ 0101)_2$

Fractional part :

$0.25 \times 2 = 0.5$ 0 MSD

$0.5 \times 2 = 1.0$ 1 LSD

$\therefore \quad (0.25)_{10} = (0.01)_2$

$\therefore \quad (101.25)_{10} = (0110\ 0101.01)_2$

Step 2 : Normalize the number

$0110\ 0101.01 = 1.10010101 \times 2^6$

Step 3 : Single precision format

For the given number

S = 0, E = 6, M = 10010101

Bias for single precision format = 127

$\therefore \quad E' = E + 127 = 6 + 127 = (133)_{10} = (10000101)_2$

$\therefore$ Number in single precision format is given as

0 | 10000101 | 10010101 0

Sign | Exponent | Mantissa(23 bits)

ii) 41.625

Step 1 : Convert decimal number into binary format

Integer part :

	Q	R
16	41	9
16	2	2
	0	

LSD ↑ MSD

$\therefore (41)_{10} = (29)_{16}$

$= (0010\ 1001)_2$

Fractional part :

$0.625 \times 2 = 1.250$ 1 MSD

$0.25 \times 2 = 0.5$ 0

$0.5 \times 2 = 1.0$ 1 LSD

$\therefore \quad (0.625)_{10} = (0.1010)_2$

$\therefore \quad (41.625)_{10} = (0010\ 1001.1010)_2$

Step 2 : Normalize the number

$00101001.1010 = 1.010011010 \times 2^5$

Step 3 : Single precision format

For the given number

S = 0, E = 5, M = 010011010

Bias for single precision format = 127

$\therefore \quad E' = E + 127 = 5 + 127 = (132)_{10} = (1000\ 0100)_2$

$\therefore$ Number in single precision format is given as

0 | 1000 0100 | 010011010 . . . 0

Sign | Exponent | Mantissa(23 bits)

Example 1.3.9 *Represent following number in single precision format : $(0.625)_{10}$.*

PU : May-13

Solution : Step 1 : Convert the decimal number into binary format

Fractional part :

		MSD
0.625 × 2 = 1.25	1	↓
0.25 × 2 = 0.5	0	
0.5 × 2 = 1.0	1	LSD

$\therefore \quad (0.625)_{10} = (0.101)_2$

Step 2 : Normalize the number

$$0.1010 = 1.010 \times 2^{-1}$$

Step 3 : Single precision format

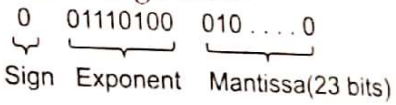
For the given number

S = 0, E = –1, M = 01

Bias for single precision format = 127

$$E' = E + 127 = -1 + 127 = (126)_{10} = (0111\ 0100)_2$$

∴ Number in single precision format is given as

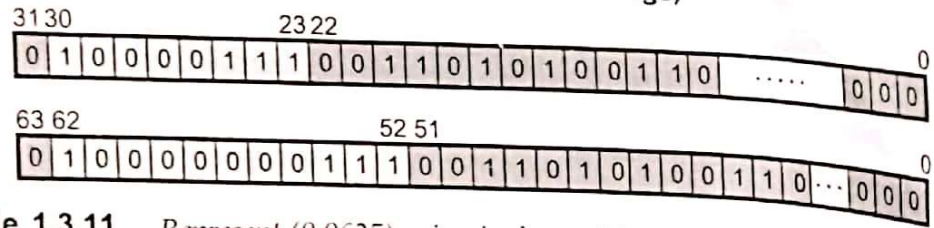


Examples for Practice

Example 1.3.10 *Represent $(309.1875)_{10}$ in single precision and double precision format.*

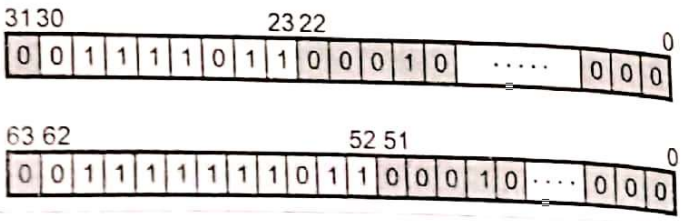
PU : Dec.-10

Ans : Refer example 1.3.2 (only sign bit will change)



Example 1.3.11 *Represent $(0.0625)_{10}$ in single precision and double precision formats.*

Ans :



2 Register Transfer and Micro-operations

Syllabus

Register transfer language, Register transfer, Bus and memory transfers (Tree-state bus buffers, Memory transfer), Arithmetic micro-operations, Logic micro-operations, Shift micro-operations, Arithmetic logical shift unit

Contents

2.1 Register Transfer Language . . . . . **June-05,06,13, Dec.-09,13 Jan.-09,10, May-12** Marks 7
2.2 Register Transfer . . . . . **June-05, Dec.-13** Marks 4
2.3 Bus and Memory Transfers . . . . . **Jan.-09,10, Dec.-11, Winter-12, May-12,14, June-05,06** Marks 8
2.4 Arithmetic Micro-operations . . . . . **June-06,13, Winter-12 Dec.-09, Feb.-10 May-12** Marks 8
2.5 Logic Micro-operations . . . . . **June-06, Dec.-09,12** Marks 7
2.6 Shift Micro-operations . . . . . **Dec.-11, June-06** Marks 7
2.7 Arithmetic Logic Shift Unit

2.1 Register Transfer Language

GTU : June-05,06,13, Dec.-09,13, Jan.-09,10, May-12

The digital system can be constructed with flip-flops and gates using sequential log
design. However, it is difficult to design large digital systems using this approach. T
overcome this difficulty, large digital systems are designed using modular approach. I
the modular approach, system is partitioned into subsystems such that each subsyste
performs some functional task. Such modules are constructed from registers, counter
decoders, multiplexers, arithmetic and logic elements and control logic. The modules a
then interconnected with common data and control paths to form a digital system.

The modules and interconnections between them cannot be described by means
combinational or sequential logic techniques. These techniques are suitable at the ga
level and the flip-flop level digital systems. At function level we use register transf
logic to describe the modules and interconnections between them.

2.1.1 Features

Let us see the features of register transfer logic.

1. Uses registers as a primitive component in the digital system instead of flip-flop
and gates.
2. Describes the information flow and processing tasks among the data stored in th
registers in a concise and precise manner.
3. Uses a set of expressions and statements which resemble the statements used
programming languages.
4. The presentation of digital functions in register transfer logic is very user friendly

2.1.2 Basic Components of Register Transfer Logic

The register transfer logic method uses four basic components to describe digit
systems. These are as follows :

1. **Registers and their functions :** The total number of registers in the system an
their functions. The register includes all its counterparts such as shift register
counters and memory unit. Counter is register which increments the informatio
stored in it by 1. A memory unit is a set of storage registers where information ca
be stored.
2. **Information :** The information is stored in the registers. The information may b
binary numbers, BCD numbers, alphanumeric characters, control information
any other binary coded information.
3. **Operations :** The operations performed on the information stored in the register
The operations performed on the data stored in registers are calle

micro-operations and it is performed in one clock cycle. The operation may be arithmetic operation or logical operation. A statement that requires a sequence of micro-operations for its implementation is called a **macro-operation**.

4. **Control function :** The control functions that activate operations and control the sequence of operations. They consists of timing signals that sequence the operations one at a time. The control function is basically a binary variable, when it is logic 1 it initiates the operation; otherwise it inhibits the operation.

2.1.3 Definition of Register Transfer Language

A micro-operation is an elementary operation performed on the information stored in one or more registers. The result of micro-operation may be stored in source register or it may be stored in the another register. Examples of micro-operation are load, store, clear, shift, count and so on. A sequence of micro-operations are performed to perform one complete operation. For example, to add two numbers following micro-operation sequence has to be performed.

1. Load first number in register 1.
2. Load second number in register 2.
3. Perform add micro-operation.
4. Store the result in the destination register.

As shown above it is possible to specify the sequence of micro-operation in a words, but it involves lengthy descriptive explanation. Hence, it is preferred to use symbolic notations to describe the micro-operations. These symbolic notations are also called a **register transfer language** or **computer hardware description language**.

The micro-operations used in the digital system can be classified as

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

Review Questions

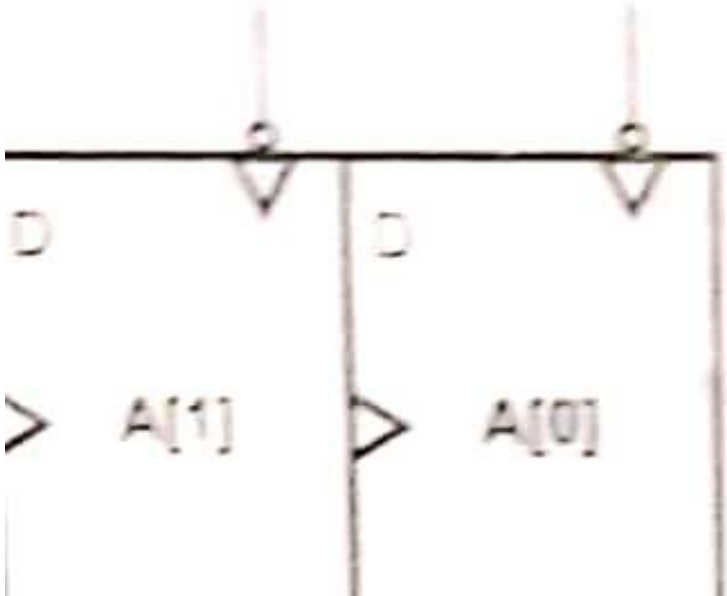
1. *List the features of register transfer logic.*
2. *State and explain the basic components of register transfer logic.*
3. *What is micro-operation ?* GTU : June-06, June-13, Marks 7
4. *What is register transfer language ?* GTU : June-05, 06, Summer-13, Winter-13, Dec.-09, Jan.-09, 10, Marks 4
5. *Explain the register transfer language with example.* GTU : May-12, Marks 7

with the colon at the left-most

oolean variable having value
R_1 are transferred to R_2 only

ce
er

his is just an implementation
; that if T = 1 then $R_2 \leftarrow R_1$
with a colon.



It is important to note that the clock is not included as a variable in the register transfer statements. It is assumed that all transfers occur during a clock edge transition either positive or negative. If hardware permits two micro-operations can be executed at the same time. For example, statement exchanges the contents of two registers, they perform $R_2 \leftarrow R_1$ and $R_1 \leftarrow R_2$ micro-operations simultaneously they are separated by comma in statement.

$T: R_2 \leftarrow R_1,\ R_3 \leftarrow R_4$

Micro-operation 1 | Micro-operation 2

Consider following two statements.

$T_1: R_1 \leftarrow A$

$T_2: R_1 \leftarrow B$

The first statement transfers the contents of register A to register R_1 at timing T_1. At timing T_2, the contents of register B are transferred to register R_1. This is stated in statement 2. The two statements have the same destination register, but have different source registers and timing variable. The Fig. 2.2.3 shows the block diagram for implementing above two statements. As shown in the Fig. 2.2.3, each register has 4-bits and quadruple 2 to 1 multiplexer is used to select the source register either A or B. When $T_1 = 1$, register A is selected and when $T_2 = 1$, register B is selected.

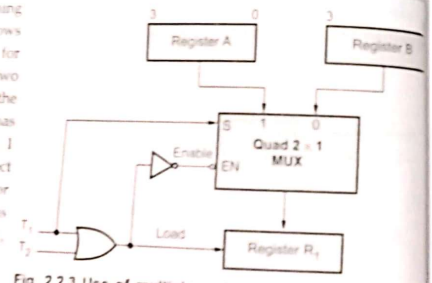


Fig. 2.2.3 Use of multiplexer to transfer information from two source registers to one destination register

Example 2.2.1 *Represent the following conditional control statement by two register transfer statements with control functions*

If (P = 1) then ($R_1 \leftarrow R_2$) else

If (Q = 1) then ($R_1 \leftarrow R_3$).

Solution : P $R_1 \leftarrow R_2$

P Q $R_1 \leftarrow R_3$

Example 2.2.2 *Show the block diagram that executes the statement*

$qT_3 : A \leftarrow B,\ B \leftarrow A$

Solution :

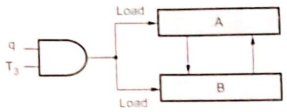


Fig. 2.2.4

Example 2.2.3 *Show the hardware implementation for the following statements. The registers are 4-bit in length.*

$T_0 : A \leftarrow R_0 \quad T_1 : A \leftarrow R_1 \quad T_2 : A \leftarrow R_2 \quad T_3 : A \leftarrow R_3$

Solution : The given statements can be implemented by two methos : Using multiplexer and using enable signals of registers.

a) Using multiplexer

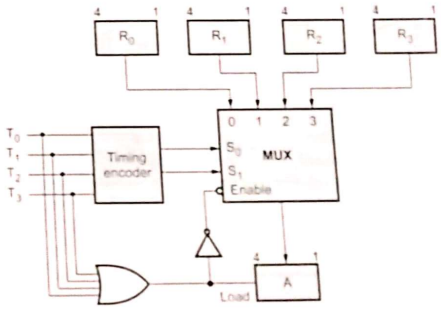
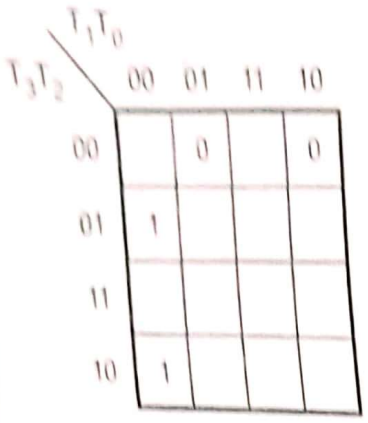


Fig. 2.2.5

The excitation table for timing encoder is as follows

T_3	T_2	T_1	T_0	S_1	S_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

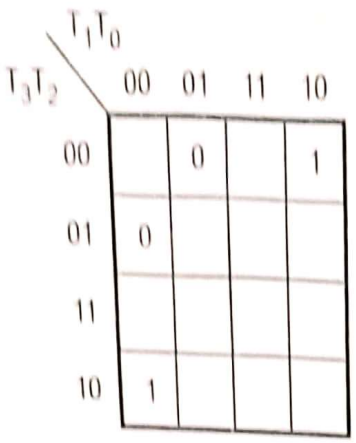
Table 2.2.1



(a) $S_1 = \bar{T}_3 T_2 \bar{T}_1 \bar{T}_0 + T_3 \bar{T}_2 \bar{T}_1 \bar{T}_0$

$= (T_2 \oplus T_3) \bar{T}_1 \bar{T}_0$

$= (T_2 \oplus T_3) \overline{(T_0 + T_1)}$



(b) $S_0 = \bar{T}_3 \bar{T}_2 T_1 \bar{T}_0 + T_3 \bar{T}_2 \bar{T}_1 \bar{T}_0$

$= (T_3 \oplus T_1) \bar{T}_2 \bar{T}_0$

$= (T_3 \oplus T_1) \overline{(T_0 + T_2)}$

Fig. 2.2.6

Therefore, hardware implementation for timing encoder is as follows :

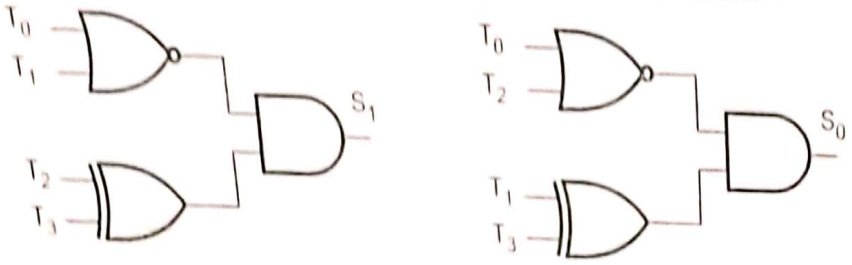


Fig. 2.2.7

b) Using enable signals of registers :

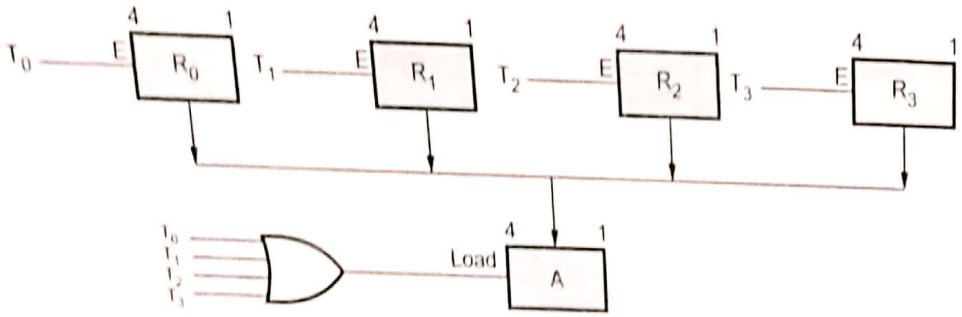


Fig. 2.2.8

The Table 2.2.2 summarizes the basic symbols used for register transfers.

Symbol	Description	Examples
Letters or letters with numerals	Denotes a register.	MAR, R_2, PC
Parentheses ( )	Denotes a part of register.	R_1 (7 - 0) : bits 7 - 0 of register R_1 R_1(H) : most significant half of register R_1

is connected to input 0, bit 0 of register 1 is connected to input 1, bit 0 of register 2 is connected to input 2 and bit 0 of register 3 is connected to input 3. Similarly, inputs for

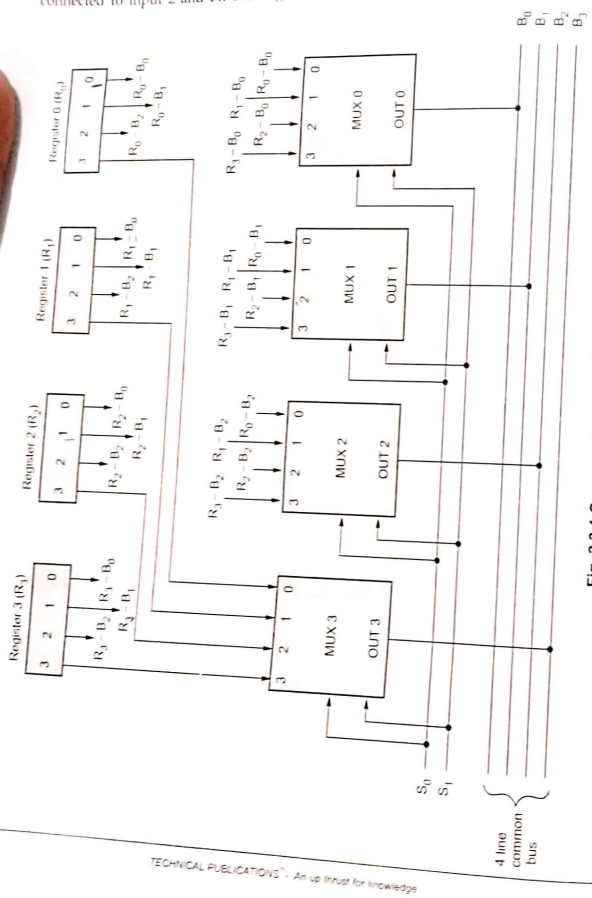


Fig. 2.3.1 Common bus system using multiplexers

MUX 1 are connected to bit 1 outputs, inputs for MUX 2 are connected to bit 2, and inputs for MUX 3 are connected to bit 3 outputs of registers 0 through 3. To avoid the complexity of the diagram, only input connections for MUX 3 are physically shown. To show other connections labels are used. Two same labels have connection between them. For example, bit-2 ($R_0 - B_2$) output of register 0 is connected to the input 0 of MUX 2.

The two selection lines S_1 and S_0 are connected to the selection inputs of all four multiplexers. These lines choose the four bits of one register and transfer them into the four-line common bus through OUT lines. When S_1 S_0 = 00, the input 0 of all four multiplexers are selected and applied to the outputs to transfer them on the common bus. As a result a common bus receives the contents of register 0. Since the outputs of register 0 are connected to the input 0 of the multiplexers. Similarly, the content of register 1 are transferred on the common bus when S_1 S_0 = 01. The Table 2.3.1 shows the register selection according to the status of S_1 S_0 lines.

S_1	S_0	Selected register
0	0	Register 0
0	1	Register 1
1	0	Register 2
1	1	Register 3

Table 2.3.1 Selection table

In general, common bus system will have n multiplexers and n-line common bus where n represents the number of bits in each register and k number of inputs to each multiplexer where k represents the total number of registers. Therefore, the size of each multiplexer will be k × 1.

The data transfer from a bus to any destination register can be accomplished by connecting the bus lines to the inputs of all destination registers and activating the load control input of the particular (selected) destination register.

In a register transfer notations we can specify the transfer of data between two registers considering a bus is as given below.

$$BUS \leftarrow R_1,\ R_2 \leftarrow BUS$$

Here, content of register R_1 is placed on the bus and then the content of the bus is loaded into the register R_2 by activating its load control input. Normally, it is known that bus exist in the system and hence for convenience, most of the times we prefer direct method of representing data transfer between registers. For example, above data transfer is specified as,

$$R_2 \leftarrow R_1$$

Example 2.3.1 *A digital computer has a common bus system for 16 registers of 32 bits each.*

i) How many selection input are there in each multiplexer ?

ii) What size of multiplexers are needed ?

iii) How many multiplexers are there in a bus ?

GTU : Jan.-09, 10, Marks 4

Solution :

i) Number of inputs to each multiplexer is the total number of registers, i.e. 16. W 16 inputs multiplexer must have 4 selection inputs.

ii) Size of the multiplexer will be 16 : 1.

iii) Total number of bits in each register gives the number of multiplexers to be us Thus there are 32 multiplexers in a bus.

Implementing common bus using tri-state buffer -

A common bus system can be implemented using **tri-state** or **three-state gates** inste of multiplexers. A tri-state gate is a digital circuit that can have three output state HIGH, LOW and high impedance. The high impedance state behaves like an op circuit, which means that the output is disconnected.

We know that bus is a common connection between number of registers and oth units. Therefore, the data transfer through tri-state gates may require larger sinking and sourcing of current. The tri-state gate which provides more sinking and sourcing capacities is called **tri-state buffer**. The Fig. 2.3.2 shows the logic symbol for tri-state buffer.

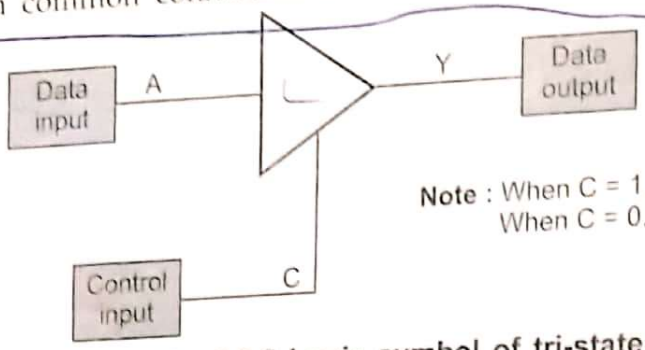


Fig. 2.3.2 Logic symbol of tri-state buffer

As shown in the Fig. 2.3.2, the control input decides the state of the output. When control input is 1, the data is available at the output, i.e. Y = A. On the other hand, when control input is 0, the output goes to a high-impedance state.

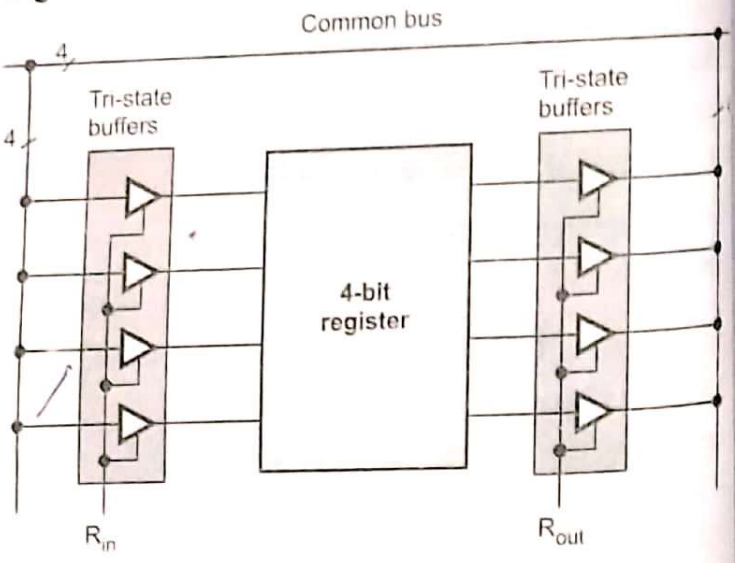


Fig. 2.3.3

The Fig. 2.3.3 shows, how the data transfer takes place between register and common bus. Here, 4-bit register is shown with tri-state buffers at inpu

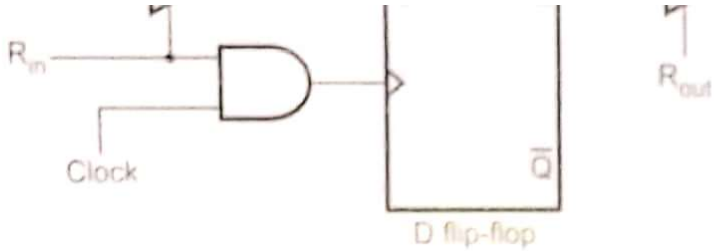


Fig. 2.3.5 One-bit register

It is important to note that when two or more output tri-state buffers are active at time, there is bus contention. To avoid bus contention no more than one output buffe may be in the active state at any given time. This means that the connected buffers mu be controlled such that only one output tri-state buffer has access to the bus line whi all other buffers are maintained in a high impedance state.

One way to ensure that no more than one output control input is active at any give time is to use a decoder. Let us assume that four registers are connected to the commo bus and we have four output control signals one for each register. For such system w requires 2 : 4 decoder. The four outputs of decoder act as four control signals, as show in the Fig. 2.3.6. When enable input of the decoder is 0, all of its four outputs are 0 an the bus line is in a high-impedance state because all four output tri-state buffers a disabled.

When the enable input is active, one of the output tri-state buffers will be activ depending on the binary values at the select inputs of the decoder.

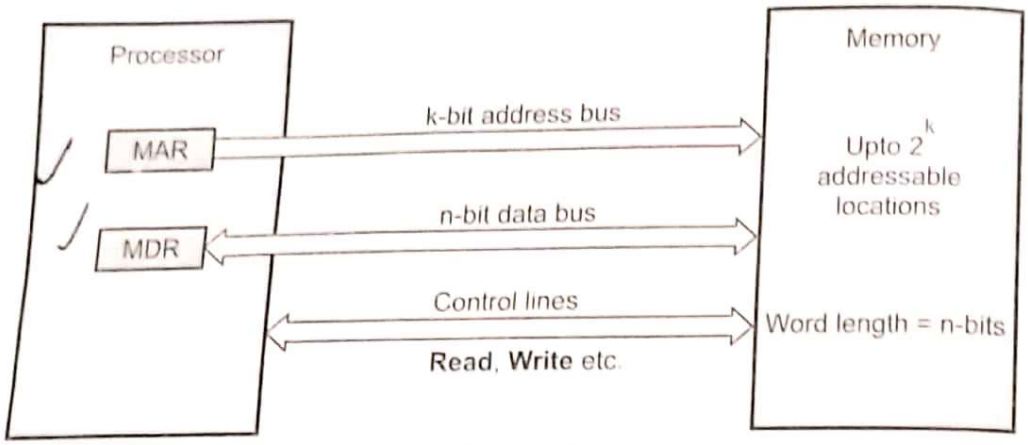


Fig. 2.3.7 Connection between memory and processor

In read operation, the address of the memory word is specified by address register AR and the data word read from the memory is loaded into the data register, DR. This read operation can be represented as,

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

As shown above letter M represents memory word whose address is specified by address register, AR. The control signal **Read** indicates that the operation is performed only when **Read** signal is active. Once the data is available in the DR register it can be transferred to any other register within the CPU. For example, it can be transferred to register, R_2 by following operation.

$$R_2 \leftarrow DR$$

It is important to note that for above operation the activation of **Read** signal is not necessary; it is necessary only when data is read from memory unit. The write operation transfers the contents of a data register to a memory word M selected by the address in the address register, AR. The write operation can be represented as,

Write : $M\ [AR] \leftarrow DR$

The control signal **Write** indicates that the operation is performed only when **Write** signal is active. When it is necessary to transfer data from any other CPU register to the memory we have to transfer the data from that register to the data register, DR before write operation. For example, if we want to transfer data from register R_1 to the memory whose address is in AR then this operation can be represented as,

$$DR \leftarrow R_1$$

Write : $M\ [AR] \leftarrow DR$

The Fig. 2.3.8 shows the communication between memory unit and multiple registers. As shown in the Fig. 2.3.8, register A_0 through A_3 are the address registers. The MUX 1 selects one of the address source for memory. The MUX 2 selects one of data source for write operation of memory. The decoder selects the destination register to

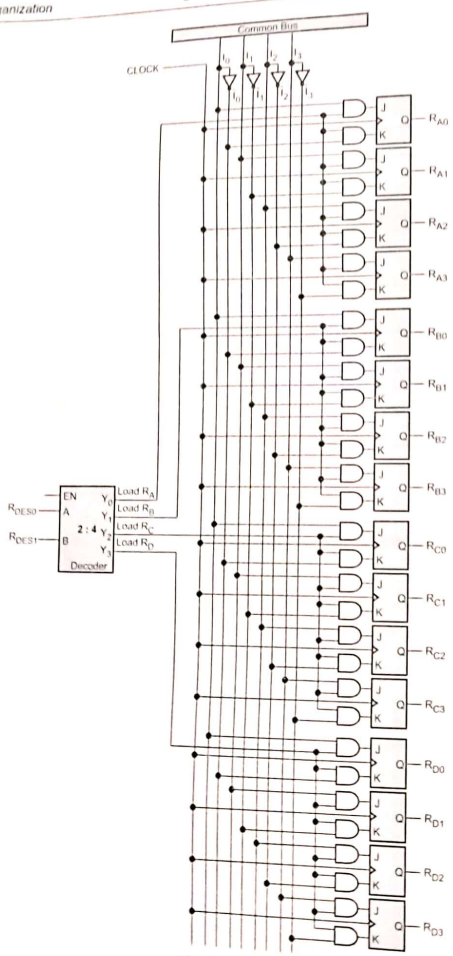


Fig. 2.3.9

Example 2.3.3 *Design a circuit for parallel load operation into one of the four 4-bit registers from a bus. Mention clearly control/selection bits and selection logic. Assume RS flip-flops.*

Solution : Use RS flip-flops instead of JK flip-flops in the solution of example 2.3.2.

Example 2.3.4 *Design a circuit which can be used to transfer data from any register to any other register out of four 4-bit registers A, B, C, D which uses RS flip-flop.*

Solution : The Fig. 2.3.10 shows the circuit for the desired operation. It consists of four 4-bit registers with RS flip-flops. Four multiplexers are used to select the source registers. Load inputs for registers are derived using 2 : 4 decoder which decides the destination register. The Table 2.3.3 shows the function table for the circuit.

RS_1	RS_0	R_{DES1}	R_{DES0}	Operation
0	0	0	0	$R_A \leftarrow R_A$
0	0	0	1	$R_B \leftarrow R_A$
0	0	1	0	$R_C \leftarrow R_A$
0	0	1	1	$R_D \leftarrow R_A$
0	1	0	0	$R_A \leftarrow R_B$
0	1	0	1	$R_B \leftarrow R_B$
0	1	1	0	$R_C \leftarrow R_B$
0	1	1	1	$R_D \leftarrow R_B$
1	0	0	0	$R_A \leftarrow R_C$
1	0	0	1	$R_B \leftarrow R_C$
1	0	1	0	$R_C \leftarrow R_C$
1	0	1	1	$R_D \leftarrow R_C$
1	1	0	0	$R_A \leftarrow R_D$
1	1	0	1	$R_B \leftarrow R_D$
1	1	1	0	$R_C \leftarrow R_D$
1	1	1	1	$R_D \leftarrow R_D$

Table 2.3.3

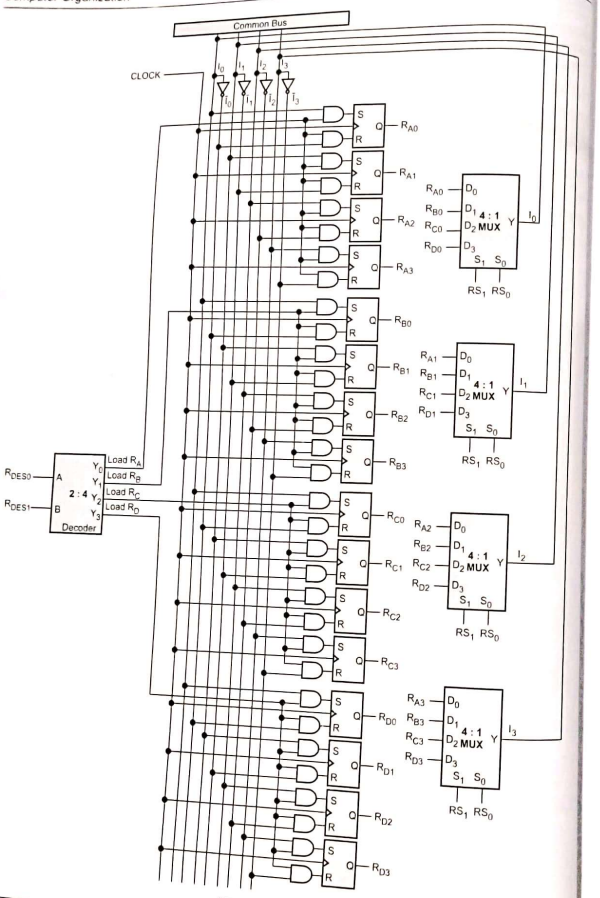


Fig. 2.3.10

Example 2.3.5 *There exists three 4-bit registers (A, B and C) and 4-bit data lines (say D). Design a circuit to transfer data from any register to any other register including data lines and vice-versa.*

Solution : The Fig. 2.3.11 shows the circuit for the desired operation. It consists of three 4-bit registers. Four multiplexers are used to select the data source. Load inputs for registers are derived using 2 : 4 decoder which decides the destination. The Table 2.3.4 shows the function table for the circuit.

RS_1	RS_0	R_{DES1}	R_{DES0}	Operation
0	0	0	0	$R_A \leftarrow R_A$
0	0	0	1	$R_B \leftarrow R_A$
0	0	1	0	$R_C \leftarrow R_A$
0	0	1	1	$D \leftarrow R_A$
0	1	0	0	$R_A \leftarrow R_B$
0	1	0	1	$R_B \leftarrow R_B$
0	1	1	0	$R_C \leftarrow R_B$
0	1	1	1	$D \leftarrow R_B$
1	0	0	0	$R_A \leftarrow R_C$
1	0	0	1	$R_B \leftarrow R_C$
1	0	1	0	$R_C \leftarrow R_C$
1	0	1	1	$D \leftarrow R_C$
1	1	0	0	$R_A \leftarrow D$
1	1	0	1	$R_B \leftarrow D$
1	1	1	0	$R_C \leftarrow D$
1	1	1	1	$D \leftarrow D$

Table 2.3.4

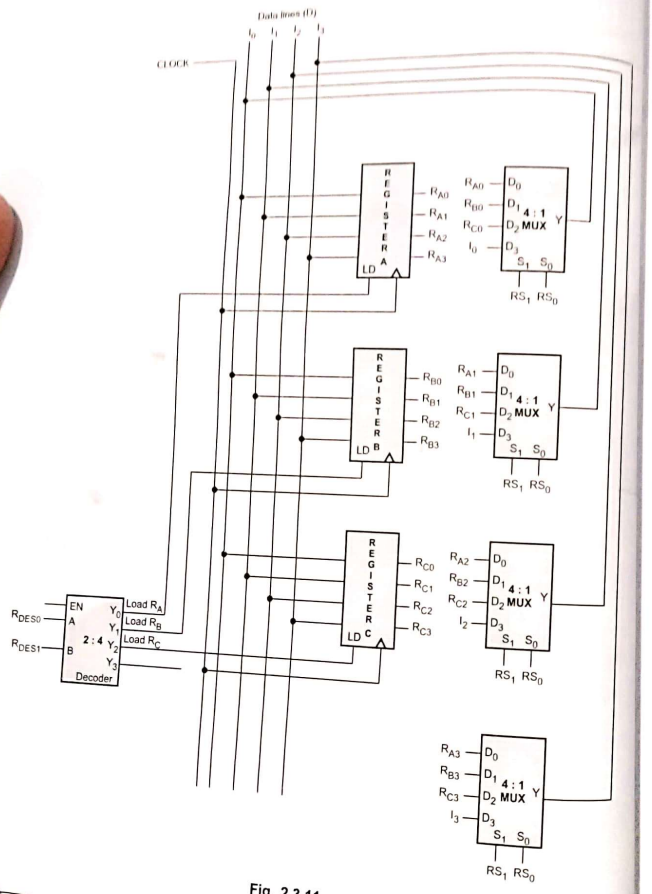


Fig. 2.3.11

Review Questions

1. *Explain how basic computer registers are connected to a common bus using a diagram.*
   GTU : Winter-12, Marks 7
2. *Design and explain a common bus system for four register each of 4 bit using multiplexer.*
   GTU : Dec.-11, Marks 7
3. *What is high impedance state in three-state buffer ? Explain three-state gate in detail.*
   GTU : May-12, Marks 7
4. *Design and explain common bus system using multiplexer.* **GTU : Summer-14, Marks 7**
5. *Explain the implementation of common bus using tri-state buffer.*
6. *Draw and explain the implementation of 1-bit register.*
7. *How can we use decoder to avoid bus contention ?*
8. *Explain the data transfer between register and memory.*
9. *Draw the common bus system with multiplexers for eight registers each having storage capacity four bits. How many multiplexers are required ? How many selection lines are required ?*
   GTU : June-05, Marks 8
10. *What is a tri-state device? Discuss its role in designing Bus system for 4 registers of 8 bits each.*
    GTU : June-06, Marks 5
11. *Explain the operation of three state bus buffers and show its use in design of common bus.*

2.4 Arithmetic Micro-operations

GTU : June-06,13, Winter-12, Dec.-09, Feb.-10, May-12

Recall that the micro-operation is an elementary operation performed with the data stored in registers. The micro-operation can be grouped into four major categories :

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

We have already seen the register transfer operations in section 2.1. This type of micro-operation does the transfer of information from register to another register; it does not change the information content at the time of transfer. On the other hand, other three types of micro-operations change the information contents. In this section we introduce a set of arithmetic micro-operations. The remaining two types of micro-operations are discussed in further two sections.

Operation	Representation	Description
Add	$R_3 \leftarrow R_1 + R_2$	Contents of R_1 and R_2 are added and result is transferred to R_3
Subtract	$R_3 \leftarrow R_1 - R_2$	Content of R_2 are subtracted from content of R_1 and result is transferred to R_3
1's Complement	$\overline{R_1}$	Complement the content of R_1
2's complement	$\overline{R_1} + 1$	Complement the contents of R_1 and add 1 in it
2's complement subtraction	$R_3 \leftarrow R_1 + \overline{R_2} + 1$	Add R_1 and the 2's complement of R_2
Increment	$R_1 \leftarrow R_1 + 1$	Increment the contents of R_1 by one.
Decrement	$R_1 \leftarrow R_1 - 1$	Decrement the contents of R_1 by one.

2.4.2 Parallel Adder

We have seen, a single full-adder is capable of adding two one-bit numbers and input carry. In order to add binary numbers with more than one bit, addit full-adders must be employed. A n-bit, parallel adder can be constructed using num of full adder circuits connected in parallel. Fig. 2.4.4 shows the block diagram of parallel adder using n number of full-adder circuits connected in cascade, i.e. the output of each adder is connected to the carry input of the next higher-order adder.

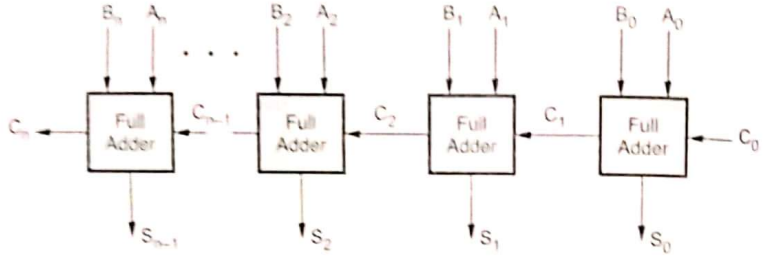


Fig. 2.4.4 Block diagram of n-bit parallel adder

It should be noted that either a half-adder can be used for the least signific position or the carry input of a full-adder is made 0 because there is no carry into least significant bit position.

2.4.3 Parallel Subtractor

The subtraction of binary numbers can be done most conveniently by means complements. The subtraction A - B can be done by taking the 2's complement of B a adding it to A. The 2's complement can be obtained by taking the 1's complement a adding one to the least significant pair of bits. The 1's complement can be implemen with inverters and a one can be added to the sum through the input carry, as shown the Fig. 2.4.5.

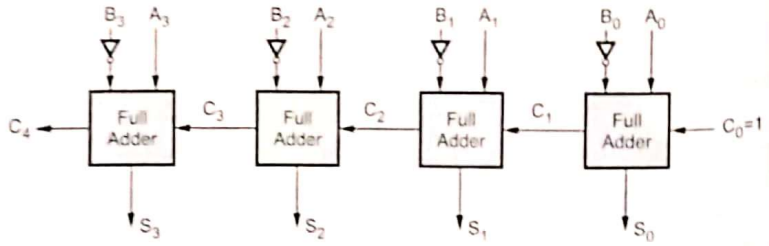


Fig. 2.4.5 4-bit parallel subtractor

The counter is a sequential circuit. The Fig. 2.4.7 shows a combinational circuit to implement increment micro-operation. Here, 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 increment. The output carry from lower order half adder is connected to the one of the inputs of the next higher order half adder. Such circuit gets the four bit number from bits A_0 through A_3, adds one to it, and generates the incremented output in S_0 through S_3. If number to be incremented is 1111, the output carry C_4 is generated and we get 0000 at S_0 through S_3.

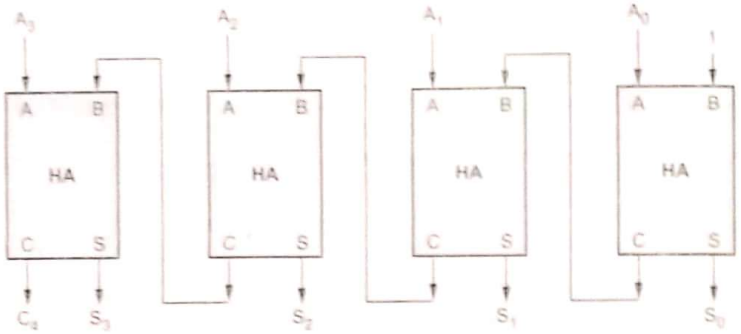
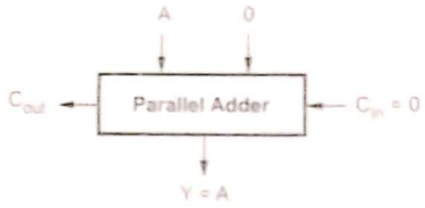
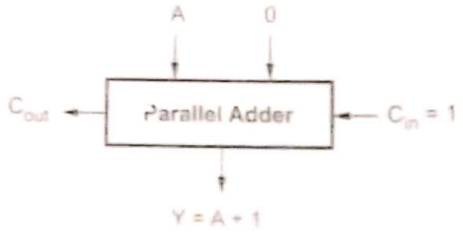


Fig. 2.4.7 4-bit incrementer using combinational circuit

2.4.6 Arithmetic Circuit

We have already seen how parallel adder can be constructed using number of full adder circuits connected in cascade. Let us see the implementation of various functions of arithmetic unit using parallel adder. Table 2.4.3 shows these implementations.

Function	Implementation
1. $Y \leftarrow A$	
2. $Y \leftarrow A + 1$	

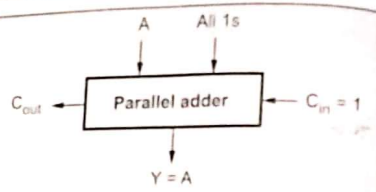
8. Y = A	

Table 2.4.3 Implementation of basic arithmetic operations

Looking at Table 2.4.3 we can realize that by selecting proper B and C_{in} inputs can implement basic arithmetic operations. There are four possible B inputs: complement, all 1s and all 0s and two possible C_{in} inputs 0 and 1.

Referring Table 2.4.4, we can write truth table for B input with respect to S_0 and signals as follows :

S_1	S_0	B'
0	0	0
0	1	B
1	0	$\overline{B}$
1	1	1

Note - B' indicates modified B input

Table 2.4.4 Truth table

K-map simplification

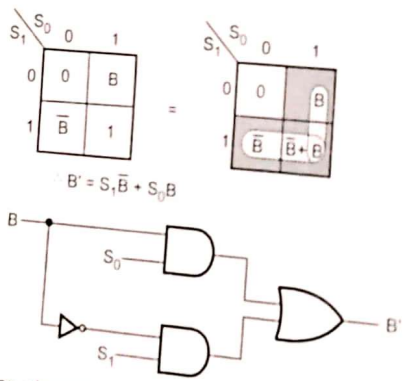


Fig. 2.4.8 K-map simplification and logic diagram of modified B input

Using the logic diagram of modified B input we can draw the logic diagram arithmetic circuit, as shown in Fig. 2.4.9 (See Fig. 2.4.9 on next page).

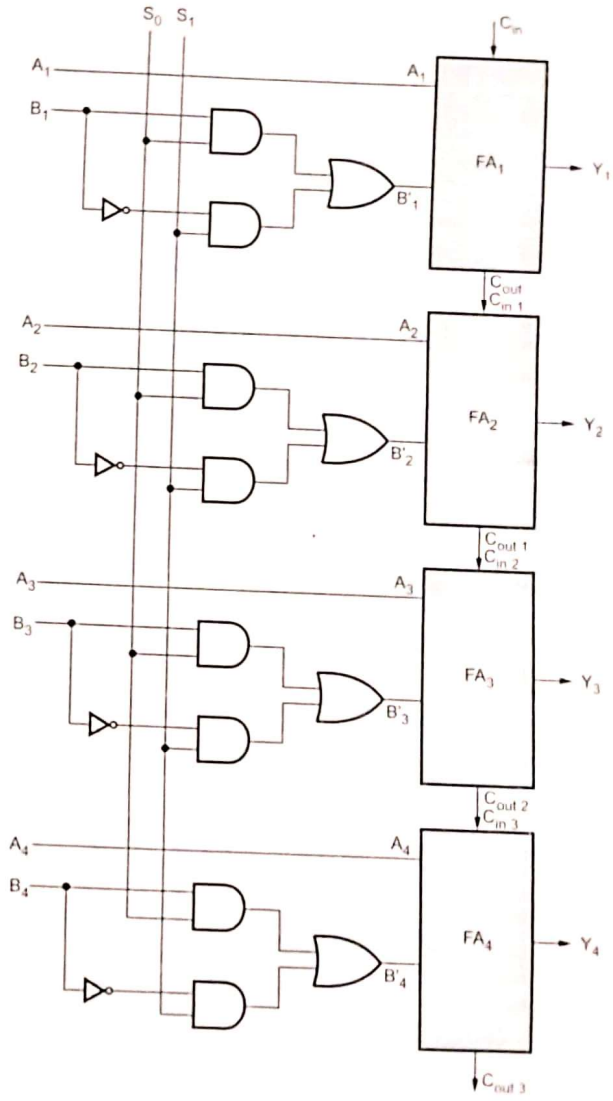
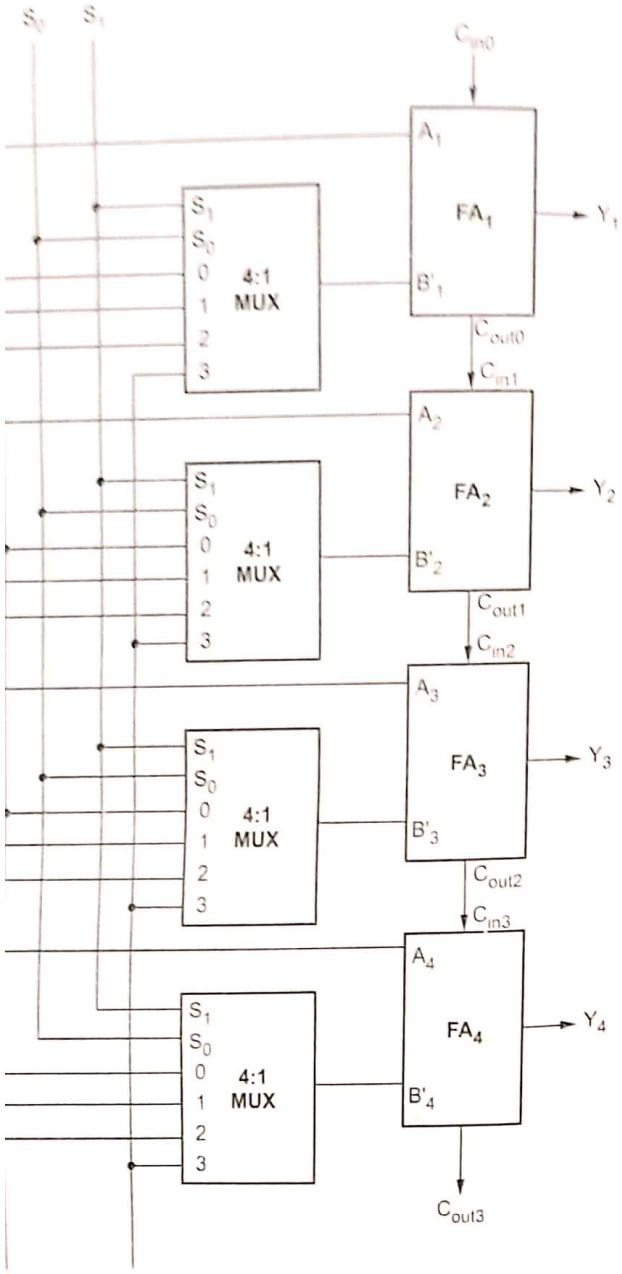


Fig. 2.4.9 Logic diagram of arithmetic circuit

e arithmetic circuit using multiplexers. Refer Table 2.4.4
input of the full adder. We have to select one of the
input S_1 and S_0. We can implement a circuit for
ltiplexer. The Fig. 2.4.10 shows the arithmetic circuit



Four bit arithmetic circuit using multiplexer

Example 2.4.2 *Consider the following register transfer statements for two 4-bit registers R_1 and R_2*

$xT : R_1 \leftarrow R_1 + R_2, \quad \bar{x}T = R_1 \leftarrow R_2$

Every time that variable $T = 1$, either the contents of R_2 is added to the content of R_1 if $x = 1$ or the content of R_2 is transferred to R_1 if $x = 0$. Draw a diagram showing the hardware implementation of the two statements. Use block diagram for the two 4-bit register, a 4-bit adder and a quadruple 2:1 line multiplexer that selects the input to R_1. In the diagram show the control variables x and T, select the inputs of the multiplexer and the load input of register R_1.

Solution :

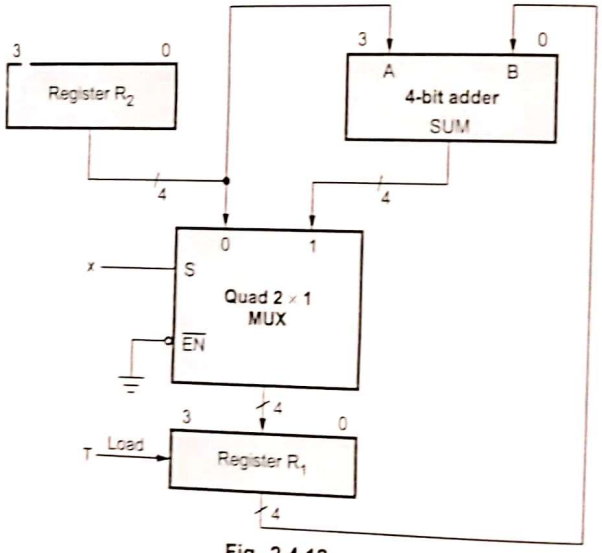


Fig. 2.4.12

Example 2.4.3 *Show the hardware implementation for following statement.*

$\bar{X}\,Y\,T_0 + T_1 + \bar{X}\,Y\,T_2 : A \leftarrow A + B$

Solution :

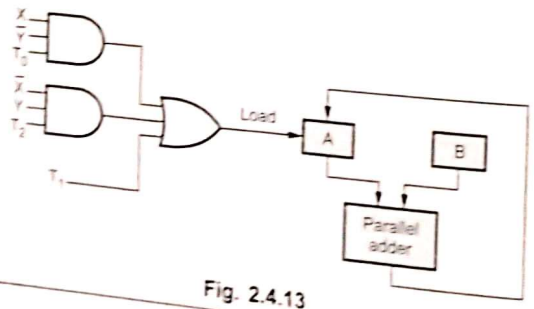


Fig. 2.4.13

Example 2.4.4 *Define RTL and Micro-operation. Write RTL statement for following :*

If (p = 1) then R1 = R1 + R1 else if (p = 0) then R1 = R1 + R2. Also draw the block diagram for the same.

GTU : June-06, Marks 9

Solution : P : R1 ← R1 + R1

$\bar{P}$: R1 ← R1 + R2

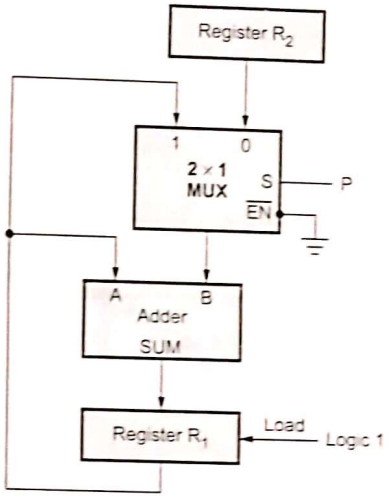


Fig. 2.4.14

Example 2.4.5 *Show that the statement :*

$A \leftarrow A + A$ symbolizes a shift left micro-operation.

Solution : Shift left by 1-bit position of any register contents is equivalent to multiply contents by 2, i.e. adding contents itself (A * 2 = A + A).

For example, A = 06

Add operation

		1	1			← Carry
+	A	0	1	1	0	+ (6)
	A	0	1	1	0	+ (6)
	A + A	1	1	0	0	12

Left shift operation

0	1	1	0	Initial contents of A
1	1	0	0	Contents of A shifted by 1-bit

Results are same, thus it is proved at A + A = Left shift by 1-bit position

Example 2.4.6 *Show the block diagram that executes the statement*

$qT_3 : A \leftarrow B, B \leftarrow A$.

Solution :

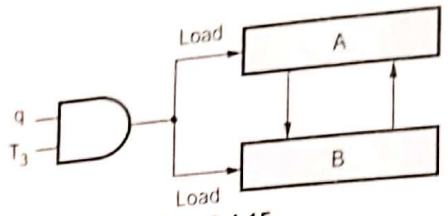


Fig. 2.4.15

Example 2.4.7 *An 8-bit register A has one input q. The register operation is described symbolically as* $Y : A_1 \leftarrow q,\ A_{i+1} \leftarrow A_i,\ i = 1, 2, 3 \ldots 7$.

Solution : The register operation described symbolically can be drawn as follows :

The Fig. 2.4.16 shows that it is shift left register with serial input q and contr input Y.

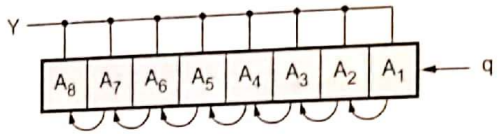


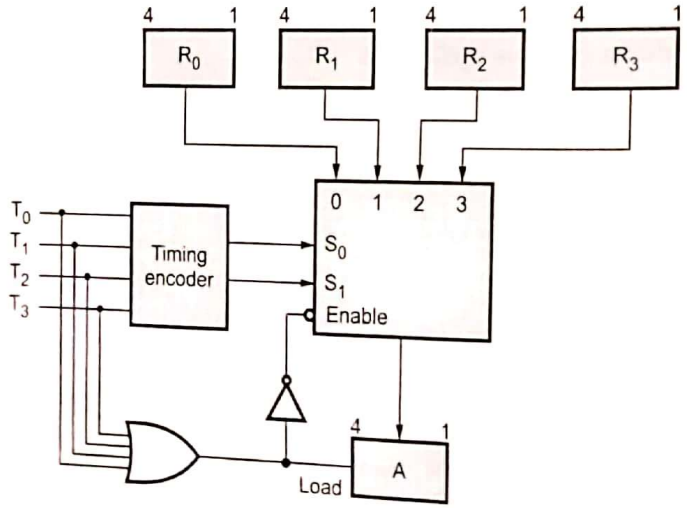
Fig. 2.4.16

Example 2.4.8 *Show the hardware implementation for the following statements. The registers are 4-bit in length.*

$T_0 : A \leftarrow R_0 \quad T_1 : A \leftarrow R_1 \quad T_2 : A \leftarrow R_2 \quad T_3 : A \leftarrow R_3$

Solution : The given statements can be implemented by two methos: using multiplexe and using enable signals of registers.

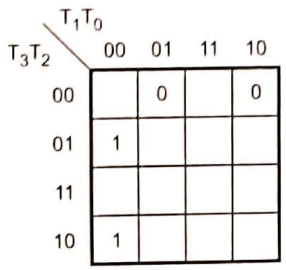
a) Using multiplexer :



The excitation table for timing encoder is as follows

T_3	T_2	T_1	T_0	S_1	S_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

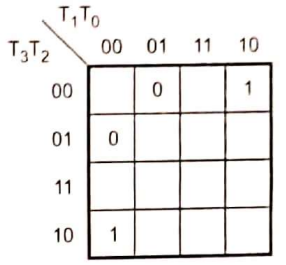
Table 2.4.6



(a) $S_1 = \overline{T}_3 T_2 \overline{T}_1 \overline{T}_0 + T_3 \overline{T}_2 T_1 \overline{T}_0$

$= (T_2 \oplus T_3) \overline{T}_1 \overline{T}_0$

$= (T_2 \oplus T_3) \overline{(T_0 + T_1)}$



(b) $S_0 = \overline{T}_3 \overline{T}_2 T_1 \overline{T}_0 + T_3 \overline{T}_2 \overline{T}_1 \overline{T}_0$

$= (T_3 \oplus T_1) \overline{T}_2 \overline{T}_0$

$= (T_3 \oplus T_1) \overline{(T_0 + T_2)}$

Fig. 2.4.18

Therefore, hardware implementation for timing encoder is as follows :

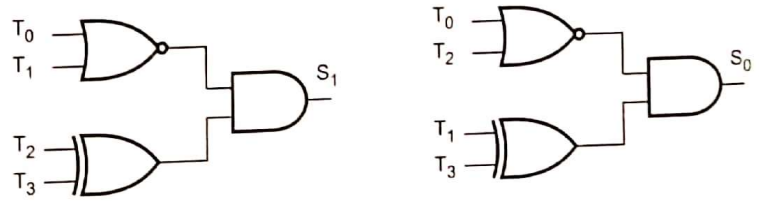


Fig. 2.4.19

b) Using enable signals of registers

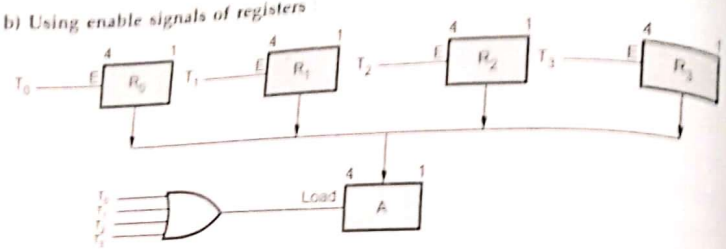


Fig. 2.4.20

Example 2.4.9 *Show the hardware required to implement the following logic micro-operations*
a) $T_1 : F \leftarrow A \wedge B$ *b)* $T_2 : G \leftarrow C \vee D$ *c)* $T_3 : E \leftarrow \overline{E}$

Solution :

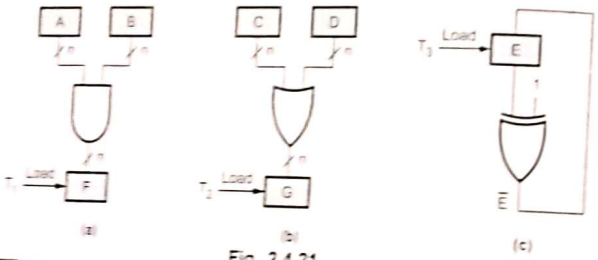


Fig. 2.4.21

Example 2.4.10 *Show the hardware implementation for following statement.*
$\overline{X}\ Y\ T_0 + T_1 + \overline{X}\ Y\ T_2 : A \leftarrow A + B$

Solution :

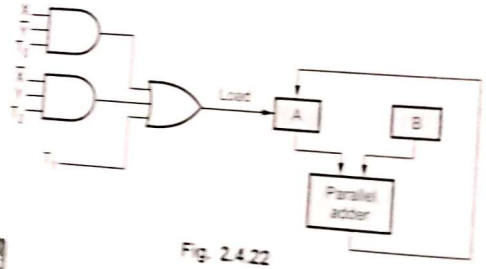


Fig. 2.4.22

Review Questions

1. Explain various arithmetic micro-operations.
2. What is half adder ?
3. What is full adder ?
4. Explain the n-bit parallel adder with the help of neat diagram.

5. Explain the n bit subtractor with the help of neat diagram.
6. Draw and explain the hardware for integer addition and subtraction.
7. Draw and explain the overflow detector logic circuit.
8. Explain the 4 bit arithmetic circuit using multiplexer.
9. Explain 4 bit arithmetic circuit with its function table. **GTU : Winter-12, Marks 7**
10. Show the block diagram of the hardware that implements the following register transfer statement.
T2 : R2 ← R1 + R2 **GTU : Dec.-09, Marks 3**
11. Explain 4 bit incrementer with a necessary diagram. **GTU : Feb.-10, May-12, Marks 7**

2.5 Logic Micro-operations

GTU : June-06, Dec.-07, 12

Logic micro-operations perform logic operations such as AND, OR, complement and XOR on the strings of bits stored in registers. These operations consider each bit of the register separately and treat them as binary variables. Special symbols are used for the logic micro-operations OR, AND and complement to distinguish them from the corresponding symbols used to express Boolean functions. The Table 2.5.1 gives the list of logic micro-operations and their symbols.

Logic micro-operation	Symbol
OR	$\vee$
AND	$\wedge$
Complement	Bar (¯)
EX-OR	$\oplus$

Table 2.5.1 Basic logic micro-operations and symbols

AND micro-operation : This micro-operation logically ANDs the bits of one register with the bits of another register having same word length. Table 2.5.2 shows the truth table for AND micro-operation.

B_n of Register	B_n of another Register	B_n of Result
0	0	0
0	1	0
1	0	0
1	1	1

B_n represents the particular bit position in the register. Here n can be 0, 1, 2, 3, ...

Table 2.5.2 Truth table of AND micro-operation

For example,

$T : R_1 \leftarrow R_1 \wedge R_2$

B_n of Register	B_n of another Register	B_n of Result
0	0	0
0	1	1
1	0	1
1	1	1

B_n represents the particular bit position in the register. Here n can be 0, 1, 2,3, ...

Table 2.5.3

For example,

$$T_1 : R_1 \leftarrow R_1 \vee R_2$$

Statement represents the OR micro-operation with the contents of two registers R_1 and R_2. The result of micro-operation is stored in the register R_1. In the statement T_1 is control input and when it is logic 1 the micro-operation is performed.

Complement micro-operation : This micro-operation complements all bits in the register.

For example,

$$T_2 : \overline{R}_1$$

Statement represents the complement micro-operation with the contents of register R_1. In the statement T_2 is a control input and when it is logic 1 the micro-operation is performed.

XOR micro-operation : This micro-operation logically XORs the bits of one register with the bits of another register having same word length. Table 2.5.4 shows the truth table for XOR micro-operation.

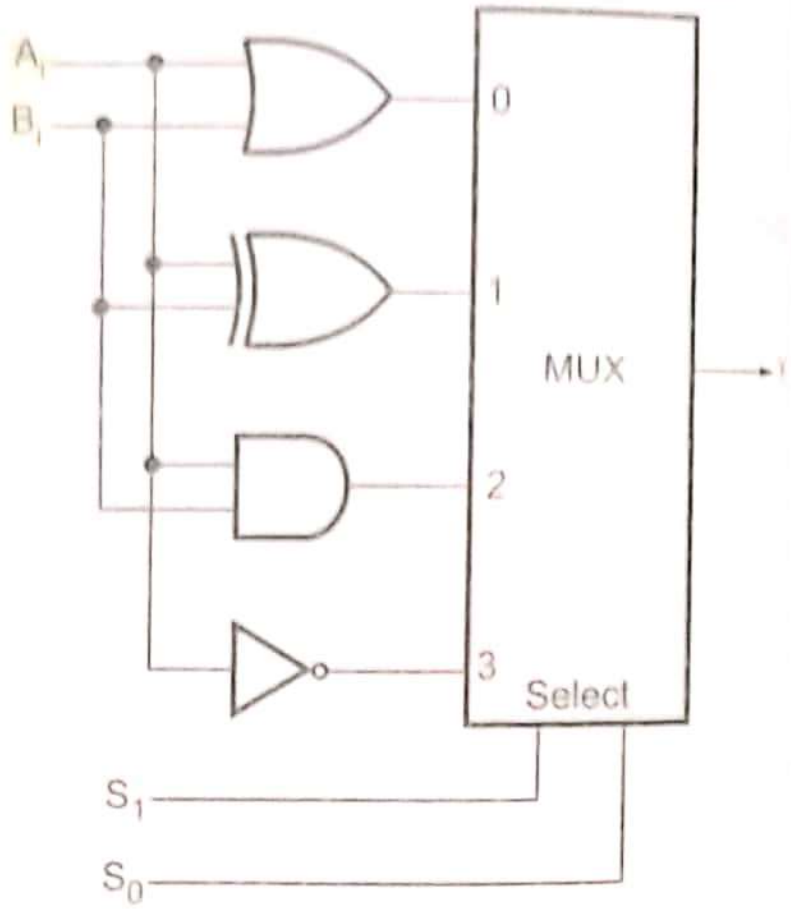
Table 2.5.6 List of logic micro-operations

Hardware implementation

The design of logic circuit is comparatively simple than the design of arith circuit. Fig. 2.5.1 shows the function table and the logic diagram for logic circuit.

S_1	S_0	Output	Function
0	0	$Z = A \vee B$	OR
0	1	$Z = A \oplus B$	XOR
1	0	$Z = A \wedge B$	AND
1	1	$Z = \overline{A}$	NOT

a) Function table



b) Logic diagram

Fig. 2.5.1 Logic circuit

Note The logic diagram shown in Fig. 2.5.1 (b) is repeated n times for n-bit lo circuit. Therefore, inputs A and B, and output Y are shown with subscript i, wh

As shown in the Fig. 2.5.3, the leftmost four bits of B are 1's, so the corresp bits of A are complemented. The rightmost four bits of A with corresponding 0 remain unchanged.

Example 3 : Selective mask

This operation clears to 0 the bits in A only where there are corresponding 0 We know that in AND operation result is logic 0 when one of the inputs are logic 0. Therefore, the AND micro-operation can be used to selectively mask the bi register.

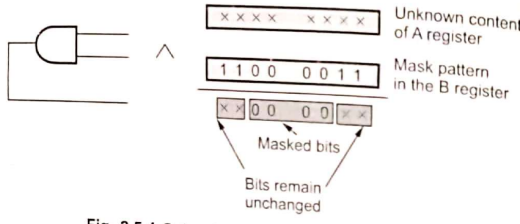


Fig. 2.5.4 Selective mask micro-operation

As shown in the Fig. 2.5.4, the four middle bits of B are 0's, so the correspondin of A are cleared. The two leftmost and two rightmost bits of A with correspondin in B remain unchanged.

Example 4 : Selective clear

The selective clear operation is similar to the selective mask operation except tha bits of A are cleared only when there are corresponding 1's in B. This operation ca performed using AND operation with complemented B input.

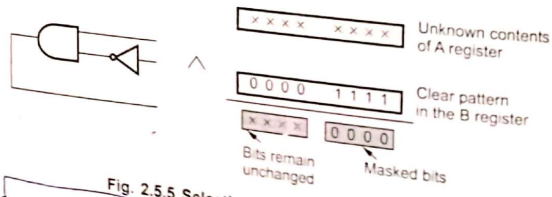


Fig. 2.5.5 Selective clear micro-operation

Example 5 : Insert micro-operation

This operation inserts a new value into a group of bits. This is done by first mask the bits and then ORing them with the required value. For example.

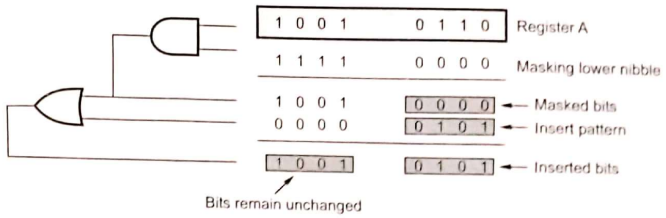


Fig. 2.5.6

Example 2.5.1 *Show the hardware required to implement the following logic micro-operations :*
a) $T_1 : F \leftarrow A \wedge B$ *b)* $T_2 : G \leftarrow C \vee D$ *c)* $T_3 : E \leftarrow \bar{E}$.

Solution :

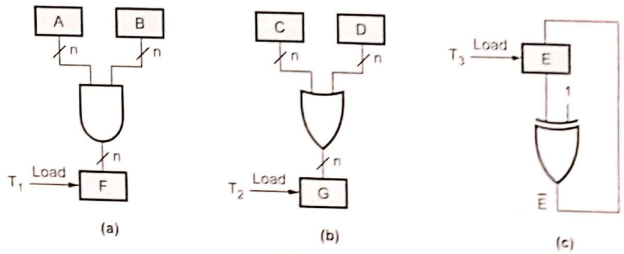


Fig. 2.5.7

Review Questions

1. *Explain any two basic logic operations.*
2. *Explain the applications of logic operations.*
3. *What is masking ?* GTU : June-06, Marks 2
4. *Explain selective set, selective complement and selective clear.* GTU : Dec.-09, Marks 2
5. *What is logic micro-operations ? List out different logic micro-operations that can be performed with two binary variables.* GTU : Winter-12, Marks 7

2.6 Shift Micro-operations

GTU : Dec.-11, June-06

Shift micro-operations are used to shift the contents of a register to the left or the right. This functionality of shift micro-operation is useful for serial transfer of data. When the bits are shifted, first flip-flop in the register receives its binary information from the serial input and the last flip flop of register gives the information to the serial

output. During a shift left operation the serial input transfers a bit into the ri position. During a shift right operation the serial input transfers a bit into the l position. There are three types of shift operations :

- Logical shift
- Arithmetic shift
- Circular shift or rotate

2.6.1 Logic Shift

In logical shift operation the serial input is 0 i.e., the vacant position created in the register due to shift operation is filled with zero. This is illustrated in Fig. There are two logical shift micro-operations logical shift left (Lshl) and logical shi (LshR). For example,

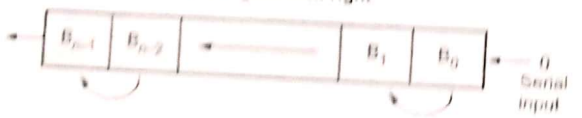
R1 ← Lshl R1

R2 ← LshR R2

are two micro-operations that specify a 1-bit shift to the left of the content of R1 and a 1-bit shift to the right of the content of register R2. Since sourc destination registers are same, the register symbol must be the same on both the s arrow.



(a) Logical shift right

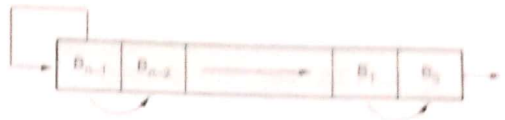


(b) Logical shift left

Fig. 2.6.1 Logical shift micro-operations

2.6.2 Arithmetic Shift

In logical shift operation we have seen that the vacant positions created while register due to shift operation are filled with zeroes. In arithmetic right shift necessary to repeat the sign bit as the fill-in bit for the vacated position. requirement on right shifting distinguishes arithmetic shifts from logical Otherwise both shift operations are very similar. The Fig. 2.6.2 shows the arithmetic shift (Ashr) and the arithmetic left shift (Ashl). The arithmetic shift left is same as shift left.



(a) Arithmetic shift right

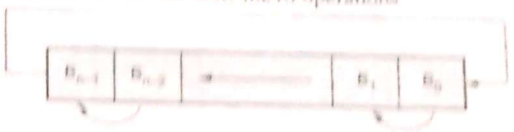


(b) Arithmetic shift left

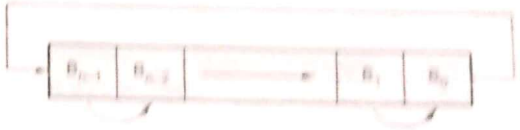
Fig. 2.6.2 Arithmetic shift micro-operations

2.6.3 Rotate or Circular Shift Micro-operations

In shift micro-operations the bit shifted out of the register is lost. The rotate micro-operations, on the other hand, preserve this bit. They move the bit that is shifted out of one end of the register back into the other end as shown in Fig. 2.6.3. The two types of rotate micro-operations are Rotate Left (RL) and Rotate Right (RR). These operations are also known as circular shift micro-operations.



(a) Rotate left



(b) Rotate right

Fig. 2.6.3 Rotate micro-operations

The Table 2.6.1 gives all shift micro-operations.

Symbol	Description
R1 ← Lshl R	Logical shift left register R
R1 ← LshR R	Logical shift right register R
R1 ← Ashl R	Arithmetic shift left register R
R1 ← AshR R	Arithmetic shift right register R

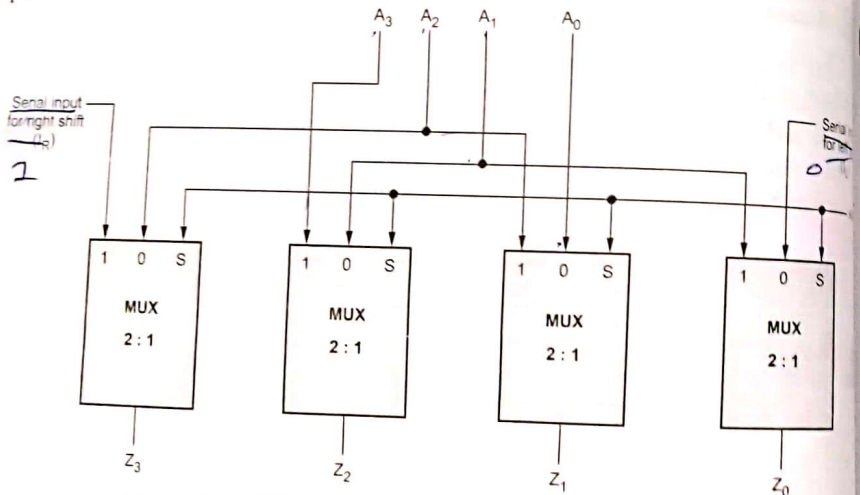
$R \leftarrow RL\ R$	Rotate (circular-shift) left register R
$R \leftarrow RR\ R$	Rotate (circular-shift) right register R

Table 2.6.1 Shift micro-operations

Hardware implementation

It is possible to implement shift operations using a bidirectional shift register... parallel load. However, it is a sequential circuit, it needs a clock pulse to load d... another pulse to initiate the shift. In a processor unit with many register it ... efficient to implement the shift operations with a combinational circuit.

The Fig. 2.6.4 shows the combinational shifter circuit. It consists of ... 2 : 1 multiplexers. The shifter circuit has four data inputs, A_0 through A_3, and four ... outputs, Z_0 through Z_3. There are two serial inputs, one for shift left and the oth... shift right. The select input is used to select shift right or shift left operation. ... S = 1, shift left operation is performed and when S = 0 shift right operat... performed.



Function Table	
Select	Output
S	$Z_3\ Z_2\ Z_1\ Z_0$
0	$A_2\ A_1\ A_0\ I_L$
1	$I_R\ A_3\ A_2\ A_1$

Note :

S = 0 Left shift operation
S = 1 Right shift operation

Fig. 2.6.4 4-bit combinational shifter cir...

Example 2.6.1 *Show that the statement :*

$A \leftarrow A + A$ *symbolizes a shift left micro-operation.*

Solution : Shift left by 1-bit position of any register contents is equivalent to multiply contents by 2, i.e. adding contents itself (A * 2 = A + A)

For example, A = 06

Add operation

		1	1			← Carry
	A	0	1	1	0	(6)
+	A	0	1	1	0	+ (6)
A + A		1	1	0	0	12

Left shift operation

0	1	1	0	Initial contents of A
1	1	0	0	Contents of A shifted by 1-bit

Results are same, thus it is proved at A + A = Left shift by 1-bit position

Example 2.6.2 *An 8-bit register A has one input q. The register operation is described symbolically as* $Y : A_1 \leftarrow q,\ A_{i+1} \leftarrow A_i,\ i = 1, 2, 3 \ldots 7$.

Solution : The register operation described symbolically can be drawn as follows :

The Fig. 2.6.5 shows that it is shift left register with serial input q and control input Y.

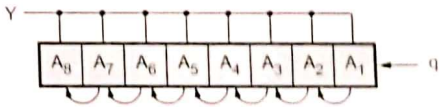


Fig. 2.6.5

Review Questions

1. *What is logic shift and arithmetic shift ?*
2. *Explain the difference between logic shift and arithmetic shift.*
3. *Explain circular shift micro-operations.*
4. *Explain 4-bit combinational shifter circuit.*
5. *List and explain different types of shift micro-operation.* GTU : Dec.-11, Marks 7
6. *Answer in brief : Types of shift micro-operations.* GTU : June-06, Marks 2

2.7 Arithmetic Logic Shift Unit

We have seen the design of arithmetic and logic circuits for a simple ALU. By combining both circuits with the help of multiplexer we can get the arithmetic and logic circuit, as shown in the Fig. 2.7.1. The Fig. 2.7.1 shows one stage of an arithmetic logic shift unit. The subscript i designates a typical stage. Inputs A_i and B_i are applied to both the arithmetic and logic units. Inputs S_1 and S_0 are used to select a particular

micro-operation. A 4 : 1 multiplexer at the output chooses between an arithmetic ... in Y_i and a logic output in Z_i. The data in the multiplexer are selected with ... and S_2. The other two data inputs to the multiplexer receive inputs A_{i-1} for ... operation and A_{i+1} for shift right operation. It is important to note that the ... shown in Fig. 2.7.1 shows just one typical stage. The circuit shown in Fig. 2.7.1 ... repeated n times for an n-bit ALU. For multistage ALU, the output carry C_{i+1} ... previous arithmetic stage must be connected to the input carry C_i of the next ... sequence.

The Table 2.7.1 gives the list of operations that can be performed using ... logic shift circuit shown in Fig. 2.7.1.

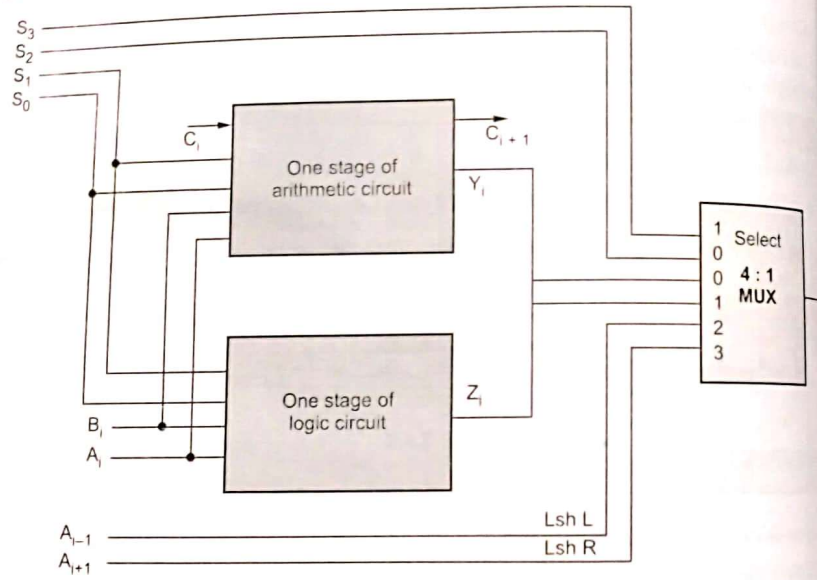


Fig. 2.7.1 One stage of arithmetic logic shift circuit

Select inputs					Operation	Function
S_3	S_2	S_1	S_0	C_{in}		
0	0	0	0	0	F = A	Transfer A
0	0	0	0	1	F = A + 1	Increment A
0	0	0	1	0	F = A + B	Add A and B
0	0	0	1	1	F = A + B + 1	Add A and B with ca...
0	0	1	0	0	F = A + $\overline{B}$	Subtract with borrow

0	0	1	0	1	F = A + $\overline{B}$ + 1	Subtract
0	0	1	1	0	F = A – 1	Decrement A
0	0	1	1	1	F = A	Transfer A
0	1	0	0	X	F = A ∧ B	AND
0	1	0	1	X	F = A ∨ B	OR
0	1	1	0	X	F = A ⊕ B	EX-OR
0	1	1	1	X	F = $\overline{A}$	Complement A
1	0	X	X	X	F = LshL A	Logical shift left A
1	1	X	X	X	F = LshR A	Logical shift right A

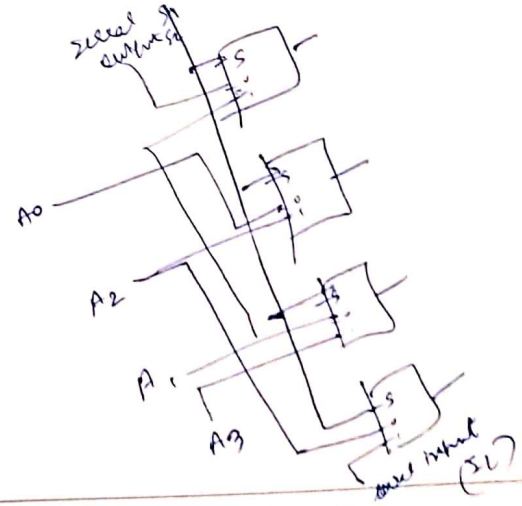
Table 2.7.1 Function table for arithmetic logic shift circuit

As shown in the Table 2.7.1 first eight operation are arithmetic operations and are selected when S_3 S_2 = 00. The next four are logic operations and are selected when S_3 S_2 = 01. The last two operations are shift operations and are selected when S_3 S_2 = 10 and S_3 S_2 = 11. The carry input has no effect during logic and shift operations and hence marked as don't care.

Review Question

1. *Draw and explain the one stage of arithmetic logic shift circuit.*

❑❑❑



Notes

3 Basic Computer Organization and Design

Syllabus

Instruction codes, Computer registers, Computer instructions, Timing and control, Instruction cycle, Memory-reference instructions, Input-output and interrupt, Complete computer description, Design of basic computer, Design of accumulator unit

Contents

3.1 Instruction Codes ..... **Dec.-05,06, June-06,** Marks 6

3.2 Computer Registers ..... **Dec.-02,05,06, June-05,06, May-11, Winter-13,** Marks 8

3.3 Computer Instructions ..... **June-05,06,08, Dec.-05,06,11, Jan.-10, May-12, Winter-13, Summer-14,** Marks 8

3.4 Timing and Control ..... **June-05, May-11,** Marks 7

3.5 Instruction Cycle ..... **Jan-07,13, Dec.-11, May-12,13, Winter-12,13, Summer-13,14,** Marks 7

3.6 Input-Output and Interrupt ..... **Dec.-05,06, June-06, May-11, Summer-14,** Marks 8

3.7 Complete Computer Description

3.8 Design of Basic Computer ..... **Dec.-10,** Marks 7

3.9 Design of Accumulator Logic ..... **Dec.-03, 06, Mar-12,** Marks 8

3.1 Instruction Codes

GTU : Dec.-05,06, 2...

Every computer system has a central processing unit. The operation of the processing unit and the computer system is determined by the instructions executed central processing unit. These instructions are known as **machine instructions** or **computer instructions**. Machine instructions are in the form of binary codes known as **instruction codes**. An instruction code is a group of bits that instructs the computer to perform a specific operation.

Each instruction of the CPU contain specific information fields, which are required to execute it. These information fields of instructions are called elements of instruction. These are :

Operation code : The operation code field in the instruction specifies the operation to be performed. The operation is specified by binary code, hence the name operation code or simply opcode.

Source / Destination operand : The source/destination operand field directly specifies the source/destination operand for the instruction.

Source operand address : We know that the operation specified by the instruction may require one or more operands. The source operand may be in the CPU register or in the memory. Many times the instruction specifies the address of the source operand so that operand(s) can be accessed and operated by the CPU according to the instruction.

Destination operand address : The operation executed by the CPU may produce a result. Most of the times the result is stored in one of the operand. Such operand is known as destination operand. The instruction which produce result specifies the destination operand address.

Next instruction address : The next instruction address tells the CPU from where to fetch the next instruction after completion of execution of current instruction. For JUMP and BRANCH instructions the address of the next instruction is specified within the instruction. However, for other instructions, the next instruction to be fetched immediately follows the current instruction.

Types of operands : We have seen that each instruction in a program specifies operation to be performed and data to be processed. For this reason, an instruction is divided into two parts : its operation code (opcode) and its operands. The operand is another name for data. It may appear in different forms :

- Addresses
- Numbers

- Characters
- Logical Data

Addresses : The addresses are in fact a form of data. In many situations, some calculation must be performed on the operand reference in an instruction to determine physical address.

Numbers : All computer supports numeric data types. The common numeric data types are :

- Integer or fixed point
- Point
- Decimal

Characters : For documentation a common form of data is text or character strings. Today, most of the computers use ASCII (American Standard Code for Information Interchange) code for character represented by a unique 7 bit pattern ; thus, 128 different characters can be represented. However, the ASCII encoded characters are always stored and transmitted using 8-bits per character. The eighth bit may be set to 0 or used as a parity bit for error detection.

Another code used to encode characters is the Extended Binary Coded Decimal Interchange Code (EBCDIC).

Logical data : Most of the processors interpret data as a bit, byte, word or double word. These are referred to as units of data. When data is viewed as n 1-bit items of data, each item having the value 0 or 1, it is considered as a logical data. The logical data is used to store an array of Boolean or binary data items and with logical data we can manipulate the bits of data items.

Stored program organization

The organization of the computer is defined by its internal registers, the timing and control structure and the set of instructions that it uses. The Fig. 2.1.1 shows the simple computer organization. It consists of processor register, instruction register and operand register. The instruction register is divided into two parts. The first part specifies the operation to be performed and the second specifies an address. The address part tells where to find an operand in memory. This operand is read from the memory and stored in the operand register. It is then used as a data to be operated on together with the data stored in the processor register.

The memory consists of 4096 words of 16-bit each. To address 4096 words we need 12 bits since $2^{12} = 4096$. Thus 12-bits in the instruction are reserved for address and the remaining four bits are available for operation code or opcode. Four bits of opcode are used to specify one out of 16 possible operations.

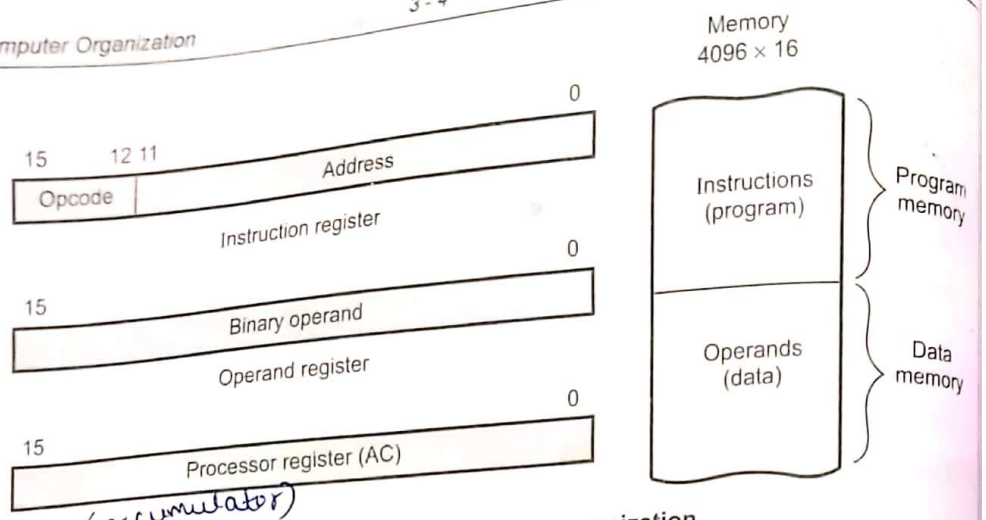


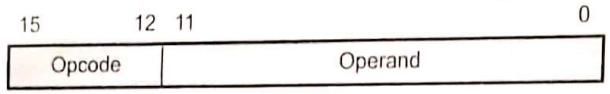
Fig. 3.1.1 Stored program organization

The control reads the 16-bit instruction from the program memory. It uses 12 lea nificant bits as an address to read a 16-bit operand from the data memory. It the executes the operation specified by the operation code. Most of the times the result the operation is stored (accumulated) in the processor register. Hence processor regist is also known as accumulator (AC).

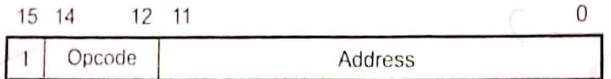
Some instructions do not need an operand from memory. For example, clear AC complement AC etc. In such cases, the second part of the instruction code (bit 0 - bit 11 is not needed for specifying a memory address and it can be used to specify othe operations for the computer.

Immediate, direct and indirect operand

When the second part of an instruction code specifies an operand, the instruction said to have an immediate operand. When the second part specifies the address of a operand, the instruction is said to have a direct address and when the second par specifies the address of the memory from where we get address of the operand, th instruction is said to have an indirect address. This is illustrated in the Fig. 3.1.3.



Instruction format for immediate operand



Instruction format for direct and indirect address

used to find the address of the operand from the memory. The address of the op... in this case is 148. The operand is read from address 148 and it is then added wi... contents of AC and the result of addition is stored in the AC. Indirect ad... instruction requires two memory references - To read the address of an operand a... read operand itself.

Effective address : It is the address of the operand in a computer type instructio... target address in a branch type instruction. It is 158 for instruction in Fig. 3.1.3 (b) a... is 148 for instruction in Fig. 3.1.3 (c).

Review Questions

1. *What do you mean by instruction code ?*
2. *Write a note on stored program organization.*
3. *Draw the instruction format for immediate operand.*
4. *Draw the instruction format for direct and indirect address.*
5. *Which are differences among indirect address, direct address, effective address and target add... Draw necessary figures. Also explain instruction code and operation code.*

GTU : Dec.-05, Marks

6. *Instruction format of a basic computer system.*

GTU : Dec.-05, June-06, Marks

7. *Explain in detail indirect address, Direct address, effective address and target address.*

GTU : Dec.-06, Marks

3.2 Computer Registers

GTU : Dec.-02,05,06, June-05,06, May-11, Winter-

Computer executes instructions which are stored sequentially in a computer mem... These instructions are fetched, decoded and then executed in a proper sequenc... perform the task of fetching the instruction, computer uses program counter... Instruction Register (IR). We have also seen that MAR and MDR registers are use... handle the data transfer between the main memory and the processor. The Temp... Register (TR) is used for holding temporary data during the processing. The proces... that supports direct input and output operations may have registers called I... (Input Register) and OUTR (Output Register). The input register holds the... character and output register holds the character to be sent. Some processors use... Accumulator register (AC) to hold the one operand for the Arithmetic Logic Unit (AL... The Table 3.2.1 summarize the various registers and their functions.

Register	Symbol used	Number of bits	Function
Data Register or Memory Data Register	DR or MDR	16	Holds the data read from memory or to be written in the memory.
Address Register or Memory Address register	AR or MAR	12	Holds the address for the memory.
Accumulator Register	AC	16	Holds the operand for the ALU and can be used as a general-purpose registers.
Instruction Register	IR	16	Holds the instruction code.
Program Counter	PC	12	Holds the address of the next instruction to be executed.
Temporary Register	TR	16	Holds the temporary data.
Input Register	INPR	8	Holds the data read from input device.
Output Register	OUTR	8	Holds the data to be sent to the output device.

Table 3.2.1 Basic computer registers

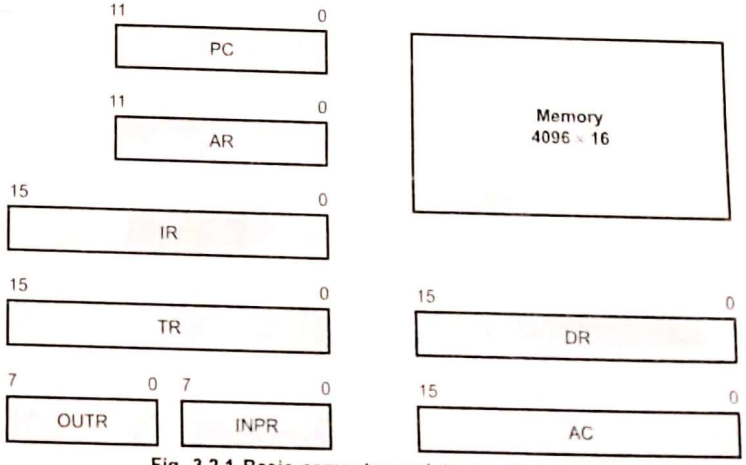


Fig. 3.2.1 Basic computer registers and memory

Common bus system

The Fig. 3.2.1 shows the basic computer that has eight registers, a memory unit and a control unit. In this Fig. 3.2.1, the data path for transferring information from register to register or between register and memory is not shown. The computer uses a common bus system for transferring information from register to register or between registers and memory instead of one to one connections to avoid excessive connections.

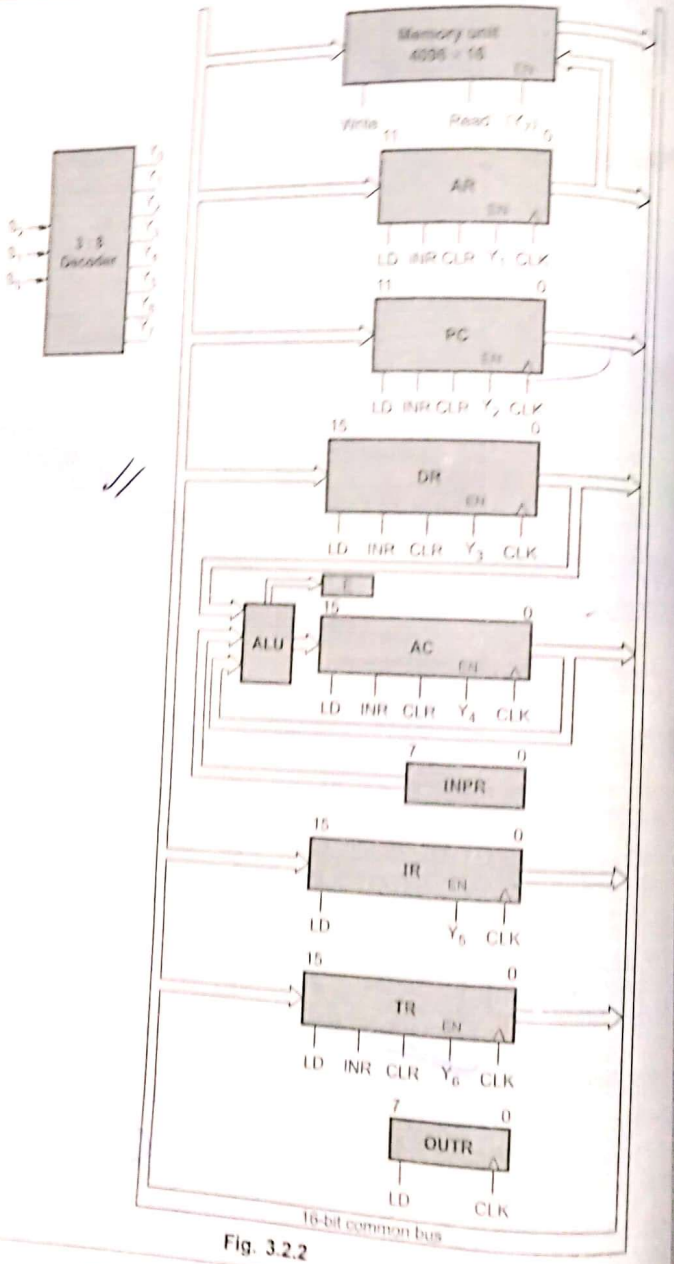


Fig. 3.2.2

The Fig. 3.2.2 shows the connections of the registers and a memory to the common bus. The control inputs LD (Load) and EN (Enable) are used to load data into register from common bus and to send data from register on to the common bus, respectively. The 3 : 8 decoder is used to select the specific register whose data to be sent on the common bus. The registers AR, PC, DR and AC have a CLR (Clear) and INR (Increment) inputs along with LD and OE inputs. These registers are equivalent to a binary counter with parallel load and synchronous clear. Register IR is used only to hold instruction read from memory so it does not have INR and CLR control signals.

Registers DR, AC, IR and TR are 16-bit registers. On the other hand, AR and PC are 12-bit registers, thus they hold a 12-bit memory address. When the contents of AR or PC are sent to the 16-bit common bus, the four most significant bits are set to 0s. When AR or PC receive information from the bus, only the 12 least significant bits are transferred into the register.

INPR and OUTR are the 8-bit registers and they communicate with the eight least significant bits in the bus. INPR receives a character from an input device which is then transferred to AC register, thus it serves as an information provider. On the other hand, OUTR receives a character from AC and delivers it to an output device.

AR register provides address to the memory and data input and data output of memory is connected to the common bus.

The ALU has three 16-bit input sources and one 16-bit destination, i.e. AC register. The three 16-bit input sources are AC, DR and INPR registers. The ALU performs arithmetic and logic micro-operations and result is transferred to the AC register. Thus AC gets input from ALU.

In this configuration, the contents of any register can be applied onto the bus and an operation can be performed in the ALU during the same clock cycle.

Review Questions

1. *Write a short note on computer registers.*
2. *Explain the computer bus system.*
3. *List all registers of basic computer. What is their usefulness ? What are their sizes ? Explain, with diagram, connection of registers with common bus.* **GTU : Dec.-02, June-08, Marks 8**
4. *How the basic computer registers are connected to a common bus ? Explain with diagram. Why AC input is not directly connected to common bus? Why IR does not have INR and CLR control signals ?* **GTU : June-05, Marks 8**
5. *Write a short note on common bus system.* **GTU : Dec.-05, 06, Marks 8**
6. *Draw neat and clean block diagram to connect general register set with ALU.* **GTU : June-06, Marks 5**
7. *Draw a block diagram of a computer's CPU showing all the basic building blocks such as program counter, accumulator, address and data registers, instruction register, control unit etc. and describe how such an arrangement can work as a computer, if connected properly to memory, input / output etc.* **GTU : May-11, Marks 7**
8. *List and explain functionalities of the basic computer registers.* **GTU : Winter-13, Marks 7**



I = 0 : Direct addressing mode
I = 1 : Indirect addressing mode
Opcode : It can have value from 000 through 110
since there are 7 memory reference instructions

Fig. 3.3.1 Memory reference instruction format

Table 3.3.1 shows the memory reference instructions and their instruction cod

Mnemonic	Description	Instruction code in Hex	
		I = 0	I = 1
AND	Logically ANDs the contents of specified memory location and AC AC ← AC × M [AR]	0 X X X	8 X X X
ADD	Adds the contents of specified memory location with AC AC ← AC + M [AR]	1 X X X	9 X X X
LDA	Loads the contents of specified memory location in AC AC ← M [AR]	2 X X X	A X X X
STA	Store the contents of AC in the specified memory location M [AR] ← AC	3 X X X	B X X X

3 lists the I/O

Mnemonic	Description
INP	Load a character in AC register from input port.
OUT	Send a character to output port from AC register.
SKI	Skip next instruction if input flag = 1.
SKO	Skip next instruction if output flag = 1.
ION	Enable interrupt, IEN ← 1.
IOF	Disable interrupt, IEN ← 0.

(0 through 15 in decimal). These four bits are decoded by 4 : 16 decoder to generate timing signals from 0 through 15. The I bit from IR is also connected to the combinational control circuitry. Therefore, combinational control circuitry takes the input from 3 : 18 decoder, 4 : 16 decoder, I bit and bits 0 through 11 to generate control signals.

Along with the clock input Sequence Counter (SC) has two other inputs : Clear and increment. Initially, sequence counter is cleared by activating clear input. Once the SC is cleared, clear signal is negated and SC is incremented with every positive clock transition to produce the sequence of timing signals T_0, T_1, T_2, T_3 and so on. If SC is not cleared, the timing signal will continue with T_4, T_5, up to T_{15} and back to T_0. However, if SC is cleared in between, a new sequence will start from T_0, T_1 and so on, as shown in the Fig. 3.4.1 (b) (Fig. 3.4.1. (b) on previous page.)

As shown in the Fig. 3.4.1 (b) the SC is cleared to 0 in T_4 if decoder output D_2 is active (logic 1). This can be expressed symbolically as

$$D_2T_4 : SC \leftarrow 0$$

Review Questions

1. *What is hardwired control ?*
2. *What is microprogrammed control ?*
3. *Draw the block diagram of control unit of basic computer. Explain in detail with control timing diagrams.* **GTU : June-05, Marks 8**
4. *Explain showing a basic block diagram, how the control unit of a CPU can be designed using hardwired control.* **GTU : May-11, Marks 7**

3.5 Instruction Cycle

GTU : Jan-07,13, Dec.-11, May-12,13, Winter-12,13, Summer-13,14

An instruction cycle involves three subcycles,

- Fetch
- Decode
- Execute.

Fig. 3.5.1 shows the basic instruction cycle.

The fetch phase reads the next instruction from memory into the CPU.

The decode phase interprets the opcode by decoding it.

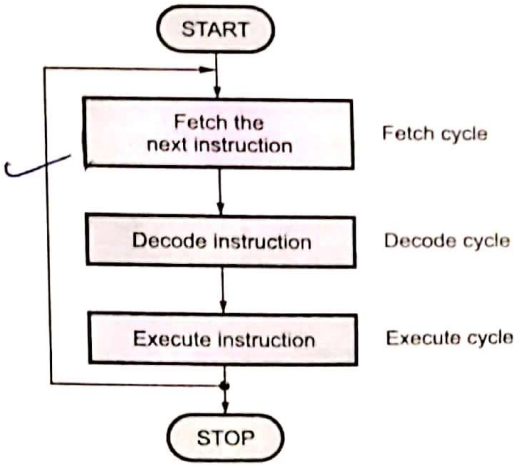


Fig. 3.5.1 Basic instruction cycle

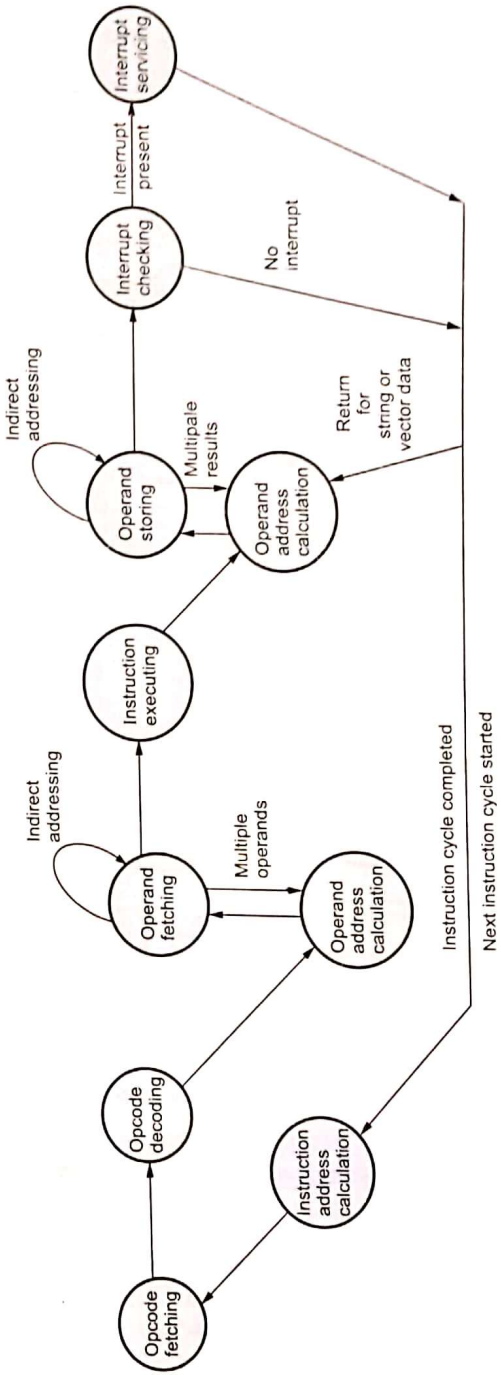


Fig. 3.5.3 Instruction cycle state diagram

Decode cycle

At T_2, decoding of instruction is done.

Register transfer statement for decode cycle is :

$T_2 : D_0, \ldots D_7 \leftarrow$ Decode IR (12-14), AR $\leftarrow$ IR (0-11), I $\leftarrow$ IR (15).

In T_2 : 1. Bits 12-14 from IR are decoded using 3 : 8 decoder.

2. Bits 0-11 from IR are loaded into the AR.

3. Bit 15 of IR is loaded into the addressing mode (I).

Determination of type of instruction

The Fig. 3.5.5 shows how the control circuitry determines the type of instruction after the decoding. As shown in the Fig. 3.5.5, if decoder output $D_7 = 0$, it is memory reference instruction, otherwise, it is register reference or I/O instruction. According to D_7 and I bits, different instructions are executed as listed in Table 3.5.1.

D_7	I	Instruction executed
0	0	Memory reference instruction with a direct address
0	1	Memory reference instruction with an indirect address
1	0	Register reference instruction
1	1	I/O instruction

Table 3.5.1

In case of indirect address, it is necessary to read the effective address from memory. The register transfer statement for this operation is :

$$AR \leftarrow M[AR]$$

Register transfer statements associated with timing signal T_3 are as follows :

$D_7 I T_3$: No operation

$D_7 I T_3$: $AR \leftarrow M[AR]$

$D_7 I T_3$: Execute a register reference instruction

$D_7 I T_3$: Execute an input-output instruction

Register transfer statement associated with

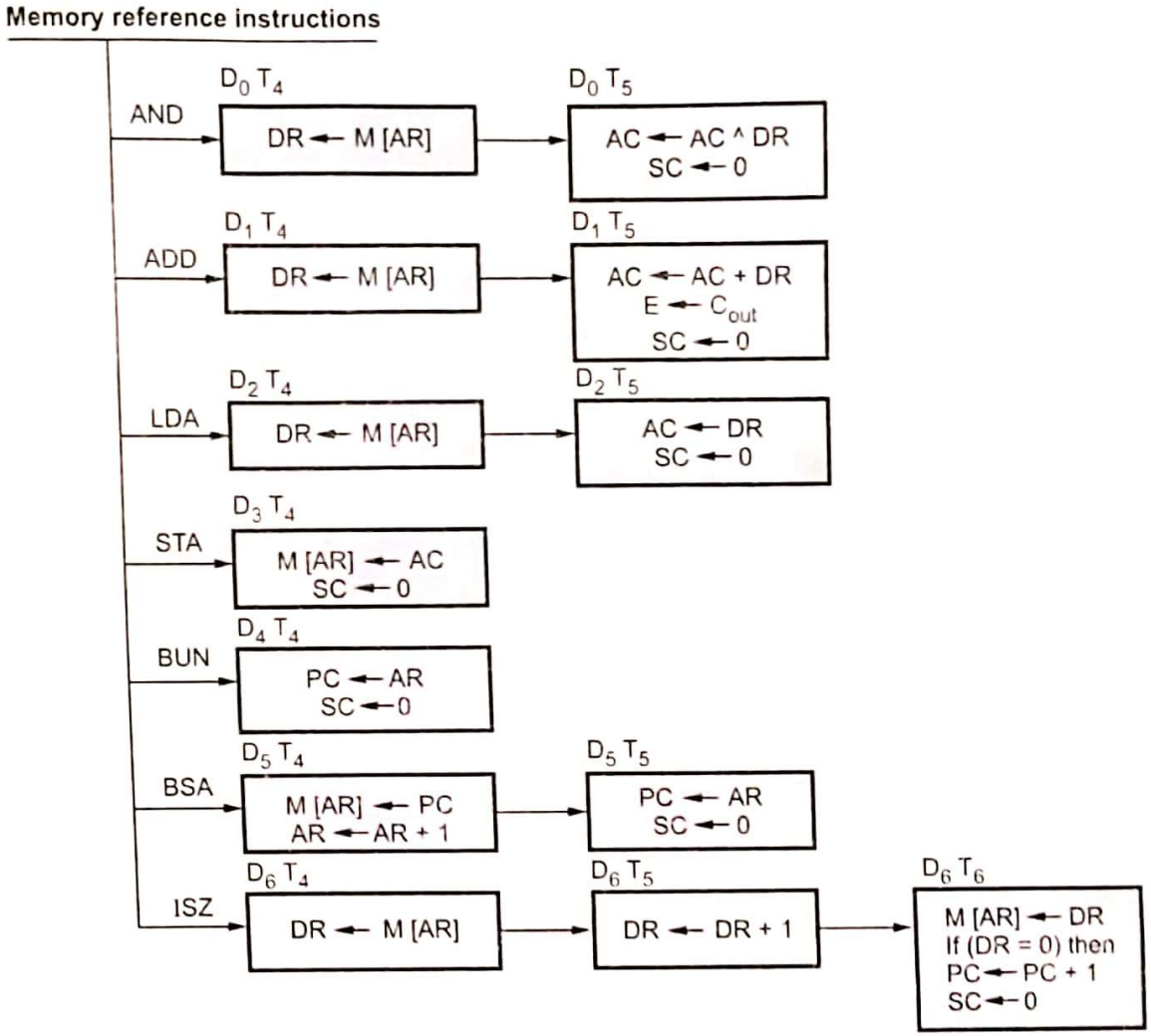
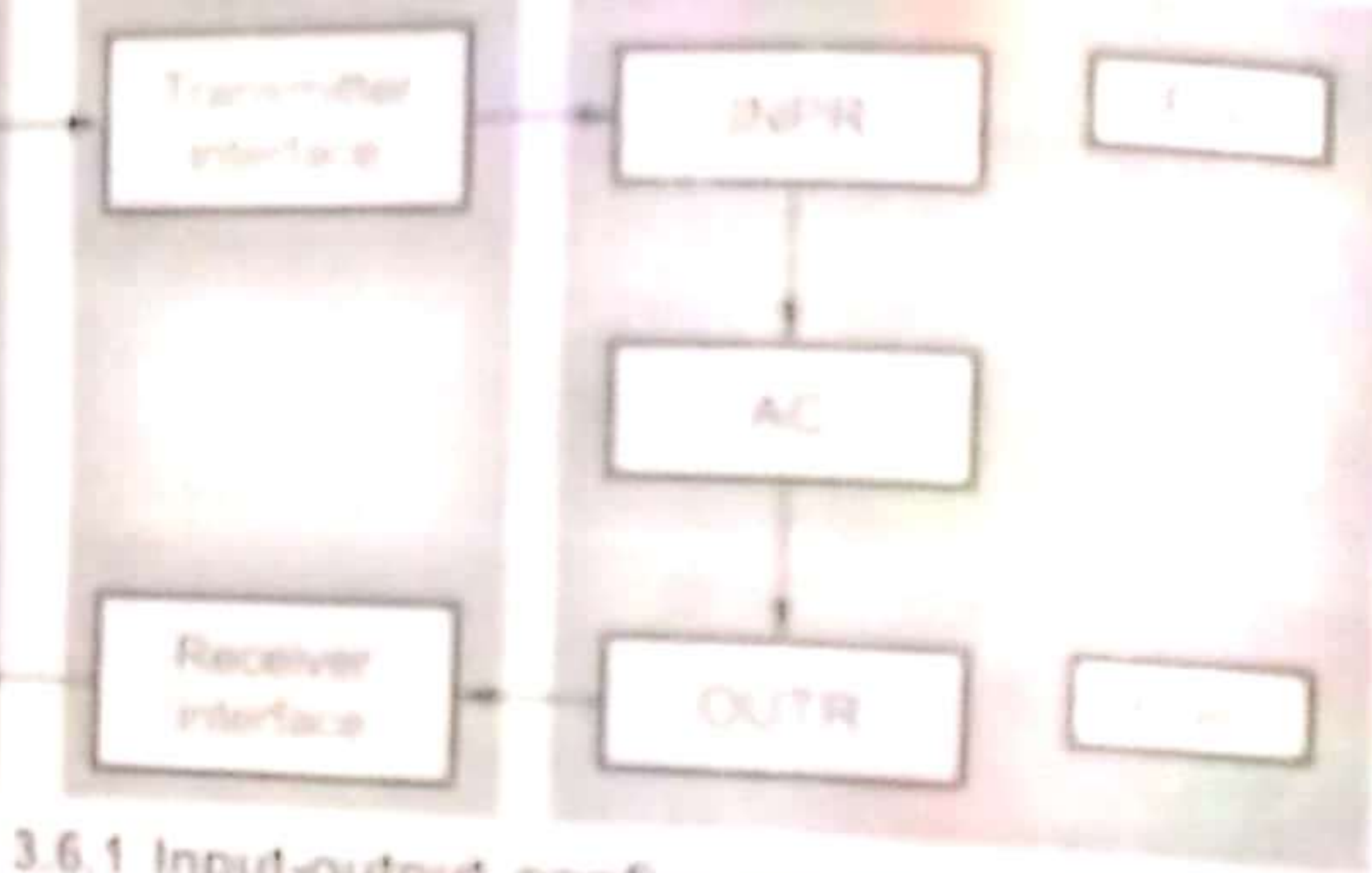


Fig. 3.5.6 Flowchart for memory reference instructions

Review Questions

1. *Write a note on instruction cycle.*
2. *Explain the implementation of register transfer instructions for fetch cycle.*
3. *Draw the flowchart for memory reference instructions.*
4. *Draw the flowchart for instruction cycle and briefly explain it.*



3.6.1 Input-output configuration

d with 1-bit R.J flag. The flag bit is set to 1 whe
available in the INPR and is cleared to 0 whe
pted by the computer, i.e. it is stored in the AC. A
INPR cannot be cleared by

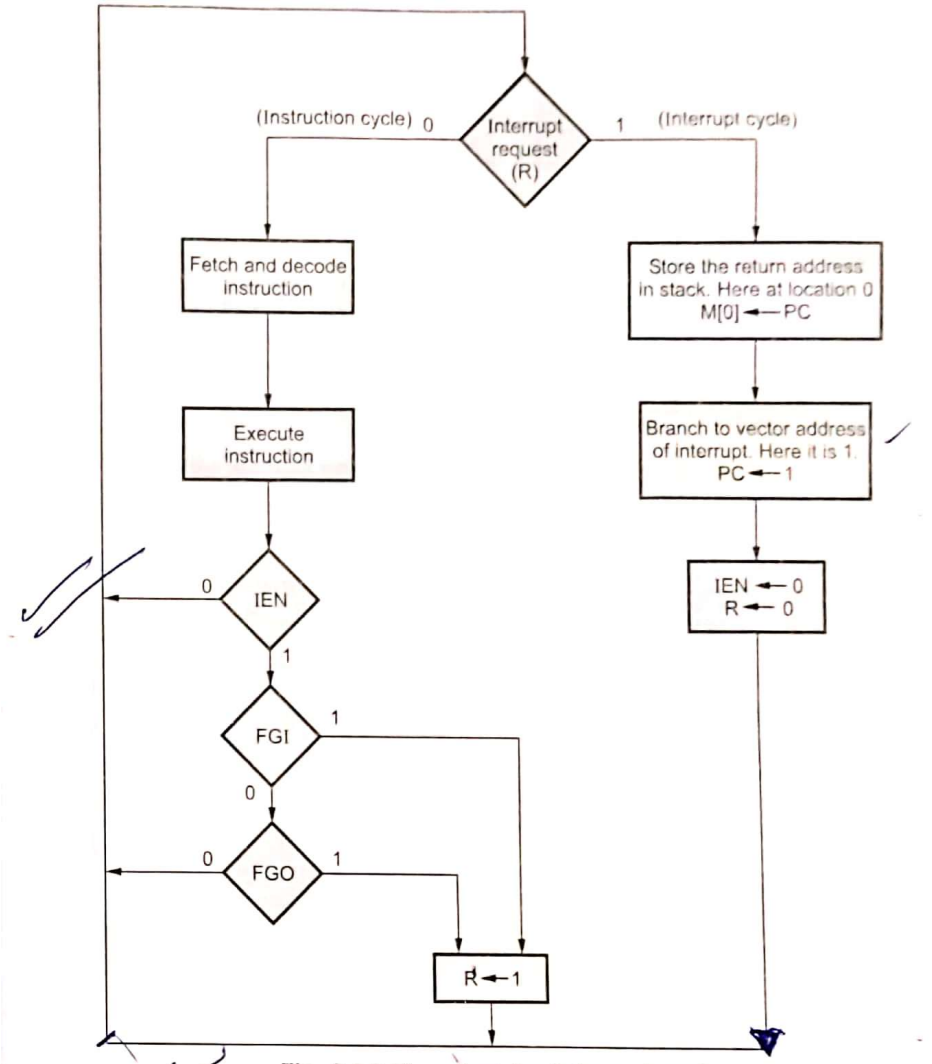


Fig. 3.6.3 Flowchart for interrupt cycle

ıterrupt cycle

In the interrupt cycle, first return address is stored in the memory stack or in a ɔecific memory location. Here we use memory location 0 to store the return address. lere, the vector location for interrupt is 1. Thus PC is loaded with 1 and, IEN and R are eared to disable further interrupts until current interrupt request is processed.

The Fig. 3.6.4 shows how the interrupt cycle is executed. Branch instruction at ɔcation 1 causes the program to transfer to the input-output service program at address 500. At the end of input-output service program IEN flag is enabled to enable further ıterrupts and at last indirect BUN instruction is executed to place return address from ɔcation 0 into the PC.

T_0 : AR ← 0 TR ← PC	AR is cleared and contents of PC (return address) are transferred to temporary register TR.
T_1 : M [AR] ← TR PC ← 0	Return address in TR is stored at memory location 0 and PC is cleared.
T_2 : PC ← PC + 1 IEN ← 0 R ← 0 SC ← 0	Increment PC to point to the interrupt vector address. IEN, R and SC are cleared.

Review Questions

1. *Draw and explain the input-output configuration of a basic computer.*
2. *Draw and explain the flowchart for interrupt cycle.*
3. *Explain the interrupt cycle execution with the help of diagram.*
4. *Write a short note on interrupt.* **GTU : Dec.-05, Ma**
5. *List steps that are performed by CPU before execution and after execution of interrupt routine.* **GTU : June-06, Ma**
6. *Write a short note on interrupt and interrupt cycle.* **GTU : Dec.-06, Ma**
7. *Explain program interrupts. Explain clearly, discussing the role of stack, PSW and return interrupt instruction, how interrupts are implemented on computers.* **GTU : May-11, Ma**
8. *What are main types of interrupts? Explain each type clearly.* **GTU : May-11, Ma**
9. *Explain input-output instructions.* **GTU : Summer-14, Ma**

CMA	$D_7 \bar{I} T_3$ IR(9) : $AC \leftarrow \overline{AC}$, $SC \leftarrow 0$
CME	$D_7 \bar{I} T_3$ IR(8) : $E \leftarrow \bar{E}$, $SC \leftarrow 0$
CIR	$D_7 \bar{I} T_3$ IR(7) : $AC \leftarrow$ shr AC, $AC(15) \leftarrow E$, $E \leftarrow AC(0)$, $SC \leftarrow 0$
CIL	$D_7 \bar{I} T_3$ IR(6) : $AC \leftarrow$ shl AC, $AC(0) \leftarrow E$, $E \leftarrow AC(15)$, $SC \leftarrow 0$
INC	$D_7 \bar{I} T_3$ IR(5) : $AC \leftarrow AC + 1$, $SC \leftarrow 0$
SPA	$D_7 \bar{I} T_3$ IR(4) : If $(AC(15) = 0)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$
SNA	$D_7 \bar{I} T_3$ IR(3) : If $(AC(15) = 1)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$
SZA	$D_7 \bar{I} T_3$ IR(2) : If $(AC = 0)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$
SZE	$D_7 \bar{I} T_3$ IR(1) : If $(E = 0)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$
HLT	$D_7 \bar{I} T_3$ IR(0) : $S \leftarrow 0$, $SC \leftarrow 0$

Table 3.7.1 (c) Control functions for register reference instructions

Input-output instructions	Control functions
INP	$D_7 I T_3$ IR(11) : $AC(0-7) \leftarrow INPR$, $FGI \leftarrow 0$, $SC \leftarrow 0$
OUT	$D_7 I T_3$ IR(10) : $OUTR \leftarrow AC(0-7)$, $FGO \leftarrow 0$, $SC \leftarrow 0$
SKI	$D_7 I T_3$ IR(9) : If $(FGI = 1)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$
SKO	$D_7 I T_3$ IR(8) : If $(FGO = 1)$ then $(PC \leftarrow PC + 1)$, $SC \leftarrow 0$
ION	$D_7 I T_3$ IR(7) : $IEN \leftarrow 1$, $SC \leftarrow 0$
IOF	$D_7 I T_3$ IR(6) : $IEN \leftarrow 0$, $SC \leftarrow 0$

Table 3.7.1 (d) Control functions for input/output instructions

Review Question

1. *Draw the complete flowchart of the instruction cycle.*

3.8 Design of Basic Computer

GTU : Dec.-10

In basic computer seen in section 3.2 we have following hardware components.

1. **Memory unit :** 4096 × 16, i.e. 4096 words of 16-bit each
2. **Nine register :** AR, PC, DR, AC, INPR, IR, TR, OUTR and SC.
3. **Seven flip-flops :** I, S, E, R, IEN, FGI and FGO
4. **Two decoders :** Operation decoder (3 : 8) and timing decoder (4 : 16)

th statement increments AR by 1. Thus CLR and INR signals must be enabled to ecute forth and fifth statements, respectively. With this information, we can form olean expressions for LD, CLR, and INR control signals as given below :

$$LD\ (AR) = \bar{R}\,T_0 + \bar{R}\,T_2 + \bar{D}_7\,I\,T_3$$

$$CLR\ (AR) = R\,T_0$$

$$INR\ (AR) = D_5\,T_4$$

The Fig. 3.8.1 shows the implementation of above Boolean expressions.

Using similar procedure we can derive control signals for other registers, read and rite signals for memory. The statements for memory read operation are as follows :

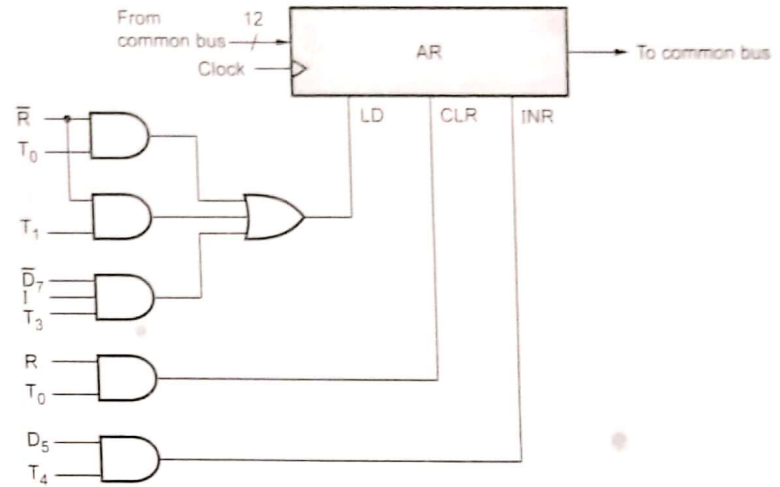


Fig. 3.8.1 Generation of control signals for AR

$\bar{R}\,T_1$: IR ← M [AR] ; During fetch

$\bar{D}_7\,I\,T_3$: AR ← M [AR] ; Indirect data transfer

D_0T_4 : DR ← M [AR] ; Loading operand for AND operation

D_1T_4 : DR ← M [AR] ; Loading operand for ADD operation

D_2T_4 : DR ← M [R] ; Loading the contents of memory into AC

D_6T_4 : DR ← M [R] ; Loading the operand for increment operation

Thus, Boolean expression for read operation is

$$Read = \bar{R}\,T_1 + \bar{D}_7\,I\,T_3 + (D_0 + D_1 + D_2 + D_6)\,T_4$$

Similarly, the statements for memory write operations are as following :

$R T_1$: $M[AR] \leftarrow TR$; Storing contents of PC during interrupt cycle

$D_3 T_4$: $M[AR] \leftarrow AC$; Storing the contents of AC

$D_5 T_4$: $M[AR] \leftarrow PC$; Storing the contents of PC

$D_6 T_4$: $M[AR] \leftarrow DR$; Storing result after increment

Thus, Boolean expression for write operation is

$$\text{Write} = R T_1 + (D_3 + D_5) T_4 + D_6 T_6$$

The Fig. 3.8.2 shows the implementation of read and write signals for memory

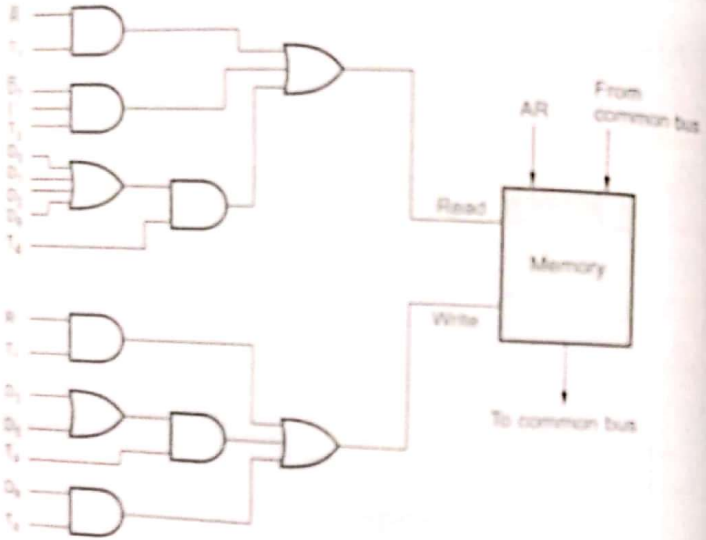


Fig. 3.8.2 Generating of read and write control signals for memory

Control of IEN flip-flop

Referring Table 3.7.1, we have statement which changes IEN are as follows :

$D_7 I T_3 B_7$: $IEN \leftarrow 1$

$D_7 I T_3 B_6$: $IEN \leftarrow 0$

$R T_2$: $IEN \leftarrow 0$

With these statements, the IEN flip-flop can be implemented using JK flip-flop as shown in the Fig. 3.8.3.

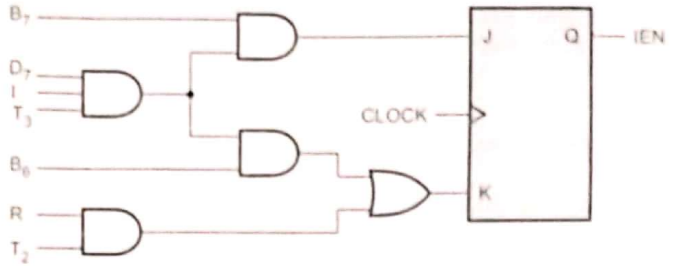


Fig. 3.8.3 Generation control inputs for IEN

Inputs								Outputs			Register selected for bus
Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7	S_2	S_1	S_0	
1	0	0	0	0	0	0	0	0	0	0	None
0	1	0	0	0	0	0	0	0	0	1	AR
0	0	1	0	0	0	0	0	0	1	0	PC
0	0	0	1	0	0	0	0	0	1	1	DR
0	0	0	0	1	0	0	0	1	0	0	AC
0	0	0	0	0	1	0	0	1	0	1	IR
0	0	0	0	0	0	1	0	1	1	0	TR
0	0	0	0	0	0	0	1	1	1	1	Memory

Table 3.8.2 Truth table of encoder for bus selection circuit

As shown in the Fig. 3.2.2, the sharing of common bus between various registers and memory is controlled by the selection signals S_2, S_1 and S_0. Depending on the enable inputs particular register or memory is connected to the common bus, as shown in the Table 3.8.2. Table 3.8.2 gives the status of selection signals for enable inputs and corresponding register/memory selected for bus. From the Table 3.8.2, the Boolean functions for the encoder can be given as,

$$S_2 = Y_4 + Y_5 + Y_6 + Y_7$$

$$S_1 = Y_2 + Y_3 + Y_6 + Y_7$$

$$S_0 = Y_1 + Y_3 + Y_5 + Y_7$$

To determine the logic for each ... necessary to find the ... functions that connect the corresponding register or memory onto the common ... read operation, i.e., for the transfer of data from particular register or memory ... common bus. For example, memory is selected for read operation when ... Therefore, the Boolean function for Y_7 is the same as the one obtained earlier ... operation.

$$\text{Read} = Y_7 = \bar{R}\,T_1 + \bar{D}_7\,I\,T_3 + (D_0 + D_1 + D_2 + D_6)\,T_4$$

Referring Table 3.7.1, the statements having AR as a source are :

D_4T_4 : $PC \leftarrow AR$

D_5T_5 : $PC \leftarrow AR$

Therefore, the Boolean function for Y_1 is

$$Y_1 = D_4\,T_4 + D_5\,T_5$$

Similarly, we can derive the boolean functions for all the enable inputs and implementing, and connecting them to the encoder we can generate S_2, S_1 and S_0 select signals, as shown in the Fig. 3.8.4.

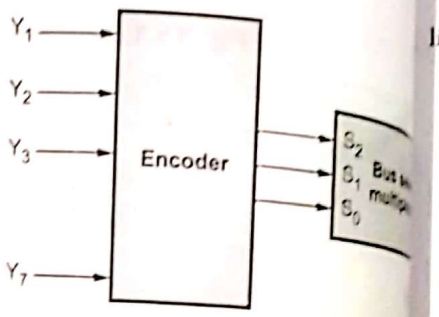


Fig. 3.8.4 Generating bus select inp...

Review Questions

1. *List the hardware component of basic computer.*
2. *Give the control functions for any four instructions of basic computer.*
3. *Explain the generation of control signal for AR.*
4. *Explain the generation of read and write control signals for memory*
5. *Derive the control gate structure associated with the Address Register (AR) in the basic com...*

GTU : Dec.-10, Ma...

3.9 Design of Accumulator Logic

GTU : Dec.-03, 06, Ma...

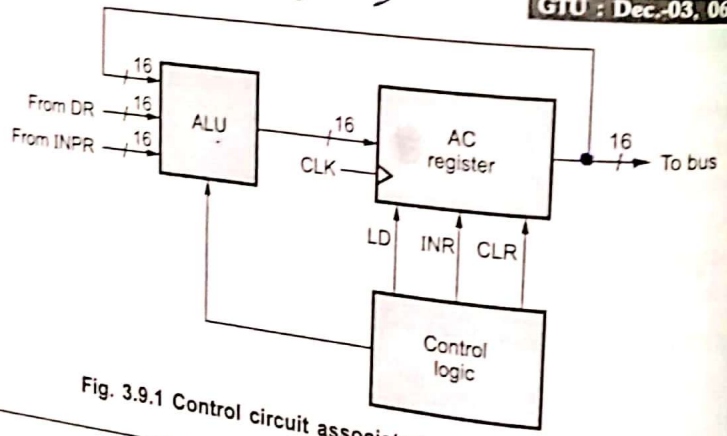


Fig. 3.9.1 Control circuit associated with AC register

The Fig. 3.9.1 shows the control circuit associated with the AC register ... the Fig. 3.9.1, the ALU has three sources of inputs : one from AC register ... DR register and third from INPR register. The output of ALU is connected ... for AC register. The AC register has three control inputs : LD, INR and ... from the control logic.

Let us see the register transfer statements associated with AC regis... listed in Table 3.9.1.

Sr. No.	Register transfer statement	Operation
1.	$D_5\,T_5$: $AC \leftarrow AC \wedge DR$	AND AC with DR
2.	$D_7\,T_5$: $AC \leftarrow AC + DR$	Add AC and DR
3.	$D_2\,T_5$: $AC \leftarrow DR$	Load AC from DR
4.	$p\,B_{11}$: $AC\,(0-7) \leftarrow INPR$	Load AC from INPR
5.	rB_9 : $AC \leftarrow \overline{AC}$	Complement AC
6.	$r\,B_7$: $AC \leftarrow Shr\ AC,\ A(15) \leftarrow E$	Shift right AC
7.	$r\,B_6$: $AC \leftarrow Shl\ AC,\ AC(0) \leftarrow E$	Shift left AC
8.	$r\,B_{11}$: $AC \leftarrow 0$	Clear AC
9.	$r\,B_5$: $AC \leftarrow AC + 1$	Increment AC

Table 3.9.1 Register transfer statements associated with AC regis...

3.9.1 Control of AC Register

From the register transfer statements we can derive the control inputs f... as shown in the Fig. 3.9.2. For first seven operations, we need, Load (LD) i... be activated. For 8th operation, we need Clear (CLR) input to be activated ... operation, we need increment (INR) input to be activated. (Refer Fig. 3.9.2 ...

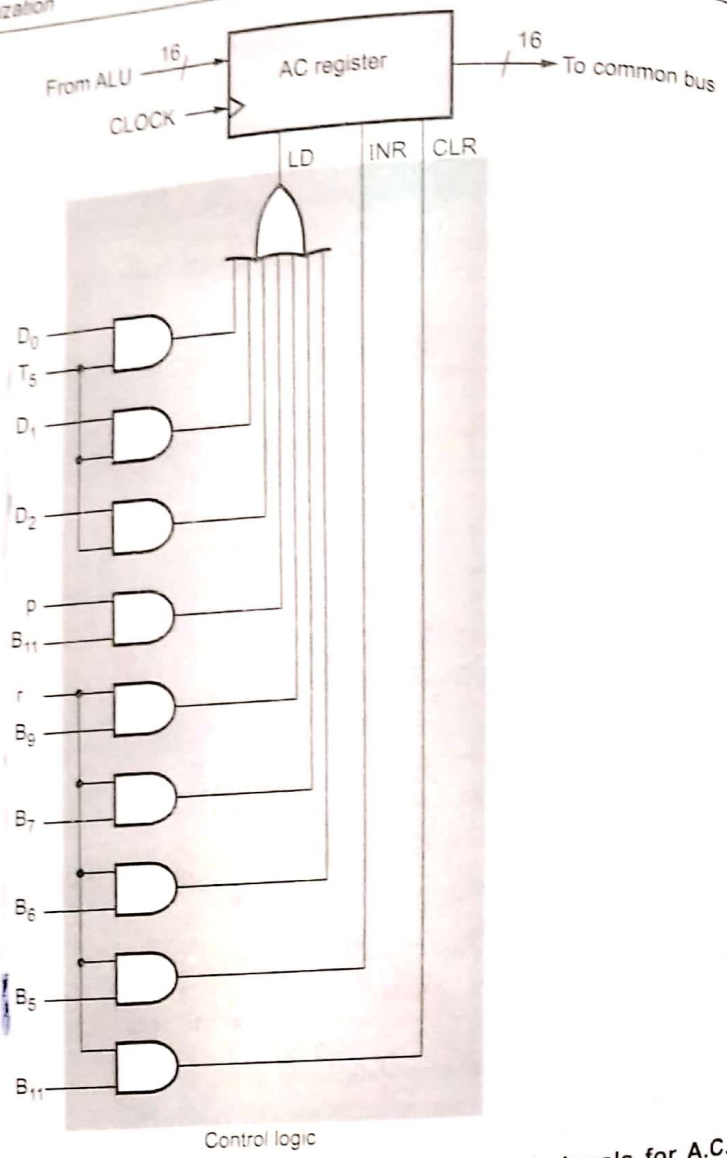


Fig. 3.9.2 Control circuit for generation of control signals for A.C.

3.9.2 Control of ALU

If we divide ALU into 16 stages, each stage will have three 1-bit inputs : AR(i), I and INPR (i) and one 1-bit output O(i). Similarly, one stage of AC register will have 1-bit input I(i) and one 1-bit output, A(i). One bit AC register stage can be implemented using JK register as shown in the Fig. 3.9.3. When LD input is enabled, one stage AC is loaded with its current input. Current input for one stage AC is derived from one-stage ALU. Depending on the operation performed one-stage ALU generates input for AC as shown in the Fig. 3.9.3. The ADD operation is achieved by using

4 Programming the Basic Computer

Syllabus

Introduction, Machine language, Assembly language, Assembler, Program loops, Programming arithmetic and logic operations, Subroutines, I-O programming.

Contents

4.1 Introduction		
4.2 Machine Language	**June-06, Winter-13,**	Marks 7
4.3 Assembly Language	**Jan.-10, June-06, May-11, Dec.-11, Winter-12,**	Marks 7
4.4 The Assembler	**June-05,06, Dec.-09,10, Feb.-10, May-11,12, Winter-12,13, Summer-13,14,**	Marks 12
4.5 Program Loops	**Dec.-09**	Marks 7
4.6 Programming Arithmetic and Logic Operations	**Dec.-10,11, Summer-13,14**	Marks 7
4.7 Subroutines	**June-05, Dec.-03,05, Jan.-10, May-12,**	Marks 7
4.8 Input-Output Programming	**Dec.-09, May-12, Summer-13,**	Marks 7

4.1 Introduction

A total computer system consists of two major components, hardware and so... The physical parts that make up a computer which are interconnected electronic d... and which you can see and touch are called **'hardware'**. A computer cannot f... perform any operation on its own. The sequential instructions are to be given to p... any operation. The set of sequential instructions and data which performs the op... is called **'program'**. This program or collection of such programs is called **'softwa...** software controls and operates the hardware to get the desired output. Every acti... function of hardware is driven by software. The various aspects of computer sy... can be studied without knowing the details of computer hardware and its ope... Also, the computer hardware can be designed without a knowledge of its so... capabilities. As far as computer architecture is concerned, a knowledge of ... hardware and software is essential because software and hardware are comp... interdependent on each other. Software tells hardware what to do and how to do a... per the instructions of software, hardware performs the operations and accomplish... tasks. Hardware and software are like two sides of a coin. One cannot do a... without other.

The computer only understands the instructions in the binary form. Progra... nothing but the sequence of instructions coded in binary form. There are three lang... levels that can be used to write a program. They are,

1. Machine level programs.
2. Assembly language level programs.
3. High level language programs.

A program which has simply a sequence of the binary codes for the instructio... called **machine language program.** This binary form of the program is referred t... machine language because it is the form required by the machine. However, to wri... program in machine language, programmer has to memorize the thousands of bi... instruction codes. This task is difficult and error prone.

To make programming easier, usually programmers write programs in asse... language. They then translate the assembly language program to machine languag... that it can be loaded in memory and executed.

Another way of writing a program for a computer is with a high-level language, s... as Basic, Pascal, C or C++. The program statements used in such languages are c... more English like than those of assembly language. Each program statement ... high-level language represents one or more than one machine code instructions. ... program statements in the high-level language are converted into machine codes w... the help of **interpreter** or **compiler**. However, the ... ma...

GTU : June-06, Winter-1

4.2 Machine Language

A program is a set of instructions arranged in the specific sequence to perform a specific task. It directs the computer to perform a required data-processing task. We have seen that the program may be written using different types of programming languages. However, the computer can execute program only when it is represented internally in binary form. Programs written in any other language must be translated to the binary form before executing by the computer.

Following are the categories which may be used to write programs for a computer.

1. **Binary code :** The instructions and operands are represented in binary and stored in the same form in computer memory.
2. **Octal or hexadecimal code :** This is the same translation of the binary code to octal or hexadecimal representation.
3. **Symbolic code :** The instructions may include symbols (such as letters, numerals or special characters) for the operation part, the address part and other parts. Each such instruction can be translated into one binary coded instruction. A special program is used for this translation called an **'assembler'**.
4. **High-level programming languages :** English-like words are used to write programs in high-level language. However, each statement must be translated into a sequence of binary instructions before the program can be executed in the computer. The program that performs this translation is called **compiler**.

Relation between Binary and Assembly Languages

Now, consider the binary program given in Table 4.2.1. The first column in this table gives the address/memory location of each instruction or operand in binary. The second column lists the instructions or operands in binary form. This program can be stored in the indicated portion of memory and then executed by the computer starting from address 0000.

Address / Memory location (Binary)	Instruction code (Binary)			
0000	0010	0000	0000	0100
0001	0001	0000	0000	0101
0010	0011	0000	0000	0110
0011	0111	0000	0000	0001
0100	0000	0000	0101	0011
0101	1111	1111	1110	1001
0110	0000	0000	0000	0000

Table 4.2.1 Binary program to add two numbers

This program is written to add two numbers. The hardware of computer will execute the instructions and perform addition of two numbers. However, just by observing the instructions, it is difficult to understand what is to be achieved when this program is executed.

The computer can understand only machine language. That is, each instruction should be written in binary form. In this program, each instruction needs 16 bits. It is very tedious to write 16 bits for each instruction. The number of digits per instruction can be reduced by writing octal or hexadecimal equivalent of binary code. Thus, the instruction code can be reduced to six digits using octal equivalent and to four digits using hexadecimal equivalent of the binary code. Table 4.2.2 shows the hexadecimal equivalent of the binary code given by Table 4.2.2.

Address/Memory location (Hexadecimal)	Instruction code (Hexadecimal)
0000	2004
0001	1005
0002	3006
0003	7001
0004	0053
0005	FFE9
0006	0000

Table 4.2.2 Hexadecimal program to add two numbers

It is to be noted that each hexadecimal digit must be converted to its equivalent 4-bit binary number when the program is entered into the computer. Though the problem of writing too many digits for each instruction is solved, the problem of understanding what is to be achieved when the program is executed, still remained. This problem can be solved by using symbolic names of instructions instead of their binary or hexadecimal equivalent. Table 4.2.3 shows the symbolic operation codes for instructions used in Tables 4.2.1 and 4.2.2. Of course, the addresses/memory locations and operands remain in their hexadecimal value. The last column of comments is added in this table to explain function of each instruction.

Address/Memory location (Hexadecimal)	Instruction	Comments
0000	LDA 004	Load first operand into accumulator
0001	ADD 005	Add second operand to accumulator
0002	STA 006	Store sum at an address 0006
0003	HLT	Halt computer

0004	0053	First operand
0005	FFE9	Second operand (-ve)
0006	0000	Store sum here

Table 4.2.3 Program with symbolic operation codes and comments to add two n

The programs written with symbolic codes are preferable and easier to hand problem which is still remained, is that we do not know exactly the numeric location of operands while writing a program. If the operands are placed at a following the instructions and if the program-length is not known in adva address of the operand is not known until the end of the program is reached. this problem, hexadecimal addresses can be replaced by symbolic addresses hexadecimal operands can be replaced by equivalent decimal numbers since numbers are more familiar than their hexadecimal equivalents. The program wit changes is shown in Table 4.2.4. It is called assembly language program.

Instruction	Comments
ORG 0	; Origin of program is address 0
LDA A	; Load operand from location A
ADD B	; Add operand from location B
STA C	; Store sum in location C
HLT	; Halt computer
A, DEC 83	; Decimal operand
B, DEC-23	; Decimal operand
C, DEC 0	; Sum stored in location C
END	; End of symbolic program

Table 4.2.4 Assembly language program to add two numbers

The purpose of symbol ORG followed by a number is to specify an origin. T the address of the next instruction below it. The next three lines have sy addresses. Decimal operands are specified following the symbol DEC. The posi negative numbers may be used. However, they must be converted to binary signed-2's complement representation. The symbol END in the last line indicates th of the program. The symbols ORG, DEC and END are called 'pseudo instructions'.

Table 4.2.5 gives the equivalent C program to add two integer numbers.

```
# include <stdio.h>
main ( )
{
        int A = 83, B = - 23, C ;
        C = A + B ;
        printf ("\n sum = % d", C) ;
}
```

Table 4.2.5 Program in C to add two integer numbers

Example 4.2.1 *Write an ALP for swapping two numbers. Indicate the comments also.*

GTU : Winter-13, Marks 7

Solution :

```
ORG 0       ; Origin of program
LDA A       ; Read the first number
STA TEM     ; Store it in temporary memory location
LDA B       ; Read the second number
STA A       ; Store it in first's memory location
LDA TEM     ; Read the contents of temporary memory
            ; location i.e. previous contents of A memory location
STA B       ; Store it in second's memory location
END         ; End of symbolic program
```

Review Questions

1. *What is the relation between binary and assembly languages ?*
2. *What do you mean by symbolic operation code ?*
3. *With the help of suitable examples explain machine language, assembly language and high level language programs.*
4. *What is machine level language ?*

GTU : June-06, Marks 3

4.3 Assembly Language

GTU : June-06, Jan.-10, May-11, Dec.-11, Winter-12

Each programming language has a set of rules. Assembly language statements are usually written in a standard form and assembly language has its own unique syntactical structure, such as requiring upper case or lower case or requiring colons after label definitions. Almost every commercial computer has its own particular assembly language. The computer manufacturer provides manual in which rules for writing assembly language programs are documented.

A line of code is the basic unit of an assembly language program. Each language has its set of rules that specify the symbols that can be used and how they may be combined to form a line of code. Here we discuss the common features of assembly languages which are used for writing symbolic programs for the basic computer.

Rules of the Language

The assembly text is usually divided into three fields, separated by spaces and tabs. A format for a typical line from assembly language program can be given as

Label : Mnemonic, Operand1, Operand2 ; Comment.

The three fields are

1. **The label field** : The first field, which is optional, is the label field, used to ... a symbolic address. A label is an identifier that is assigned to the address ... first byte of the instruction in which it appears. The presence of a label is ... but if present, the label provides a symbolic name that can be used by ... instructions to branch to the instruction.

2. **The mnemonic/instruction field** : The second field is mnemonic/instruction ... is compulsory. It specifies a machine instruction or a pseudo instruction ... presence of the operands depends on the instruction. Some instructions ... operands, some have one and some have two. If there are two operands, ... separated by a comma.

3. **The comment field** : The last field is a comment field. It may be empty or ... include a comment. It begins with a delimiter such as the semicolon and ... to the end of the line. The comments are for our benefit. They tell us ... program is trying to accomplish. They are useful for explaining the progra... are helpful in understanding the step-by-step procedure taken by the p... They are used only for explanation and are neglected while translating prog... binary. Fig. 4.3.1 shows a typical 8085 assembly language instruction.

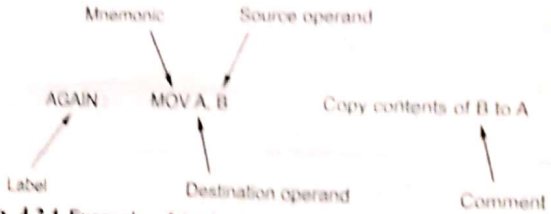


Fig. 4.3.1 Example of typical assembly language instruction

Symbolic Address

A symbolic address consists of one, two or three alphanumeric characters. ... character must be a letter and the next two may be letters or numerals. The progr... can choose the symbol arbitrarily. A symbolic address in the label field is termina... a comma so that it will be recognized as a label by the assembler.

The mnemonic or instruction field in an instruction may specify one of the fo... items.

1. A Memory Reference Instruction (MRI). The MRI instruction contains two or ... symbols separated by spaces. The first three letter symbol defines ... operation. The second symbol gives address. The third symbol may or may ... present. It is a letter I. So, if I is present, it is indirect address inst... otherwise, direct address instruction.

2. A register reference or input-output instruction (non-MRI). This type of instruction does not have an address part. It is recognized in the instruction field of a program by three letter symbols. The address of an operand is given by a symbol in the instruct field. It is necessary to define this address somewhere in the program as a label in the first column because computer has to translate an each assembly language instruction to a binary code.

3. A pseudo instruction with or without an operand.

A pseudo instruction is an instruction to the assembler giving information about some phase of the translation.

Four pseudo instructions recognized by the assembler are discussed below.

1. **ORG N** : Hexadecimal number N gives the address for the operand or the instruction listed in the following line.
2. **END** : The END symbol is placed at the end of the program. It informs the assembler that the program is terminated.
3. **DEC N** : This symbol tells the assembler how to convert the signed decimal number N to binary.
4. **HEX N** : This symbol tells the assembler how to convert the hexadecimal number N to binary.

An Example

Table 4.3.1 gives an example of an assembly language program. The first instruction in this program is ORG 100 which defines the origin of the program at memory location $(100)_{16}$. The next six instructions are machine instructions and last four are pseudo instructions. The column1 lists the symbolic addresses, which are labels. First three psedo instructions specify operands. The last psedo instruction indicates the end of the program.

```
        ORG 100       ; Origin of the program is at address 100H
        LDA SUB       ; Load subtrahend in the accumulator
        CMA           ; Complement accumulator
        INC           ; Increment accumulator
        ADD MIN       ; Add minuend to accumulator
        STA DIF       ; Store difference
        HLT           ; Halt computer
MIN,    DEC 83        ; Minuend
SUB,    DEC 23        ; Subtrahend
DIF,    HEX 0         ; Difference stored here
        END           ; End of the program
```

Table 4.3.1 Assembly language program for subtraction

Example 4.3.1 *Write an assembly level program for the following pseudocode.*

$SUM = 0$

$SUM = SUM + A + B$

$DIF = DIF - C$

$SUM = SUM + DIF$

GTU : Jan.-10, Marks 5

Solution :

```
ORG 100          ; Main program
CLA              ; Clear accumular
STA SUM          ; SUM = 0
LDA A            ; Load first operand
ADD SUM          ; Accumulator ← SUM + A
STA SUM          ; Store result in SUM
LDA B            ; Load second operand
ADD SUM          ; Accumulator ← SUM + B
STA SUM          ; Store result in SUM
LDA DIF          ; Load the value in DIF
SUB C            ; Accumulator ← DIF – C
STA DIF          ; Store result in DIF
ADD SUM          ; Accumulator ← SUM + DIF
STA SUM          ; Store the result, i.e. SUM = SUM + DIF
SUM, –
A,   –
B,   –
C,   –
DIF, –
END              ; End of symbolic program
```

Translation to Binary

As seen earlier, the computer can understand only machine language which is in the form of binary codes. So, the assembly language program is to be translated into machine language. A special program that performs this translation is called an **'assembler'**.

The translation of the assembly language program given in Table 4.2.4 into an equivalent binary code is shown in Table 4.3.2.

Address / Memory location (Hexadecimal)	Content (Hexadecimal)	Assembly language program
		ORG 100

END are not assigned a numerical location since they do not represent an instruction or operand.

After completing the first scan, we associate an address with each label. Then we form a table that defines the hexadecimal value of each symbolic address. Table 4.3.3 shows the address symbol table for the program shown in Table 4.3.2.

Address symbol	Hexadecimal address
MIN	106
SUB	107
DIF	108

Table 4.3.3 Address symbol table

During the second scan of the symbolic program, we obtain the address value of a memory-reference instruction by referring the address symbol table. For example, consider the line of code LDA SUB. During the second scan LDA SUB is translated by getting the hexadecimal value of LDA from Table 4.1.1 and the hexadecimal value of SUB from Table 4.3.3. Then we obtain a four-digit hexadecimal instruction by assembling these two parts.

We have discussed two scans of the symbolic program. An assembler program performs the translation of symbols to binary. The first scan is called the **'first pass'** and the second is called the **'second pass'**.

Review Questions

1. *What is assembly language programming ? Write rules for writing it.*
2. *Which are the three fields in an assembly language program? Explain them with example.*
3. *What is symbolic address ?*
4. *What is pseudo instruction ?*

Memory word	Symbol	Hexadecimal code	Binary representation
1	P L	50 4C	0101 0000 0100 1100
2	3	33 2C	0011 0011 0010 1100
3	L D	4C 44	0100 1100 0100 0100
4	A	41 20	0100 0001 0010 0000
5	S U	53 55	0101 0011 0101 0101
6	B	42 20	0100 0010 0010 0000
7	I CR	49 0D	0100 1001 0000 1101

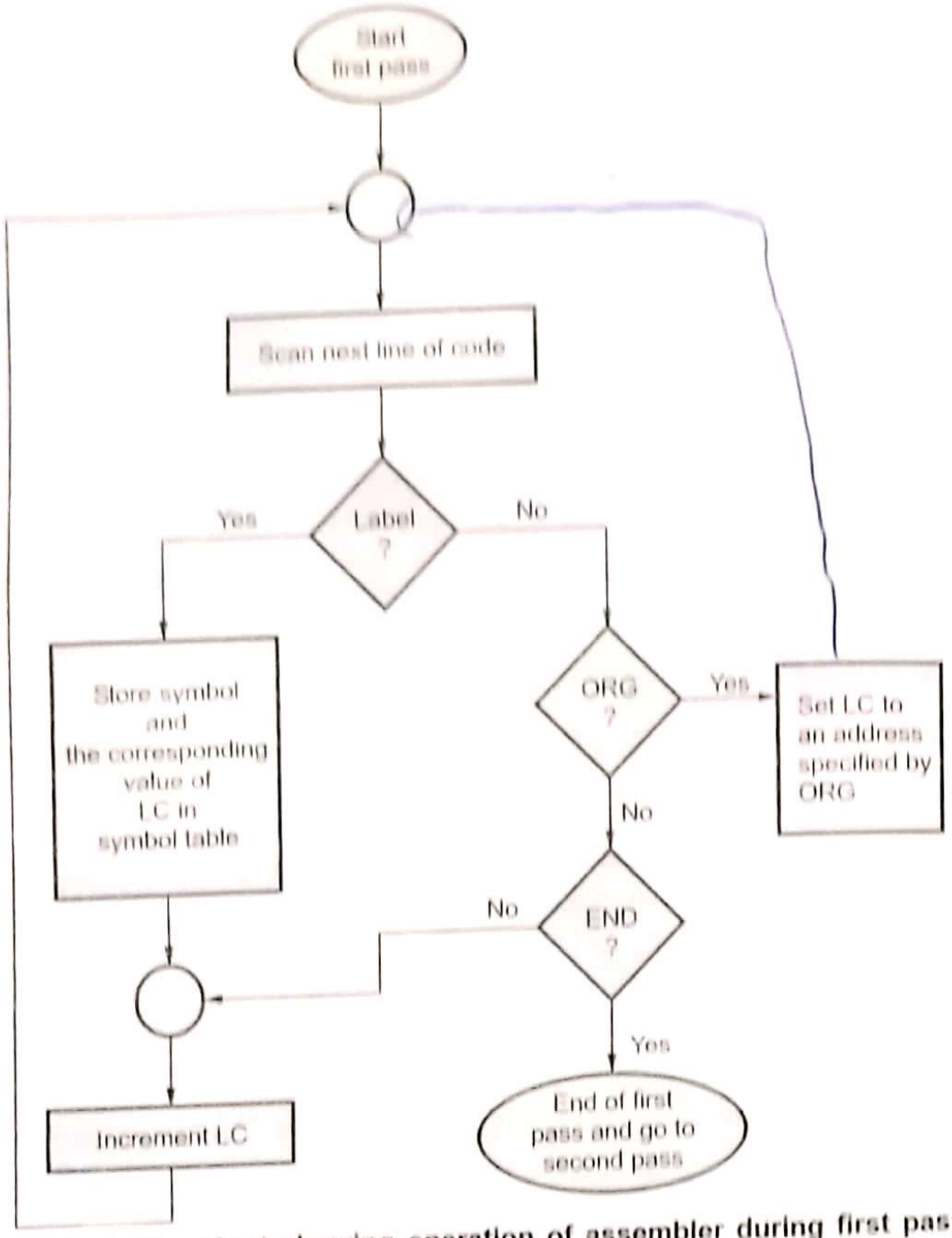
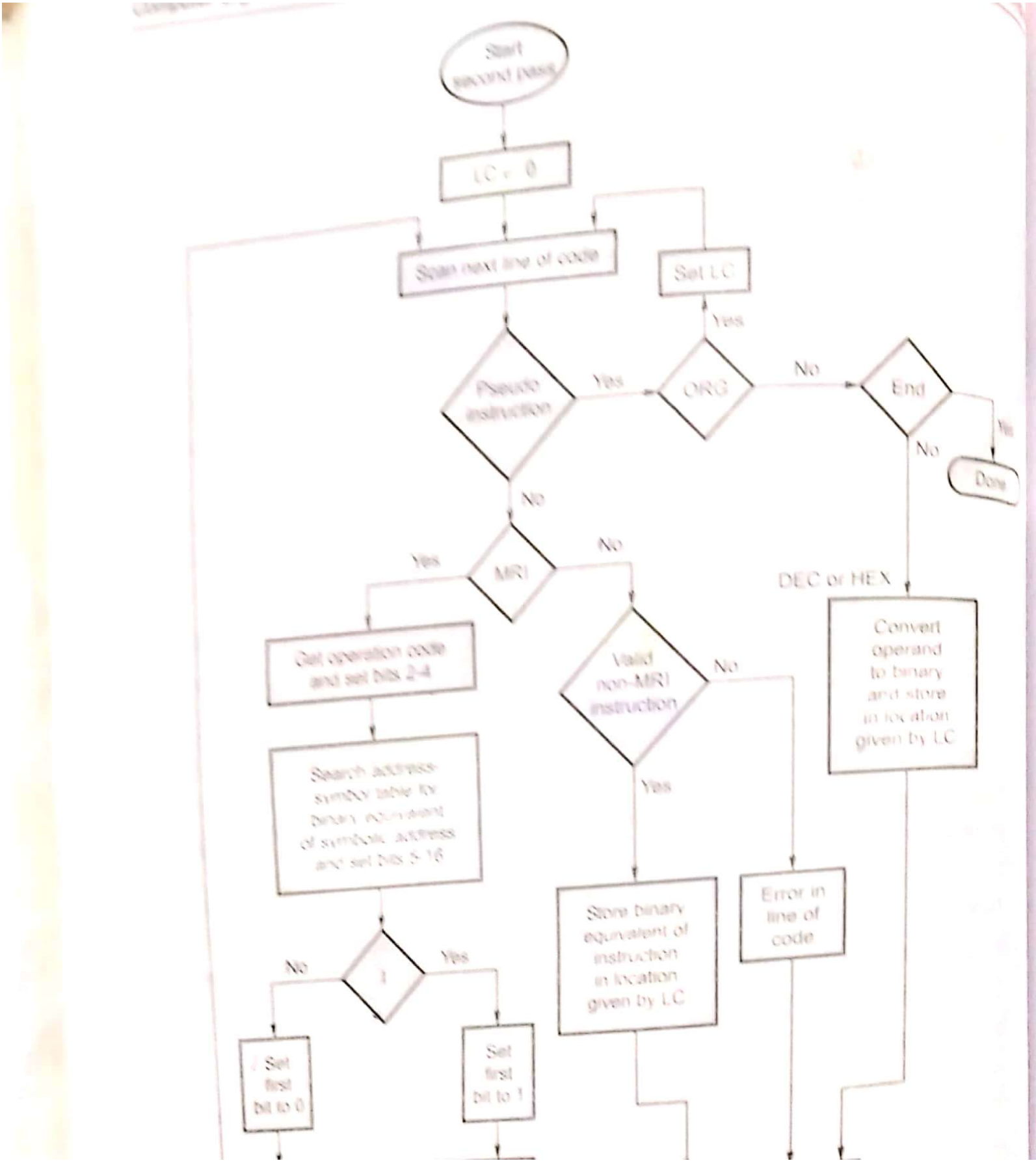


Fig. 4.4.1 Flowchart showing operation of assembler during first pass

MIN, SUB and DIF. These symbols represent 12-bit addresses equivalent to hexadecimal 106, 107 and 108 respectively.

4.4.2 Second Pass

During second pass, machine instructions are translated to binary by referring lookup tables. The entries of the tables are searched to determine whether a specific item matches one of the items stored in the table. This is called a table-lookup procedure. The symbol in the program must be available as an entry in the look-up tables. If it is not available, the symbol cannot be interpreted.


Start assembly pass
LC ← 0
Scan next line of code
Set LC
Yes
Pseudo instruction
Yes
ORG
No
End
Yes
Done
No
No
MRI
No
Yes
DEC or HEX
Get operation code and set bits 2-4
Valid non-MRI instruction
No
Convert operand to binary and store in location given by LC
Search address symbol table for binary equivalent of symbol address and set bits 5-16
Yes
Store binary equivalent of instruction in location given by LC
Error in line of code
I
No
Yes
Set first bit to 0
Set first bit to 1

3. What is address symbol table ?
4. Explain working of first pass of assembler of basic computer.
5. Explain working of second pass of assembler of basic computer.
6. Explain working of two-pass assembler with flow-chart.
7. Explain two-pass assembler with flow-chart.
8. Write a note on Error diagnostics by an assembler.
9. What is an Assembler ? Explain phases of it. **GTU : June-05, Marks**
10. Describe the first pass of assembler with the help of flowchart and show how symbol table is generated using an example. **GTU : June-06, Marks**
11. Explain the first pass of assembler with the help of flow chart and how symbol table is generated using an example. **GTU : Dec.-09, Marks**
12. Demonstrate the process of second pass of assembler using a suitable diagram. **GTU : Feb.-10, May-12, Summer-13,14, Marks**
13. What is the basic functionality of an assembler ? Explain its first pass. **GTU : Winter-12, Marks**
14. Explain the working of second pass assembler with its flowchart and its importance. **GTU : Winter-13, Marks**
15. Why do we need first pass and second pass in the process of assembly. Describe in brief what is done under each pass. **GTU : Dec-10, Summer-14, Marks**

GTU : May-11, Marks

4.5 Program Loops

GTU : Dec

When a group of sequence of instructions is executed many times in a program time with a different set of data, it is called a program loop.

A program in C that forms the sum of 1 to 30 integers using a loop is given below

```
# include <stdio.h>
main()
{   int n, sum = 0;
    for (n=1; n<=30; n++)
        sum = sum+n;          /* This statement executed 30 times */
    printf ("\nsum=%d", sum);
}
```

In the above program, a for loop is used. In this loop, n is initially set to 1 and compared with 30. Since n is less than 30, the statement sum = sum + n is executed the loop iterates.

This causes n to be incremented by 1 and again tested to see if it is still less than equal to 30. If it is, the statement sum=sum+n is executed. This process repeats until greater than 30, at which the loop terminates.

The symbolic program to add 30 integer numbers is as follows.

```
        ORG 100          ; /Origin of program is HEX 100
        LDA ADS          ; /Load first address (200H) of operands
        STA PTR          ; /Use address to initialize pointer
        LDA NUM          ; /Load minus 30
        STA CTR          ; /Use this value to initialize counter
        CLA              ; /Clear the contents of accumulator
LOP,    ADD PTR I        ; /Add an operand to AC and store result in AC
        ISZ PTR          ; /Increment pointer by 1
        ISZ CTR          ; /Increment counter by 1
        BUN LOP          ; /Repeat loop again
        STA SUM          ; /Store sum
        HLT              ; /Halt
ADS,    HEX 200          ; /First address of operands
PTR,    HEX 0            ; /This location reserved for a pointer
NUM,    DEC – 30         ; /Constant number to initialized counter
CTR,    HEX 0            ; /This location reserved for a counter
SUM,    HEX 0            ; /Sum is stored here
        ORG 200          ; /Origin of operands is HEX 200
        DEC 12           ; /First number (operand)
          :
        DEC 25           ; /Last number (operand)
        END              ; /End of symbolic program
```

Table 4.5.1 Symbolic program to add 1 to 30 integers

A compiler is a system program that translates a high-level language program to a machine language program. It requires knowledge of system programming to fully understand its operation. It is more complicated program than an assembler. The compiler may use an assembly language as an intermediate step in the translation or may translate the program directly to binary.

In the symbolic program given above, the compiler reserves locations $(200)_{16}$ to $(21D)_{16}$ for the 30 operands.

Pointer, Counter and Index Registers

The program given above uses a pointer and a counter together with the indirect address operation to form a program loop. The pointer points to the address of the current operand and the counter counts the number of times that the program loop is executed. This program uses two memory locations for these functions. If more than one processor registers are available in a computer, one of the processor register can be used

as a pointer, another as a counter and a third as an accumulator. When ... registers are used as pointers and counters, they are called **index registers.**

Example 4.5.1 *Write a program loop using a pointer and a counter to clear the ... hex locations 500 to 5FF with 0.* **GTU : Dec.-2...**

Solution :

```
        ORG 100         / Main program
        LDA ADS         / Load first address of operand.
        STA PTR         / Use address to initialize pointer
        LDA NUM         / Load minus 256
        STA CTR         / Use this value to initialize counter
        CLA             / Clear accumulator
LOP,    STA PTR I       / Store 0 at location pointed by PTR
        ISZ PTR         / Increment pointer by 1
        ISZ CTR         / Increment counter by 1
        BUN LOP         / Repeat loop again.
        HLT             / Halt
ADS,    HEX 500         / First address of operand
PTR,    HEX 0           / This location reserved for counter
NUM,    DEC-256         / Constant number to initialized counter
CTR,    HEX 0           / This location reserved for a counter
        END             / End of symbolic program
```

Review Questions

1. *What is the necessity of a loop in a program ? Explain with the help of example.*
2. *Write a program loop, using a pointer and a counter, that clears to 0 the contents of ... locations 400 through 4FF.*

4.6 Programming Arithmetic and Logic Operations

GTU : Dec.-10,11, Summer...

The number of instructions available in a computer varies from a few dozen ... hundred. In some computers, a given operation can be performed with one ... instruction; others may require a large number of machine instructions to perform ... same operation. For example, the four basic arithmetic operations are add, ... multiply and divide. Some computers have machine instructions to perform ... operations; some may not have. For example basic computer has only one arith... instruction, ADD. So for performing other operations, programs are to be written ... machine instructions available in the set of instructions for that computer.

When a single machine instruction performs an operation in a computer, that operation is said to be implemented by hardware. When a group of sequence of instructions performs an operation, that operation is said to be implemented by software. Hardware instructions speed up tasks. However, hardware implementation is more costly because of the additional circuits needed to implement the operation. Software implementation is cheaper than hardware implementation, but it results in increased program length and in execution time. Now we discuss the software implementation of some arithmetic and logic operations.

4.6.1 Multiplication

Consider two positive numbers having maximum eight significant bits. Their product cannot exceed 16-bits word. Fig. 4.6.1 shows the multiplication process. Let X be memory location to store multiplicand and Y to store multiplier. The multiplication process consists of checking the bits of the multiplier Y and adding the multiplicand X as many times as there are 1's in Y, provided that the value of X is shifted left from one line to the next. The computer can add only two numbers at a time. So a memory location is needed to store intermediate results. Let it be P. The intermediate sums are called **partial products.**

X : Memory location to store multiplicand
Y : Memory location to store multiplier
P : Memory location to store product

X : Memory location to store multiplicand
Y : Memory location to store multiplier
P : Memory location to store product

X =	0	0	0	0		1	1	1	1						
Y =	0	0	0	0		1	0	1	1				P		
	0	**0**	**0**	**0**		**1**	**1**	**1**	**1**		**0**	**0**	**0**	**0**	
	0	0	0	1		1	1	1	0		0	0	0	0	
	0	0	0	0		0	0	0	0		0	0	0	1	
	0	1	1	1		1	0	0	0		0	0	1	0	
	1	**0**	**1**	**0**		**0**	**1**	**0**	**1**		1	1	0	1	
											1	0	1	0	
											0	1	0	1	

Fig. 4.6.1 Multiplication of two 4-bit numbers

The step-by-step procedure for programming the multiplication operation is shown in the flow-chart given in Fig. 4.6.2. Refer Fig. 4.6.2 on page 4-25.

The steps involved in the multiplication operation are given below :

1. The memory location X holds the multiplicand and Y holds the multiplier. A counter CTR is set to –8 and location P holds the partial product which is cleared to zero.

2. The multiplier bit can be checked if it is transferred to the E register. This is done by clearing E, loading the value of Y into the accumulator, circulating right E and AC and storing the shifted number back into location Y.

Note This bit stored in E is the low-order bit of the multiplier.

3. Check the value of E.
4. If it is 1, the multiplicand X is added to the partial product P.
5. If it is 0, the partial product does not change.
6. Shift the value of X once to the left by loading it into the accumulator.
7. Circulate to left E and accumulator.
8. Increment CTR by 1.
9. When the CTR becomes 0, the program exists from the loop with product stored in location P.

The program to multiply two unsigned numbers is given in Table 4.6.1.

```
        ORG 100
LOP,    CLE          ; Clear E
        LDA Y        ; Load multiplier
        CIR          ; Transfer multiplier bit to E
        STA Y        ; Store shifted multiplier
        SZE          ; Check if bit is zero
        BUN ONE      ; Bit is one, go to ONE
        BUN ZRO      ; Bit is zero, go to ZRO
ONE,    LDA X        ; Load multiplicand
        ADD P        ; Add to partial product
        STA P        ; Store partial product
        CLE          ; Clear E
ZRO,    LDA X        ; Load multiplicand
        CIL          ; Shift left
        STA X        ; Store shifted multiplicand
        ISZ CTR      ; Increment counter
        BUN LOP      ; Counter not zero, repeat loop
        HLT          ; Counter is zero ; halt
CTR,    DEC-8        ; This location serves as a counter
X,      HEX 000F     ; Multiplicand stored here
Y,      HEX 000B     ; Multiplier stored here
P,      HEX 0        ; Product formed here
        END
```

Table 4.6.1 Program to multiply two positive numbers

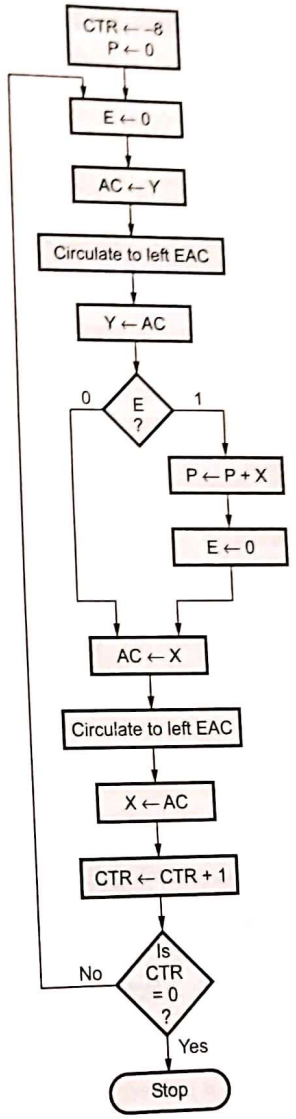


Fig. 4.6.2 Flowchart for multiplication operation

Note The initialization is not listed; but should be included when the program is loaded into the computer.

It is cleared from this example that if a computer does not have a machine instruction for a required operation, the same operation can be performed by writing a sequence of machine instructions or a program.

Example 4.6.1 *Write the program to multiply two positive numbers. By a repeated addition method. For example, to multiply 5 × 4 , the program evaluates the product by adding 5 four times, or 5 + 5 + 5 + 5.* **GTU : Dec.-10, May-11**

Solution :

Program :

```
        ORG 100         ; Origin of program is HEX 100
LOP,    LDA X           ; Load multiplicand
        ADD P           ; Add to partial product
        STA P           ; Store partial product
        ISZ CTR         ; Increment counter
        BUN LOP         ; If counter is not zero, repeat
        HLT             ; If counter is zero; stop
CTR,    DEC – 4         ; This location serves as a counter (multiplier)
X,      HEX  5          ; Multiplicand is stored here
P,      HEX  0          ; Product is formed here
        END
```

4.6.2 Double-precision Addition

When two 16-bit numbers are multiplied, the result is a 32-bit number. So two memory locations are needed to store it. A number stored in two memory locations is said to have double precision. In the multiplication operation, initially a partial product is computed. It is necessary that a double-precision number be added to the shifted multiplicand, which is also a double-precision number.

The steps involved in the program to add two double-precision numbers are given below :

1. Place one of the double-precision numbers in two consecutive memory locations AL and AH with 16 low-order bits in AL.
2. Place other number in BL and BH.
3. Add two low-order contents and transfer carry into E.
4. Clear accumulator.
5. Circulate the bit in E into the least significant position of the accumulator.

6. Add two high-order contents to the carry.
7. Store the double-precision sum in CL and CH.

The program is shown in Table 4.6.2.

```
        LDA AL          ; Load A with lower-bits
        ADD BL          ; Add lower-bits of B, carry in E
        STA CL          ; Store result of lower-bits addition in C low
        CLA             ; Clear AC
        CIL             ; Circulate to bring carry into AC (16)
        ADD AH          ; Add A higher-bits and carry
        ADD BH          ; Add higher-bits of B
        STA CH          ; Store result of higher-bits addition in C high
        HLT
AL,     –               ; Location of operands
AH,     –
BL,     –
BH,     –
CL,     –
CH,     –
```

Table 4.6.2 Program to add two double-precision numbers

4.6.3 Logic Operations

The basic computer has three machine instructions to perform logic operations. They are AND, CMA and CLA. There are a number of logic operations. Any logic function can be implemented using the AND and complement operations. So any logic operation can be implemented by using the available logic instructions in the basic computer. For example, the basic computer does not have the machine instruction to perform an OR operation. However, using DeMorgan's theorem, we can write the equation,

$$x + y = (x'y')'$$

The RHS of this equation includes only AND and complement operations. The basic computer has machine instructions to perform these two operations. So, the program can be written to perform OR operation of two logic operands A and B as shown in Table 4.6.3.

```
LDA A           ; Load first operand A
CMA             ; Complement to get Ā
STA TMP         ; Store in a temporary location
LDA B           ; Load second operand B
CMA             ; Complement to get B
AND TMP         ; AND with Ā to get Ā ∧ B
CMA             ;Complement again to get A ∨ B
```

Table 4.6.3 Program to OR two operands

The other logic operations can also be implemented by software in a similar

4.6.4 Shift Operations

There are three types of shifts

1/ Circular,

2/ Logical,

3/ Arithmetic.

The basic computer has machine instructions for the circular-shift operation other shifts, logical and arithmetic can be performed by writing program s number of instructions.

The logical shift can be performed by adding zeros to the extreme position be accomplished by clearing E and circulating the contents of accumulator a program to perform logical shift-right operation is given in Table 4.6.4.

```
CLE     . Clear E
CIR     . Circulate right AC and E
```

Table 4.6.4 Program to perform logical shift-right operation

The program to perform logical shift-left operation is shown in Table 4.6.5.

```
CLE     . Clear E
CIL     . Circulate left AC and E
```

Table 4.6.5 Program to perform logical shift-left operation

The arithmetic shifts depend on the type of representation of negative numbe arithmetic right-shift operation needs the sign bit remain unchanged in the le position. But the sign bit itself is shifted into the high-order bit position of the nu

The program to perform the arithmetic right-shift is given in Table 4.6.6.

```
CLE     Clear E to 0
SPA     Skip if AC is positive . E remains 0
CME     AC is negative  set E to 1
CIR     Circulate E and AC
```

Table 4.6.6 Program

operation is called **PUSH**. When ... ion is ... a **POP**.

The computer stores the information, much like stacking plates. Using this analogy stacking plates it is easy to illustrate the stack operation.

Fig. 4.7.1 shows the stacked plates. Here, we realize that if it is desired to take out the first stacked plate we will have to remove all plates above the first plate in the reverse order. This means that to remove first plate we will have to remove the third plate, then the second plate and finally the first plate. This means that, the first information pushed on the stack is the last information popped from the stack. This type of operation is known as a First In, Last Out (FILO). This stack is implemented with the help of special memory pointer register. The sp pointer register is called the **stack pointer.** During PUSH and POP operation pointer register gives the address of memory where the information is to be stored be read. The stack pointer's contents are automatically manipulated to point to The memory location currently pointed by stack pointer is called as **top of stack**.



Fig. 4.7.1 Stacked pla

Almost all computers provide special instructions to branch to a subroutine return to the main program from a subroutine.

In the basic computer, the branch and save return address (BSA) instruction is to form the link between the main program and a subroutine. Let us study the us this instruction in the program. The program shown below includes a subroutine subroutine is written to shift the content of the accumulator four times to the left processing BCD (Binary Coded Decimal) numbers or alphanumeric characters shifting a word four times is a useful operation. Since the machine instruction is available to perform this operation, a subroutine is written to accomplish this tasks.

```
Memory Location
                ORG 200         ; Main program
200             LDA A           ; Load A
201             BSA SHF         ; Branch to subroutine
202             STA A           ; Store shifted number
203             LDA B           ; Load B
204             BSA SHF         ; Branch to subroutine again
205             STA B           ; Store shifted number
206             HLT
207     A,      HEX 2243
208     B,      HEX 5532

                                ; Subroutine to shift left 4 times
209     SHF,    HEX 0           ; Store return address here
```

The subroutine performs the assigned task. The data on which the task is performed must be transferred from the main program to the subroutine. In previous example, an accumulator is used to store this data. The data is transferred through the accumulator from main program to the subroutine. The subroutine complements the number in the accumulator and left it there. Then the main program accepts the content of the accumulator from the subroutine. Thus, when a single input parameter and a single output parameter is to be transferred, an accumulator can be used. This is the case with basic computer and the computers with only one processor register. Computers with multiple processor registers can transfer multiple parameters with available registers.

The memory can also be used to transfer data between main program and subroutine. In this case, data are often placed in memory location following the call. A block of storage can also be used to place data. The first address of the block is then placed in the memory location following the call. The return address always gives the link information for transferring data between the main program and the subroutine.

Example 4.7.1 *Write a program which move the block of data.* **GTU : May-12, Marks 4**

Solution :

```
; Main Program
        BSA MOVE        ; Branch to subroutine MOVE
        HEX 200         ; Initialize address of source data
        HEX 300         ; Initialize address of destination data
        DEC -10         ; Initialize counter to indicate number of data
                        ; items to move (10 words)
        HLT             ; Stop
; Subroutine program
MOVE,   HEX 0           ; Store return address here
        LDA MOVE I      ; Load the address of source data
        STA PS          ; Initialize the source pointer
        ISZ MOVE        ; Increment the return address
        LDA MOVE I      ; Load the address of destination data
        STA PD          ; Initialize the destination pointer
        ISZ MOVE        ; Increment the return address
        LDA MOVE I      ; Load the counter
        STA CNT         ; Initialize the counter
        ISZ MOVE        ; Increment return address
BACK,   LDA PS I        ; Read the source data
        STA PD I
```



```
        ISZ PS          ; Increment source pointer
        ISZ PD          ; Increment destination pointer
        ISZ CNT         ; Increment counter
        BUN BACK        ; Repeat 10 times
        BUN MOVE I      ; Return to main program
PS,     —               ; Location of source pointer
PD,     —               ; Location of destination pointer
CNT,    —               ; Location of counter
```

Review Questions

1. *What is subroutine ? How stack is useful in execution of the subroutine ?*
2. *What is subroutine parameters and data linkage ?*
3. *What is subroutine ? How stack is useful in execution of the subroutine ?* **GTU : Dec.-03, Marks 4, Dec.-05, Marks 6**
4. *What is subroutine ? What is subroutine parameters and data linkage.* **GTU : June-05, Marks 4**
5. *Write a note on subroutines.* **GTU : Jan.-10, Marks 4**

4.8 Input-Output Programming

GTU : Dec.-09, May-12, Summer-13

The user has to enter program written in programming language into the computer. Each language has its symbols. These symbols are strings of characters. Each character is assigned an 8-bit code, which can be stored in computer memory. When an INP (input) instruction is executed, a binary-coded character enters the computer. When an OUT (output) instruction is executed, the binary-coded character is transferred to the output device. The output device detects the binary code and types the corresponding character.

The following instructions are needed to input a character and store it in memory :

```
CIF,    SKI         ; Check input flag
        BUN CIF     ; Flag = 0 branch to check again
        INP         ; Flag = 1, input character
        OUT         ; Print character
        STA CHR     ; Store character
        HLT
CHR,    –           ; Store character here
```

The first instruction SKI checks the input flag to see if a character is available for transfer. If the input flag bit is 1, the next instruction is skipped. The INP instruction transfers the binary-coded character into accumulator. Then, the OUT instruction prints the character. If the SKI instruction finds the flag bit 0, the next instruction in sequence is executed. This instruction is a branch to return and check the flag bit again. The two

instructions in the loop will be executed many times before a character is trans into the accumulator, because the input device is always slower than the computer

The instructions needed to print a character initially stored in memory are below.

```
        LDA CHR     ; Load character into AC
COF,    SKO         ; Check output flag
        BUN COF     ; Flag = 0, branch to check again
        OUT         ; Flag = 1, output character
        HLT
CHR,    HEX 0041    ; Character is "A"
```

The character which is to be printed is first loaded into the accumulator. The output flag is checked. If the flag bit is 0, the computer remains in a two inst loop checking the flag bit. When the flag bit is 1, the character is transferred fro accumulator to the printer.

4.8.1 Character Manipulation

We have seen that the symbols used for instructions are represented by binary characters. These can be manipulated by computer instructions to achieve various processing tasks. For example, packing two characters in one word. This task is req because each character needs 8-bits and a memory word contains 16 bits.

Example 4.8.1 *Write a subroutine to read and pack two characters.*

Solution :

```
READ2,  —           ; Subroutine
CHAR1,  SKI         ; Check input Flag
        BUN CHAR1   ; If Flag = 0, branch to check again
        INP         ; If Flag = 1, input first character
        OUT         ; Print character
        CIL         ; Circulate left eight times
        CIL
        CIL
        CIL
        CIL
        CIL
        CIL
        CIL
        AND MASK    ; Mask lower 8-bits
        SKI         ; Check input Flag
```

interrupt. First the computer stores the return address in location 0 and perform instruction in location 1 which starts the execution of service routine for the output transfer. The service routine can be stored any where in memory. However branch to the start of the routine is stored in location 1.

The service routine performs the following tasks using instructions.

1. Save the contents of the processor registers.
2. Check which flag is set.
3. Service the device whose flag is set.
4. Restore contents of processor registers.
5. Turn the interrupt facility on.
6. Return to the running program.

It is to be noted that the contents of processor registers before the interrupt and the return to the running program must be the same. It is necessary to save the cont of the registers at the start of the routine and restore them at the end because the se routine may use these registers.

The priority assigned to each device decides the sequence by which the flags checked. Only one device can be serviced at a time according to the priority.

Once the interrupt is occurred, it disables the facility from further interrupts. So service routine must turn the interrupt on before the return to the running prog This enables further interrupts. Also, the interrupt facility should be turned on after inserting the return address into the program counter.

Many input and output devices can be connected to the interrupt facility interrupt facility is not only useful for input and output transfers, but can also be u for other purposes. For example, internal processing errors, temperature control special alarm conditions etc.

Example 4.8.3 *Write a program which uses the service of an interrupt.*

Solution : **GTU : May-12, Marks**

```
Location
   0    FIRST, ——   ; Location for storing return address
   1    BUN ISR     ; Branch to Interrupt Service Routine
   :
 200    CMA         ; Part of running program
 201    ION         ; Enable interrupt
 202    ADD X       ;
 203    STA Z       : Interrupt occurs here
```



```
 204    LDA Y       ; After execution of ISR program returns here
   :
 300    ISR,  STA  LAC      ; Save the contents of AC
              CIR           ; Move E to AC(1)
              STA LE        ; Save the contents of E
              LDA P         ; Part of ISR
              :
              :
              STA Q         ;
              LDA LE        ; Restore the contents of E
              CIL           ; Shift it to E
              LDA LAC       ; Restore the contents of AC
              ION           ; Enable interrupt
              BUN FIRST I   ; Return to running program
```

As shown in the program, the location 0 is reserved to store the return address. The branch instruction to branch at the Interrupt Service Routine (ISR) is stored at location 1. The ION instruction in the running program enables interrupt.

As indicated in the program, interrupt occurs while computer is executing the instruction in location 203. Thus the binary equivalent of hex address of the next instruction in the sequence (204) is stored in location 0 and branch is taken.

Interrupt Service Routine (ISR) saves the contents of AC and E before execution of the instructions to complete interrupt task. At the end of ISR, the previous contents of AC and E are restored and program control is transferred to the running program.

Review Questions

1. *Write an ALP to input a character and store it in memory.*
2. *Write an ALP to print a character initially stored in memory.*
3. *What is program interrupt ? What happens when it comes ? What are the tasks to be performed by service routine ?* **GTU : Dec.-09, Marks 7**
4. *Write an assembly language program to take a character as input and outputs it.* **GTU : Summer-13, Marks 7**

❑❑❑

5 Microprogrammed Control

Syllabus

Control memory. Address sequencing. Micro program example. Design of control unit.

Contents

5.1	*Introduction*		
5.2	*Control Memory (CM)*	**Dec.-11,**	*Marks*
5.3	*Address Sequencing*	**Dec.-11, May-12,**	*Marks*
5.4	*Microprogram Example*	**Dec.-13,**	*Marks*
5.5	*Design of Control Unit*	**Jan.-10, Dec.-10,12, May-11**	*Marks*
5.6	*Key Definitions Related to Microprogrammed Control Unit*	**May-12,14,**	*Marks*
5.7	*Applications of Microprogramming*		
5.8	*Hardwired Control*	**May-14,**	*Mark*
5.9	*Comparison between Hardwired and Microprogrammed Control*	**Dec.-09,13,**	*Marks*

5.1 Introduction

The processor unit of a computer can be separated into two parts as data proc... unit and control unit. The data processing unit is a collection of functional units c... of performing certain operations on data, whereas control unit issues control sig... the data processing part to perform operations on data. There are two approac... design the control unit. They are hardwired control and microprogrammed contr... this chapter we will study the details of microprogrammed control approach. chapter also discusses the hardwired control.

5.2 Control Memory (CM)

GTU Dec...

Every instruction in a processor is implemented by a sequence of one or more s... concurrent micro-operations. Each micro-operation is associated with a specific w... control lines which, when activated, causes that micro-operation to take place. I... hardwired control, the control unit uses fixed logic circuits to interpret instructions... generate control signals from them. Microprogramming is an elegant and syste... method for controlling the micro-operation sequences. Since the number of instruc... and control lines is often in the hundreds, the complexity of hardwired control u... very high. Thus, it is costly and difficult to design. Furthermore, the hardwired co... unit is relatively inflexible because it is difficult to change the design, if one wishe... correct design error or modify the instruction set.

Microprogramming is a method of control unit design in which the control s... selection and sequencing information is stored in a ROM or RAM called a co... **memory** CM. The control signals to be activated at any time are specified b... **microinstruction**, which is fetched from CM in much similar way an instructio... fetched from main memory. Each microinstruction also explicitly or implicitly spec... the next microinstruction to be used, thereby providing the necessary information sequencing. A sequence of one or more micro-operations designed to control spe... operation, such as addition, multiplication is called a **microprogram**. The microprogr... for all instructions are stored in the control memory.

A control variable is a binary digit or bit (0 or 1) controls the function that specifi... micro-operation. When it is binary 1 state, the corresponding micro-operation is exec... while in the opposite binary state, the state of the registers in the system rem... unchanged. In a bus-organized system, the control signals that specify micro-operations are groups of bits that select the paths in multiplexers, decoders ALUs.

The address where the microinstructions are stored in control memory is gener... by microprogram sequencer/microprogram controller in the microprogrammed co... unit. Thus, the control unit initiates a series of sequential steps of micro-operations

per the operation, certain micro-operations are to be initiated, while others remain idle at any given time. Grouping the control variables at any given time form a string of 1's and 0's, called a **control word**. The control words are stored in the control memory to perform various operations on the components of the system. The control unit whose binary control variables are stored in memory is called a **microprogrammed control unit**. Each word in control memory contains within it a microinstruction. Each microinstruction specifies one or more micro-operations for the system. A sequence of microinstructions constitutes a microprogram. If the operations that are to be performed by a control unit are fixed, a Read-Only Memory (ROM) can be used as a control memory. According to the application, the words are stored in ROM during the hardware production of the unit.

An advanced development technique is the use of dynamic microprogramming. This technique allows a microprogram to be loaded initially from an auxiliary memory such as a magnetic disk. The control unit uses a writeable control memory. This allows to change the microprogram stored in a control memory.

A computer that uses microprogrammed control unit contains two separate memories, a main memory and a control memory. The user stores program in the main memory. This program consists of machine instructions and data can be changed as per the requirement. However, the user cannot change a microprogram in the control memory. The microprogram consists of microinstructions. Each machine instruction initiates a series of microinstructions in control memory. These microinstructions generate the micro-operations to fetch the instruction from main memory, to evaluate the effective address, to execute the operation specified by the instruction and to return control to the fetch phase in order to repeat the cycle for the next instruction.

Fig. 5.2.1 shows the general configuration of a microprogrammed control unit. It consists of control memory, control address register, microinstruction register (control data register) and microprogram sequencer. (Refer Fig. 5.2.1 on next page.)

The components of control unit work together as follows :

- The control address register (μpc) holds the address of the next microinstruction t... be read. Every time a new instruction is loaded into the IR, the output of th... block labeled "starting address generator" is loaded into the μpc.
- When address is available in control address register, the sequencer issues REA... command to the control memory.
- After issue of READ command, the word from the addressed location is read int... the microinstruction register.
- The μpc is then automatically incremented by the clock, causing successiv... microinstructions to be read from the control memory.

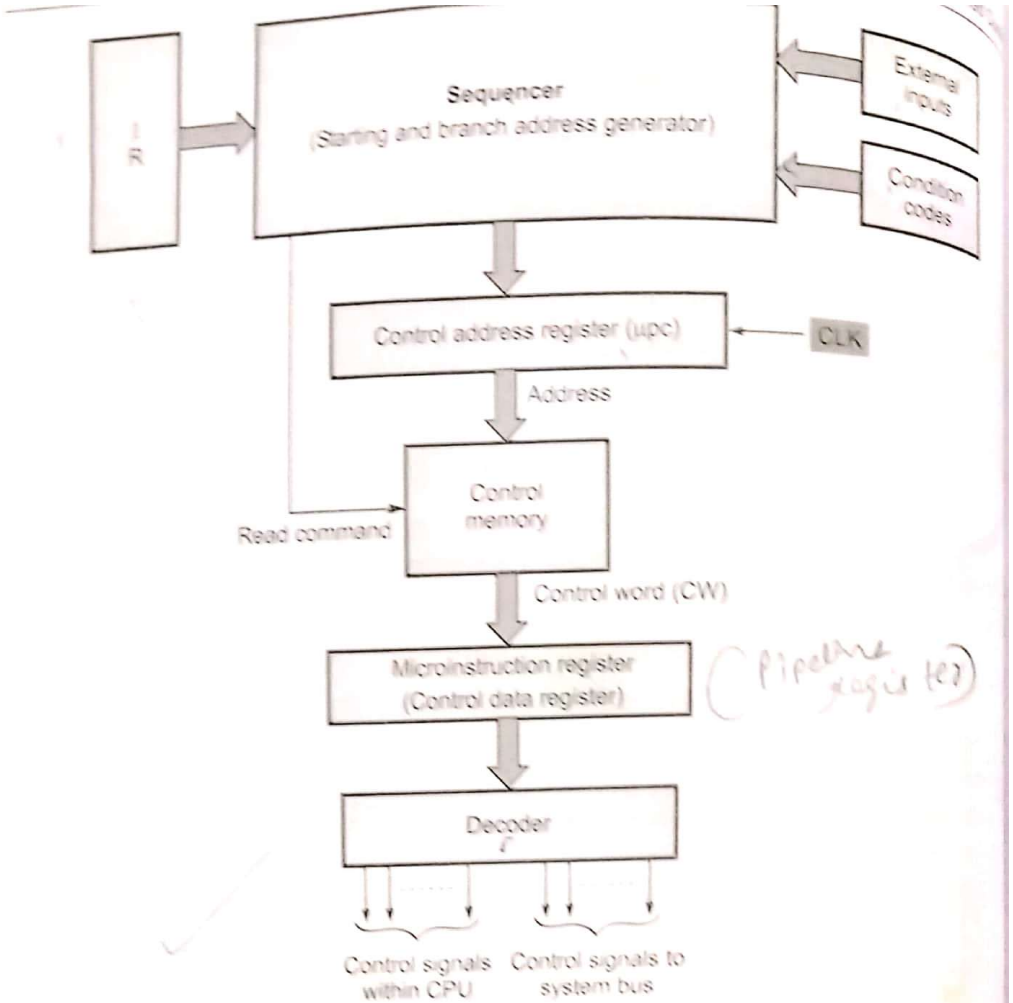


Fig. 5.2.1 Microprogrammed control unit

- The content of the microinstruction register (Control data register) gen control signals which are delivered to various parts of the processor in the sequence.

The control data register holds the present microinstruction while the next addre computed and read from memory. This data register is sometimes called a pipe **register**. It allows the execution of micro-operations specified by the control word the generation of the next microinstruction simultaneously. A two-phase clo required for this configuration, with one clock applied to the address register and other to the data register. However, the system can be operated with a single-p clock applied to the address register. The control word and next address information taken directly from the control memory. The control memory is a ROM that operate a combinational circuit. The address value is the input and the word stored at address is the output of this memory. This word remains in the output wires as lon

only a few bits are used in any given instruction. The solution of this problem group the control signals. Grouping technique is used to reduce the number of the microinstruction.

Microinstructions are stored in control memory in groups, with each group specifying a routine. Each computer instruction has its own microprogram routine. It is stored control memory to generate micro-operations that execute the instruction. The hardware controls the address sequencing of the control memory. It must be capable of sequencing the microinstructions within a routine. It must also be able to branch from one routine to another.

Now let us discuss the steps that the control must undergo during the execution of a single computer instruction.

When the computer is turned on, an initial address is loaded into the control address register. This address is usually the address of the first microinstruction in the fetch routine. The fetch routine may be sequenced by incrementing the control address register through its remaining microinstructions. The fetch routine fetches the instruction and stores it in the instruction register.

In the next step, the control memory goes through the routine that determines the effective address of the operand. A machine instruction may have bits that specify various addressing modes. We know that, machine instructions involve a number of addressing modes and there can be many instructions and addressing mode combinations. A separate microinstruction for each of these combinations would require considerable duplication of common microinstructions. As a result, large space in control memory is required to store these microinstructions. So, it is necessary to organize microprograms such that they share as many microinstructions as possible. This requires many branch microinstructions, both unconditional and conditional to transfer control amongst the various microinstructions. Thus, it is important to design control unit time-efficient techniques for microinstruction branching.

In the last step, the control unit generates micro-operations that execute the instruction fetched from memory. The opcode (operation code) part of the instruction decides the micro-operation steps to be generated in processor registers. Each instruction has its own microprogram routine stored in a given location of control memory. The total process, from the instruction code bits to the generation of address in control memory where the microprogram routine is located that is required for execution of instruction is called a **mapping process**. Once the address of the routine is obtained, microinstructions that execute the instruction may be sequenced by incrementing the control address register. However, sometimes the values of certain status bits in processor registers decide the sequence of micro-operations. If the microprogram includes subroutine, then it requires an external register for storing the return address. The return address cannot be stored in control memory because control memory is ROM and writing capability in the ROM is not available for the normal user.

After execution of the instruction, the control must be returned to the fetch routine. This is done by executing an unconditional branch microinstruction to the first address of the fetch routine.

The task of address sequencing discussed above can be summarized as given below

1. Increment control address register.
2. Perform unconditional or conditional branch according to the status bits in processor registers.
3. Obtain the control memory address of microprogram routine from the opcode of an instruction.
4. Microprograms may include subroutines, for which subroutine call and return facility is available.

From above discussion, we can draw block diagram of hardware required to select next address of microinstruction in a control memory as shown in Fig. 5.3.1.

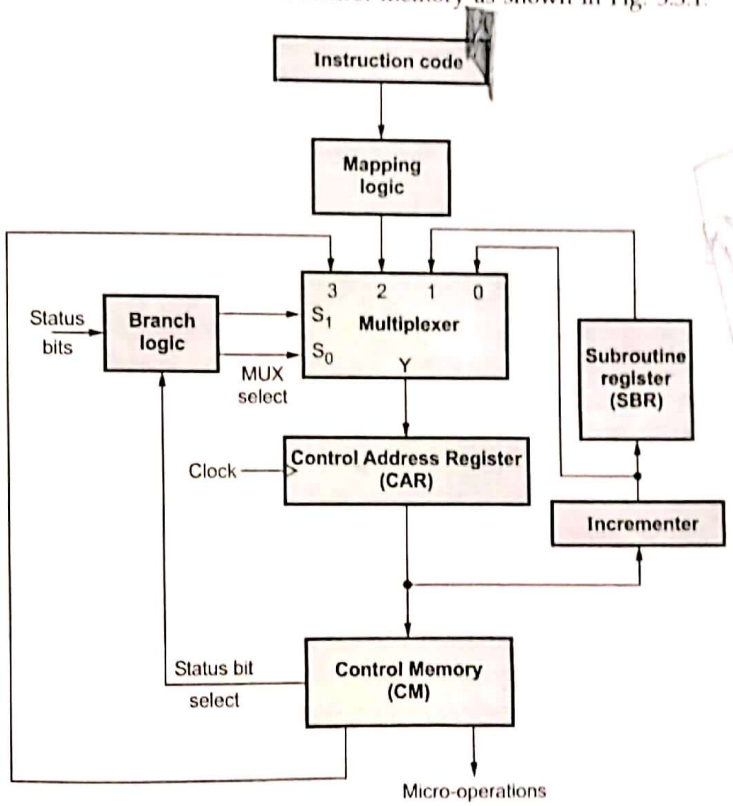


Fig. 5.3.1 Selection of address for control memory

5.3.2 Mapping of Instruction

We have seen that the transformation from the instruction code bits to an address in control memory where the microprogram routine is located is referred to as a **mapping process.** The status bits for the branch needed for execution of this routine are the bits in the operation code part of the instruction.

As an example, consider a computer having instruction format as shown in Fig. 5.3.2. In this format, operation code is of four bits. So, it can specify 16 (2^4) different instructions. Assume that the control memory has 128 words. So, it requires seven bits to store address (2^7 = 128).

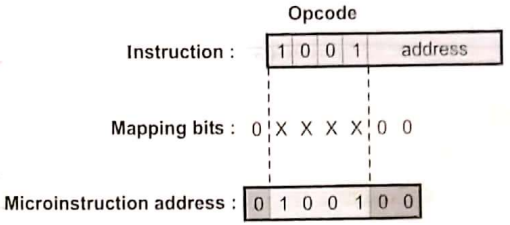


Fig. 5.3.2 Mapping of instruction

Each instruction of a computer has its operation code (opcode). For each opcode, there is a microprogram routine in control memory that is to be executed for the execution of an instruction. Fig. 5.3.2 shows the mapping process that converts the 4-bit opcode to a 7-bit address for accessing microprogram routine in a control memory.

This mapping process includes,

1. Placement of 0 in the most significant bit of the address.
2. Transfer of 4-bit operation code in the address.
3. Clearing the two least significant bits of the control address register.

This mapping provides a microprogram routine with a capacity of four microinstructions for each instruction. However, if the microprogram routine needs more than four microinstructions, it can use addresses 1000000 through 1111111 or unused memory locations of instructions which uses fewer than four microinstructions.

Example 5.3.1 *Formulate a mapping procedure that provides eight consecutive microinstructions for each routine. The operation code has six bits and the control memory has 2048 words.*

Solution : Each instruction of a computer has its operation code (opcode). For each opcode, there is a microprogram routine in control memory that is to be executed for the execution of an instruction.

It is given that the opcode has 6-bits. Since the control memory has 2048 (2^{11}) words, the address of control memory is 11-bits.

The Fig. 5.3.3 shows the mapping process that converts the 6-bit opcode to a 11-bit address for accessing microprogram routine in a control memory.

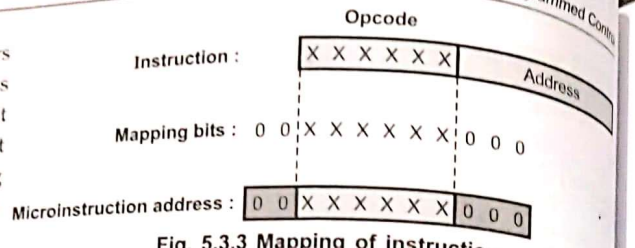


Fig. 5.3.3 Mapping of instruction

This mapping process includes :

1. Placement of two 0s in the most significant bit of the address.
2. Transfer of 6-bit opcode in the address.
3. Clearing the three least significant bits of the control address register.

This provides for each computer instruction a microprogram routine with a capacity of (2^3) eight microinstruction.

Example 5.3.2 *Formulate a mapping procedure that provides four consecutive microinstruction for each routine. The operation code has six bits and the control memory has 2048 words.*

Solution : This example is similar to previous example, except here mapping procedure provides four consecutive microinstructions instead of eight. The Fig. 5.3.4 shows the mapping of instruction.

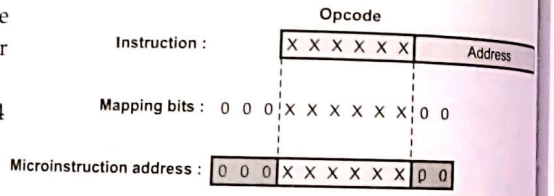


Fig. 5.3.4 Mapping of instruction

Example 5.3.3 *Show how a 9-bit micro-operation field in a microinstruction can be divided into subfields to specify 46 micro-operations. How many micro-operations can be specified in one microinstruction ?*

Solution : The Fig. 5.3.5 shows the micro-operation field of a microinstruction. It is divided into two subfields :

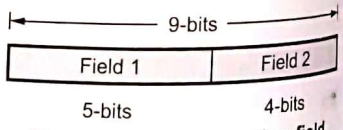


Fig. 5.3.5 Micro-operation field

Field 1 = 5-bits. These bits are encoded to specify $(2^5 - 1)$ 31 distinct micro-operations and one no operation.

Field 2 = 4-bits. These bits are encoded to specify $(2^4 - 1)$ 15 distinct micro-operations and one no operation.

In all field 1 and field 2 can specify (31 + 15) 46 micro-operations. Since there are two micro-operation fields we can specify at the most two micro-operations for a microinstruction, one from each field.

5.3.3 Subroutines

In general, subroutine can be explained as below. When a certain group of instructions is repeatedly required in a program, then instead of writing it repeatedly, it can be stored separately and can be called in a main program whenever required. This group of instructions is called **subroutine**. This is the way to use memory efficiently.

In microprogrammed control unit, many microprograms contain identical sections of code. In this case, subroutines can be used for common sections of microcode. For example, generation of the effective address of the operand for an instruction is the task commonly required for the execution of memory reference instructions. The subroutine can be written to perform this task and can be called within many routines to execute the effective address computation.

Subroutine Register

Subroutines can be used in microprograms to use control memory efficiently. When the microprogram (main routine) needs subroutine, the address of subroutine is to be loaded into the control address register. This transfer program control from main routine to the subroutine. However, after execution of subroutine, the control should be transferred back to the main routine. So, before transferring control from main routine to the subroutine, it is necessary to store return address. This may be accomplished by storing the incremented output from the control address register (return address) into a subroutine register and then branching to the beginning of the subroutine. Upon completion of subroutine execution, the return address is restored into the control address register from subroutine register. Thus, the subroutine register stores the return address during a subroutine call and restores it during a subroutine return. The stack (registers organized in LIFO fashion) can be used for the execution of subroutines.

Review Questions

1. *Explain the mapping process.*
2. *Explain the procedure for mapping of instruction.*
3. *Explain address sequencing process in a microprogrammed control unit using suitable diagram.* **GTU : Dec.-11, Marks 7**
4. *Explain address sequencing with block diagram.* **GTU : May-12, Marks 7**

5.4 Microprogram Example

5.4.1 Computer Configuration

When a computer uses microprogrammed control unit, the designer has to generate the microcode that is to be stored in control memory. The generation of microcode is called **microprogramming**. This process is similar to conventional machine language programming. In this section we see how a simple digital computer with microprogrammed control unit is microprogrammed. Let us first see the configuration of this computer.

Fig. 5.4.1 shows the block diagram of a computer with microprogrammed control unit.

The total hardware configuration of this computer can be divided mainly into two parts as processor unit and control unit.

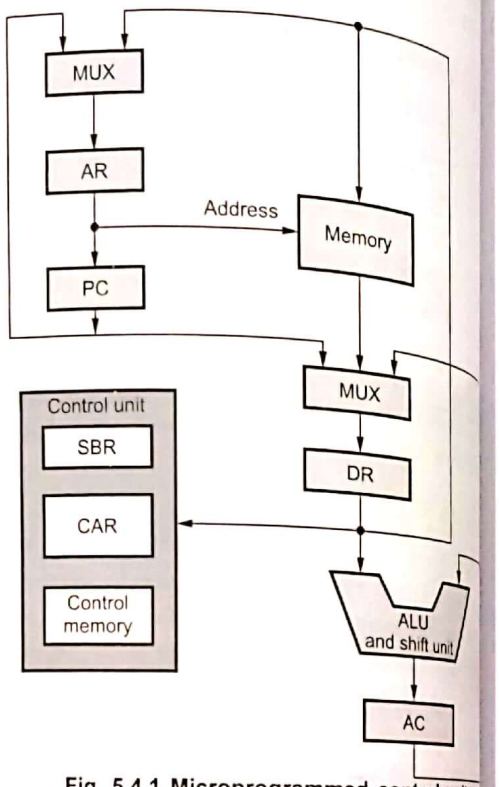


Fig. 5.4.1 Microprogrammed control unit

Processor unit

The processor unit consists of

- Main memory
- Arithmetic, logic and shift unit
- Registers

A main memory unit stores instructions and data. It receives address from A... Register (AR) to locate memory location and data from Data Register (DR). Also th... read from memory can go only to DR. The arithmetic, logic and shift unit per... micro-operations on operands received from Accumulator (AC) and Data Register... and stores result back into accumulator.

The processor registers are Program Counter (PC), Address Register (AR)... Register (DR) and Accumulator (AC). The functions of these registers are similar t... of the registers in basic computer.

Control unit

The control unit consists of

- Control memory
- Control Address Register (CAR)
- Subroutine Register (SBR)

We have already studied the organization of microprogrammed control unit and functions of its units in previous section.

It is to be noted that the information is transferred among the registers in the processor through multiplexers rather than a common bus.

Instruction format

Fig. 5.4.2 shows the format of a computer instruction.

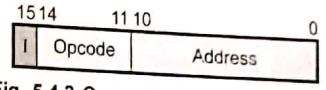


Fig. 5.4.2 Computer instruction format

As shown in Fig. 5.4.2, the instruction includes three fields.

1. **I :** A one-bit field for indirect addressing.
2. **Opcode :** A four-bit operation code.
3. **Address :** An eleven-bit address.

Example 5.4.1 *Explain why each of the following micro-operation cannot be executed during a single clock pulse.*

$IR \leftarrow M\ [PC]$

$AC \leftarrow AC + TR$

$DR \leftarrow DR + AC$

Also specify a sequence of micro-operations that will perform the above operation.

Solution : Let us consider the computer hardware configuration shown in Fig. 5.4.1. For above micro-operations,

i) IR ← M [PC]

Since AR (Address Register) specifies the address of memory we have to first transfer contents of PC to AR. Thus, above operation is split into two micro-operations :

$AR \leftarrow PC$

$IR \leftarrow M\ [AR]$

ii) AC ← AC + TR

According to hardware configuration two inputs for ALU is DR and AC. Thus we have to first transfer contents of TR to DR and then add them to AC to perform $AC \leftarrow AC + TR$. This gives us two micro-operations :

E-bit	Associated micro-operation	Symbol
000	—	NOP
001	AC ← AC − DR	SUB
002	AC ← AC ∨ DR	OR
003	AC ← AC ∧ DR	AND
100	DR ← M[AR]	READ
101	DR ← AC	ACTDR
110	DR ← DR − 1	INCDR
111	DR(0 − 10) ← PC	PCTDR

Table 5.4.1 Microinstructions with their binary code, micro-operation and symbol

It is to be noted that two or more conflicting micro-operations can be used with microinstruction.

From Table 5.4.1, we can observe,

1. Each micro-operation is defined with a register transfer statement.
2. A symbol is assigned to each micro-operation. This is useful in writing symbolic program.
3. All transfer-type micro-operations symbols use five letters. The first two letters indicate the source register, the third letter is always T and the last two letters indicate the destination register.

Condition Field

The two-bits in the condition (CD) field are encoded to specify four status conditions as listed in Table 5.4.2.

CD 2-bit	Symbol	Status bit condition	Description
00	U	(1) always	Represents unconditional branch
01	I	15th bit of DR : DR(15)	Indirect address bit.
10	S	15th bit of AC : AC(15)	Represents sign bit of AC
11	Z	AC = 0	Indicates zero value in AC

5.4.3 Symbolic Microinstructions and Microprogram

A symbolic microprogram can be written using symbols of microinstructions. It can be translated into its binary equivalent using an assembler. A microprogram assembler is similar in concept to a conventional computer assembler.

The assembly language program is written using symbolic microinstructions. Each symbolic microinstruction is represented using five fields.They are label, micro-operations, CD, BR and AD. The details about these fields are given below.

1. **Label :** This field may be empty or it may specify a symbolic address. It is terminated with a colon (:).
2. **Micro-operations :** This field consists of one, two or three symbols separated by commas. Only one symbol from each F field can be taken in each microinstruction. When the microinstruction has no operation, the NOP symbol is used.
3. **CD :** This field contains one of the letters U, I, S or Z (Refer Table 5.4.3).
4. **BR :** This field contains one of the four symbols defined in Table 5.4.3.
5. **AD :** This field specifies 7-bit address.

   This address is obtained using one of three possible ways.

   a. A symbolic address specified by a label.

   b. A symbol NEXT which specifies the next sequential address.

   c. Zero address (7 zeros) when the BR field contains RET or MAP symbol.

ORG

The pseudo-instruction ORG is used to define the first address (origin) of microprogram.

For example : ORG 100

This instruction informs the assembler to place the next microinstruction in control memory at decimal address 100. That is, at the binary address 1100100.

The Fetch Routine

We have seen that the control memory has 128 words and each word is of 20 bits. Microprogramming needs to determine each of this word and stores it in control memory. Out of 128 words, the first 64 words (addresses 0 to 63) are used to store routines for the 16 instructions. The last 64 words may be used for any other purpose. Address 64 is the convenient starting location for the fetch routine.

The fetch routine includes following micro-operations :

1. AR ← PC

The address of the instruction is transferred from PC to AR.

2. DR ← M[AR]

   The instruction at an address given by AR is read into DR. Now DR contains instruction code.

3. PC ← PC + 1

   PC contents are incremented by 1.

4. AR ← DR (0 - 10)

   The address part of the instruction code in DR is transferred to AR.

5. CAR (2 - 5) ← DR (11 - 14),

   CAR (0, 1, 6) ← 0

   Using mapping process, the instruction code (operation code) bits from DR are transformed into an address. Then this address is loaded in CAR and control is transferred to one of 16 routines.

Fetch and Decode

The symbolic microprogram for the fetch routine is given below. It includes three microinstructions which are placed in control memory at addresses 64, 65 and 66.

```
        ORG 64
FETCH:  PCTAR          U     JMP  NEXT
        READ, INCPC    U     JMP  NEXT
        DRTAR          U     MAP
```

The symbolic microprogram can be translated into binary microprogram using Tables 5.4.1, 5.4.2 and 5.4.3. It is given below.

Microinstruction	Address of microinstruction	F1	F2	F3	CD	BR	AD
PCTAR	1000000 (64)	110	000	000	00	00	1000001
Read, INC PC	1000001 (65)	000	100	101	00	00	1000010
DRTAR	1000010 (66)	101	000	000	00	11	0000000

Table 5.4.4

The symbolic representation is useful for writing microprograms in an assembly language format. The binary representation is the actual internal content that must be stored in control memory.

Review Question

1. Write a note on microinstruction format. **GTU : Dec.-13, Marks 4**

5.5 Design of Control Unit

GTU : Jan.-10, Dec.-10,12, May-13

A simple way to structure microinstructions is to assign one bit position to each control signal required in the CPU. However, this scheme has one serious drawback-assigning individual bits to each control signal results in long microinstructions, because the number of required signals is usually large. Moreover only a few bits are used in any given instruction. The solution of this problem is to group the control signals.

Thus, the number of control bits that initiate micro-operations can be reduced by grouping mutually exclusive variables into fields and encoding the k bits in each field to provide 2^k micro-operations. Each field requires a decoder to produce the corresponding control signals. This method reduces the size of the microinstruction. However it increases the hardware external to the control memory. It also increases the delay time of the control signals since they must propagate through the decoding circuits.

The micro-operation field is of 9-bits which is divided into three subfields of three bits each.

5.5.1 Decoding of Micro-operation Fields and their Further Processing

Fig. 5.5.1 shows the block diagram to decode micro-operation fields and their further processing. (see Fig. 5.5.1 on next page.)

The functions of blocks in Fig. 5.5.1 are explained below.

3 × 8 decoders

The control memory output of each subfield in microinstruction are decoded with 3 × 8 decoder to provide eight outputs.

These outputs are connected to the proper circuit to initiate the corresponding micro-operation.

Arithmetic logic shift unit

The arithmetic logic shift unit is designed to perform arithmetic, logical and shift operations. The output of the decoders are used to select the appropriate operation in the arithmetic logic shift unit. The Fig. 5.5.1 shows some of the outputs of decoder connected to the arithmetic logic shift unit. The other outputs of the decoders that are associated with an AC operation are also connected (not shown in the Fig. 5.5.1) to the arithmetic logic shift unit.

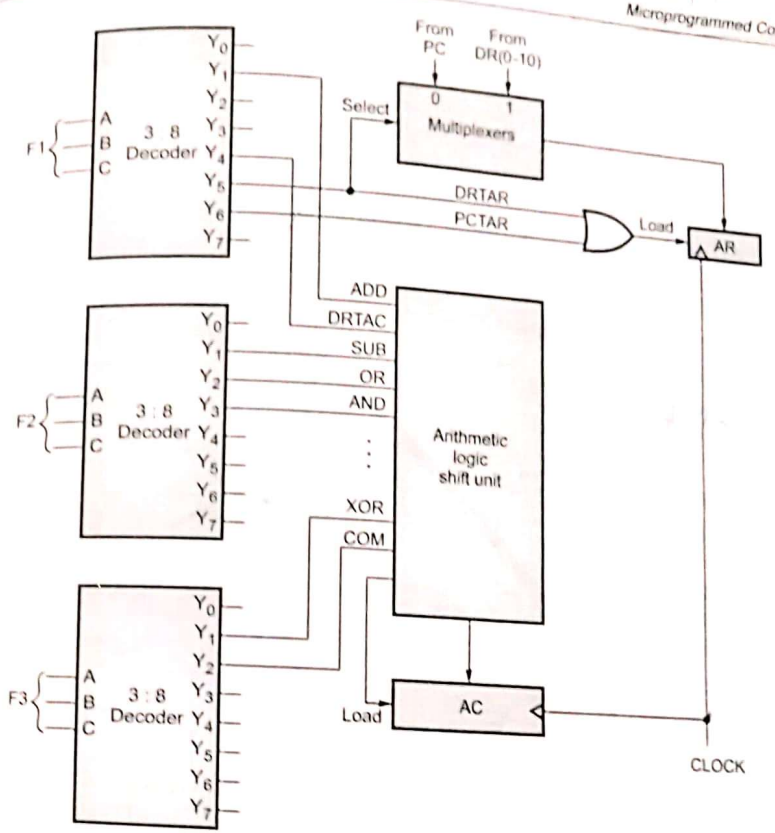


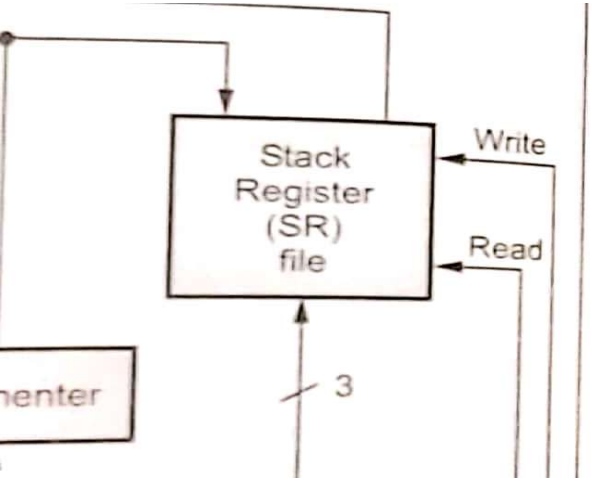
Fig. 5.5.1 Decoding of micro-operation fields and their further processing

Multiplexer

Multiplexer is used to select the source for the AR register. When select input is 0, multiplexer selects the contents of PC as a source, otherwise it uses the contents of DR as a source.

5.5.2 Microprogram Sequencer

The subunit of the microprogrammed control unit which presents an address to the control memory is called **microprogram sequencer.** The next-address logic of the sequencer determines the specific address source to be loaded into the control address register.



I_2	I_1	I_0	T	S_1	S_0	Operation	Description
X	0	0	X	0	0	CAR ← EXA	Transfer external address.
1	0	1	1	0	1	CAR ← BRA, SR ← CAR + 1	Branch to subroutine and save the next instruction address in stack (Push operatio
0	0	1	1	0	1	CAR ← BRA	Transfer branch address.
X	1	0	X	1	0	CAR ← SR	Transfer from stack register.
0	1	1	0	1	1	CAR ← CAR + 1	Increment address.

5.6 Key Definitions Related to Microprogrammed Control Unit

GTU : May-12,14

Control memory : A memory that is part of a control unit is referred to as a control memory. It stores the sequences of micro-operations to be performed to execute microinstructions.

Microprogram sequencing : Determining the address of the next microinstruction to be executed using microprogram sequencer is called **microprogram sequencing.**

Address sequencing : Each computer instruction has its own microprogram routine in control memory to generate the micro-operations that execute the instruction. To execute a particular computer instruction, accessing a corresponding microinstruction routine, sequencing the microinstructions within the routine and if necessary branching from one routine to another is known as **address sequencing**.

Register transfer language : The symbolic notation used to describe the micro-operation transfers among registers is called a **register transfer language**.

Micro-operation : To perform fetch, decode and execute cycles, the processor unit has to perform set of operations called **micro-operations**.

Microinstruction : Each word in the control memory is a microinstruction which specifies the control signals to be activated to perform one or more **micro-operations.**

Microcode : The translation of symbolic microprogram to binary produces a binary microprogram called **microcode**.

Microprogram : A sequence of one or more micro-operations designed to perform specific operation, such as addition, multiplication is called a **microprogram**.

Pipeline register : The control data register holds the present microinstruction while the next address is computed and read from memory. This data register is sometimes called a **pipeline register**. It allows the execution of micro-operations specified by the control word and the generation of the next microinstruction simultaneously.

Review Questions

1. *Describe the following terms with proper example in each case : Microoperation, microinstruction, microprogram microcode.* GTU : May-12, Marks 7
2. *Define terms : Register transfer language and micro-operation.* GTU : May-14, Marks 2
3. *Draw and explain the microprogrammed control unit.*

5.7 Applications of Microprogramming

The various microprogramming applications are :

Realization of computers : Microprograms are used to implement the control unit of the computer.

Emulation : Microprograms are used in emulation process. Emulation refers to the use of a microprogram on one machine to execute programs originally written for another. The emulation process is used as an aid to users in migrating from one computer to another.

Operating system support : Microprograms can be used to implement some of the primitives of operating system. This simplifies the task of operating system implementation. The implementation of primitives using microprogram also improves the performance of operating system.

High-level language support : In high-level language various sub functions and data types can be implemented using microprogramming. This makes it easy to compile the program into an efficient machine language form.

Micro-diagnostics : Microprogramming can be used to detection, isolation, monitoring and repair of system errors. These features are known as micro-diagnostics and they significantly enhance the system maintenance facility.

User tailoring : By implementing control memory using RAM it is possible to tailor the machine to the desired application.

Review Question

1. *Give applications of microprogramming.*

5.8 Hardwired Control

GTU : May-14

In the hardwired control, the control units use fixed logic circuits to interpret instructions and generate control signals from them. The fixed logic circuits use contents of the control step counter, contents of the instruction register, contents of the condition code flag and the external input signals such as MFC and interrupt requests

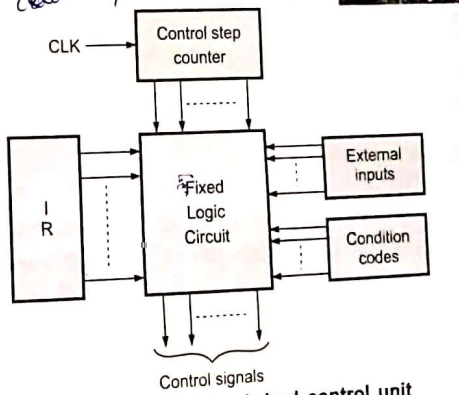


Fig. 5.8.1 Typical hardwired control unit

to generate control signals. Fig. 5.8.1 shows the typical hardwired control unit. It fixed logic circuit block includes combinational circuit (decoder and encoder generates the required control outputs, depending on the state of all its inputs

5.8.1 Advantages of Hardwired Control

- Hardwired control unit is fast because control signals are generate combinational circuits.
- The delay in generation of control signals depends upon the number of gate
- It has greater chip area efficiency since its uses less area on-chip.

5.8.2 Disadvantages of Hardwired Control

- More the control signals required by CPU; more complex will be the de control unit.
- Modifications in control signal are very difficult. That means it re rearranging of wires in the hardware circuit.
- It is difficult to correct mistake in original design or adding new feat existing design of control unit.

Review Questions

1. *Define hardwired control.* GTU : May-14, Ma
2. *Draw and explain typical hardwired control unit.*
3. *State the advantages and disadvantages of hardwired control.*

5.9 Comparison between Hardwired and Microprogrammed Contr

GTU : Dec.-09

Attribute	Hardwired control	Microprogrammed contr
Speed	Fast	Slow
Control functions	Implemented in hardware	Implemented in software
Flexibility	Not flexible, to accommodate new system specifications or new instructions.	More flexible, to accommodate system specification or new instructions redesign is requir
Ability to handle large/complex instruction sets	Somewhat difficult	Easier

Ability to support operating systems and diagnostic features	Very difficult (unless anticipated during design)	Easy
Design process	Somewhat complicated	Orderly and systematic
Applications	Mostly RISC microprocessors	Mainframes, some microprocessors
Instruction set size	Usually under 100 instructions	Usually over 100 instructions
ROM size		2 K to 10 K by 20-400 bit microinstructions
Chip area efficiency	Uses least area	Uses more area

Review Question

1. *State the differences between hardwired and microprogrammed control unit.* GTU : Dec.-09,13, Marks 4

❑❑❑

6 Central Processing Unit

Syllabus

Introduction, General register organization, Stack organization, Instruction format, Addressing modes, Data transfer and manipulation, Program control, Reduced Instruction Set Computer (RISC)

Contents

6.1	Introduction	
6.2	General Register Organization	
6.3	Stack Organization	Dec.-09, 12,13, May-11,12,13, June-08, Jan.-10, Feb.-10 Marks 7
6.4	Instruction Format	Dec.-02,11,12,10, Dec.-02,09, 11,12,13, June-06, 08, 10,Jan.-10, Feb.-10, May-12,13, Marks 16
6.5	Addressing Modes	Dec.-09, Marks 8
6.6	Data Transfer and Manipulation	Dec.-02, 09, 11, June-06, Feb.-10, Winter-13, May-13, Marks 7
6.7	Program Control	Dec.-02,05, 06, 09, 11, 12,13, May-11,12, 13, June-06, 08, Feb.-10, Winter-13 Marks 8
6.8	Reduced Instruction Set Computer (RISC)	

6.1 Introduction

In addition to execute programs, the processor controls the functioning of system components with the help of control signals. It directly or indirectly controls operations such as data transfers between I/O devices and main memory. It supports interrupt facility by which external devices can request for processor service. The major functions of the processor are summarized in the flowchart shown in Fig. 6.1.1 Flowchart showing major functions of processor.

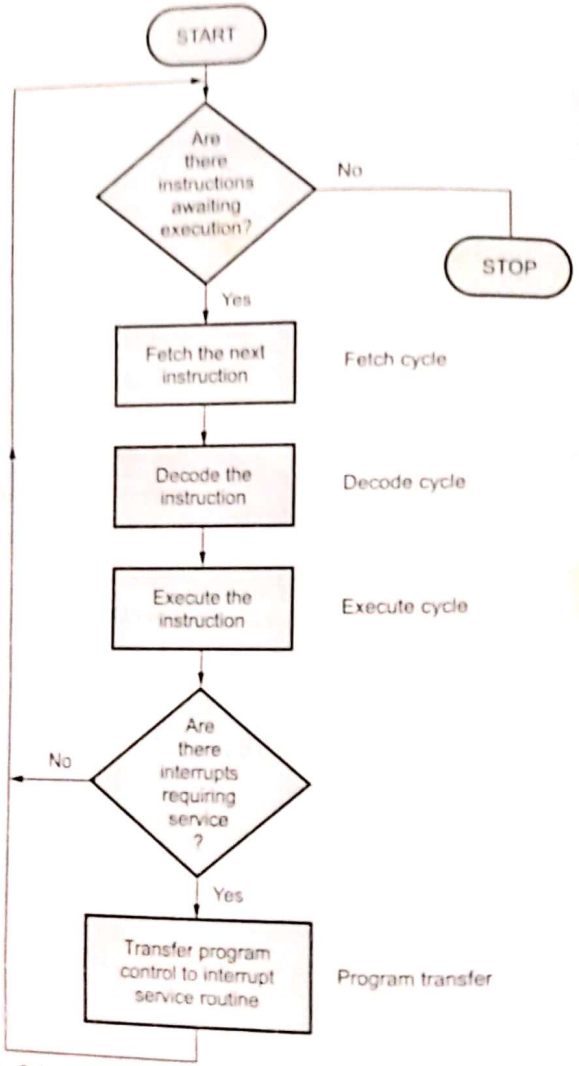


Fig. 6.1.1 Flowchart showing major functions of processor

To perform above mentioned functions processor organisation is divided into many functional unit and each functional unit is responsible for execution of particular tasks. Let us study the general processor organization.

The Fig. 6.1.2 shows the general processor organization. It includes three major logic devices.

- ALU
- Several registers
- Control unit

The internal data bus is used to transmit data between these logic devices.

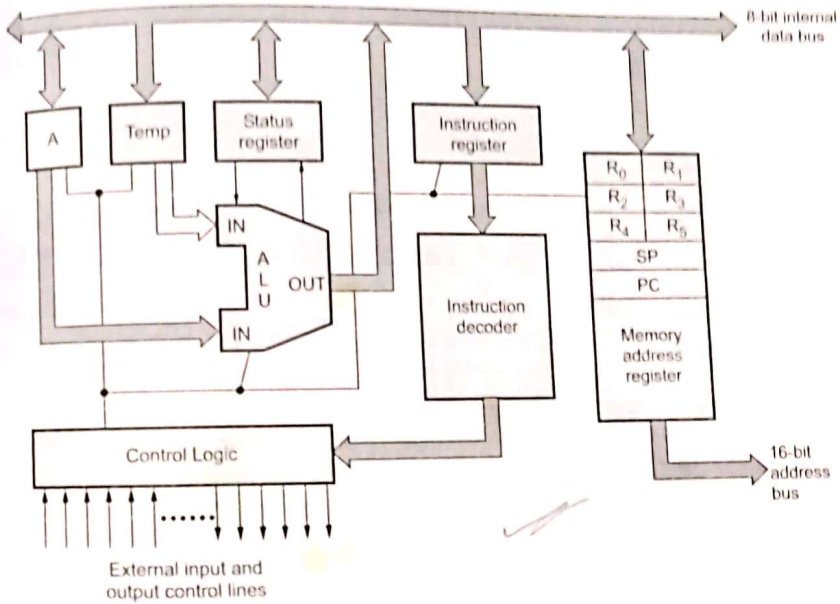


Fig. 6.1.2 General processor organization

ALU :

One of the processor's major logic devices is the Arithmetic Logic Unit (ALU). It contains the processor's data processing logic. It has two inputs and an output. The internal data bus of processor is connected to the two inputs of ALU through the temporary register and the accumulator.

The ALU's single output is connected to the internal data bus. It allows to send the output of ALU over the bus to any device connected to the bus. In most of the

processors register A gives data for the ALU and after performing the operation, resulting data word is sent to the register A and stored there. This special reg where the result is accumulated is commonly known as **accumulator**.

The ALU works on either one or two data words, depending on the kind operation. The ALU uses input ports as necessary. For example, addition operation both ALU inputs while complementing data operation uses only one input complement the data word, all the bits of the word's that are logic 1 are set to log and all the bits of the word at logic 0 are set to logic 1.

The ALU of the most of the processors can perform following functions.

- Add
- Subtract
- AND
- OR
- Exclusive OR
- Complement
- Shift right
- Shift left
- Increment
- Decrement

Registers :

Registers are a prominent part of the block diagram and the programming mod any processor. The basic registers found in most of the processors are the accum the program counter, the stack pointer, the status register, the general purpose reg the memory address register, the instruction register and the temporary data regist

Control Logic :

The control logic is a important block in the processor. The control log responsible for working of all other parts of the processor together. It maintains synchronization in operation of different parts in the processor. The synchronizati achieved with the help of one of the control logic's major external inputs, proces clock. The clock is a signal which is the basis of all the timings inside the processor

Usually processor's control logic is **microprogrammed.** This means that organization of the control logic itself is much like the organization of a very sp purpose processor.

The control logic receives the signal from instruction decoder which decodes instruction stored in the instruction register. The control logic then generates the co

signals necessary to carry out this instruction. The control logic does a few other special functions. It looks after the processor power-up sequence. It also processes interrupts. An interrupt is like a request to the processor from other external devices such as the memory and I/O. The interrupt asks the processor to execute a special program.

Internal Data Bus :

The internal data bus connects the different parts of processor together and it enables the communication between these parts. The data transfer through this internal data bus is controlled by control logic.

Processor's internal data bus usually connected to an external data bus. Due to this processor can communicate with external memory or I/O devices. Usually the internal data bus is connected to the external data bus by logic called a bi-directional bus (transreceiver).

Review Questions

1. *Explain the major functions of processor.*
2. *Draw the general processor organization.*

6.2 General Register Organization

Bus organization

The Fig. 6.2.1 shows the general bus organization for seven CPU registers. It shows that how registers are selected and how data flow between register and ALU take place. A decoder is used to select a particular register. 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 the input data for the particular bus. The A and B buses form the two inputs of an Arithmetic Logic Unit (ALU). The operation select lines decide the micro-operation to be performed by ALU. The result of the micro-operation is available at the output bus. The output bus connected to the inputs of all registers, thus by selecting a destination register it is possible to store the result in it.

(See Fig. 6.2.1 on next page)

Example

To perform the operation R3 ← R1 + R2 we have to provide following binary selection variables to the select inputs. (Refer Tables 6.2.1 and 6.2.2)

1. SELA : 001 - To place the contents of R1 into bus A
2. SELB : 010 - To place the contents of R2 into bus B
3. SELOPR : 10010 - To perform the arithmetic addition A + B
4. SELREG : 011 - To place the result available on output bus in R3.

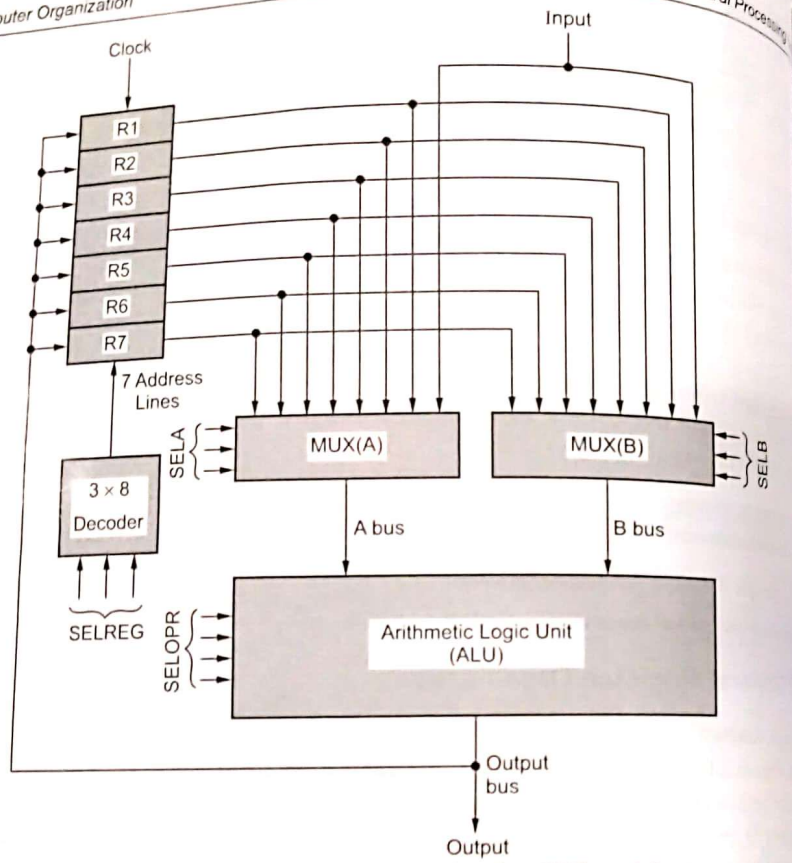


Fig. 6.2.1 General bus organization for CPU registers

Binary code	SELA	SELB	SELREG
0 0 0	Input	Input	-
0 0 1	R1	R1	R1
0 1 0	R2	R2	R2
0 1 1	R3	R3	R3
1 0 0	R4	R4	R4
1 0 1	R5	R5	R5
1 1 0	R6	R6	R6
1 1 1	R7	R7	R7

Table 6.2.1 Register and multiplexer input selection codes

Operation selection code	Operation
0 0 0 0	Transfer A
0 0 0 1	Increment A
0 0 1 0	A + B
0 0 1 1	A – B
0 1 0 0	Decrement A
0 1 0 1	A AND B
0 1 1 0	A OR B
0 1 1 1	A XOR B
1 0 0 0	Complement A
1 0 0 1	Shift right A
1 0 1 0	Shift left A

Table 6.2.2 ALU operation selection codes

Concept of control word

The combined value of a binary selection inputs specifies the control word. The Fig. 6.2.2 shows the control word format. It consists of four fields SELA, SELB and SELREG contain three bits each and SELOPR field contains four bits. Thus the total bits in the control word are 13-bits.

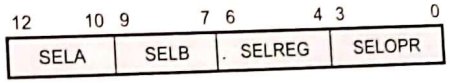


Fig. 6.2.2 Format of control word

The three bits of SELA select a source registers of the A input of the ALU. The three bits of SELB select a source registers of the B input of the ALU. The three bits of SELREG select a destination register using the decoder. The four bits of SELOPR select the operation to be performed by ALU.

Example 6.2.1 *Obtain the control word for operation* $R2 \leftarrow R1 + R3$

Solution : Referring Tables 6.2.1 and 6.2.2 we have

SELA	SELB	SELREG	SELOPR
0 0 1	0 1 1	0 1 0	0 0 1 0

The control words for all microoperation are stored in the memory called control memory. By reading consecutive control words from memory it is possible to initiate the desired sequence of microoperation for the CPU. Such control is referred to as **microprogrammed control.**

Review Question

1. *Describe general register organization.*

6.3 Stack Organization

Data operated on by a program can be stored in the CPU registers and in the computer memory. It can be organized properly for easy access. For example, the data items of similar data type can be organized in the form of arrays and can be accessed using array pointers. Like array data structure, most of the computers supports an important data structure known as a **stack.**

A stack is a list of data elements, usually words or bytes, with the accessing restriction that the elements can be added or removed at one end of the list only. This end is called the top of the stack, and the other end is called the bottom of the stack. As only one end is used for adding and removing data elements, the element stored last is the first element retrieved. Thus stack structure is also known as Last-In-First-Out (LIFO) list. The terms **Push** and **Pop** are used to describe placing a new data element on the stack and removing the top data element from the stack, respectively.

The stack in the digital computer is a part of register unit or memory unit with a register that holds the address for the stack. The part of register array or memory used for stack is called **stack area** and the register used to hold the address of stack is called **stack pointer.** The value in the stack pointer always points at the top data element in the stack.

6.3.1 Register Stack

As mentioned earlier, a stack can be placed in a portion of a memory unit or it can be organized as a collection of a finite number of CPU registers. The Fig. 6.3.1 shows the organization of a 32-word register stack. The stack pointer holds the address of the register that is currently the top of stack. As shown in the Fig. 6.3.1 four data elements 10, 20, 30 and 40 are placed in the stack. The data element 40 is on the top of stack therefore, the content of SP is now 4.

To remove the top data element, the stack is popped by reading the register at address 4 and decrementing the content of SP. The data element 30 is now on top of the stack since SP holds address 3. To insert a new data element, stack is pushed by incrementing SP and writing a data element in the incremented location in the stack.

In 32 word stack, the stack pointer is a 5-bit register, because $2^5 = 32$. Initially, it is clear to 0 and stack is said to be empty. When data element is pushed on the stack, the stack pointer is incremented so that it points to the address of the next-higher register and then the data element is written in the register pointed by stack pointer. If SP reaches 31 and one more data element is pushed then SP is incremented to 0 and data element is stored in the register 0. Now there is no more empty registers in the stack and stack is said to be **FULL.** The Fig. 6.3.2 shows the operation of stack when specified sequence of instructions are executed.

(See Fig. 6.3.2 on next page.)

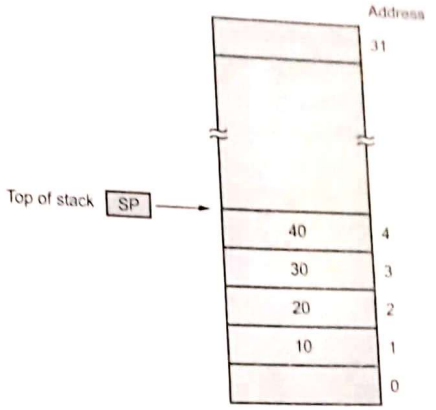


Fig. 6.3.1 Block diagram of a 32-word register stack

6.3.2 Memory Stack

The operation of memory stack is exactly similar to the register stack. However, it is implemented using computer memory instead of CPU register array. We know that, the number of registers in the CPU are limited and it restricts the size of stack in the stack computer. But when stack is implemented using memory its size is extended upto the memory addressing capacity of the CPU. But truely speaking computer memory is also shared by program and data, as shown in the Fig. 6.3.3. Hence a part of computer memory is used for the stack. The memory stack has an advantage of large size but the operation on it is slower than that of register stack. This is because register stack is internal to the CPU and does not need any memory access.

In the stack organized computers instruction's operands are stored at the top of the stack, so data processing instructions do not need to contain addresses as they do in a conventional, Von Neumann computer. The add operation A + B is specified for a stack computer by the following sequence of three instructions. (See Fig. 6.3.3 on next page).

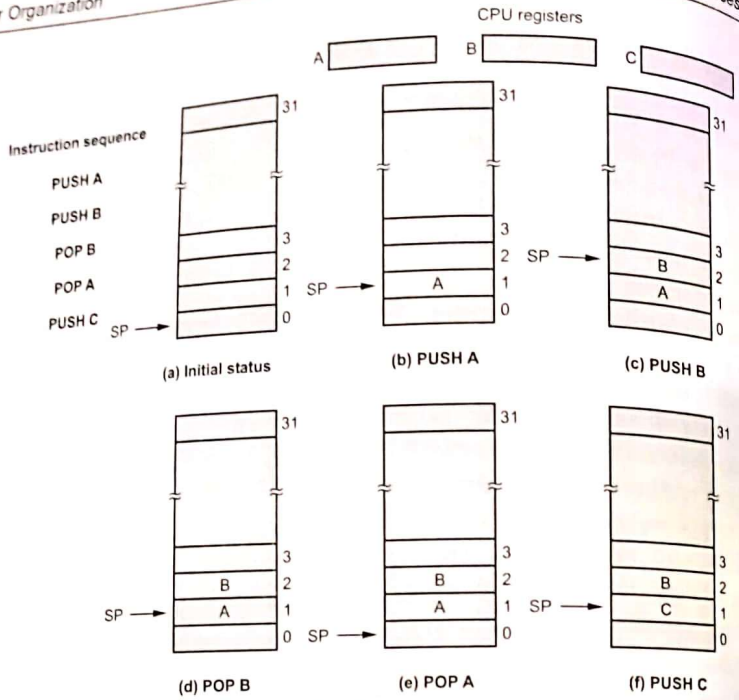


Fig. 6.3.2

PUSH A

PUSH B

ADD

The first PUSH instruction loads A into top of stack (TOS). Execution of PUSH B causes A's location to become TOS + 1 and places B in the new TOS immediately above A. To execute ADD, the top two words of the stack are popped into the ALU where they are added, and the sum is pushed back into the stack. As shown by program sequence, in the stack computer, operator is specified after the operands i.e. A B + instead of A + B. This method of representing arithmetic expression is known as **Reverse Polish notation**, named after the Polish logician

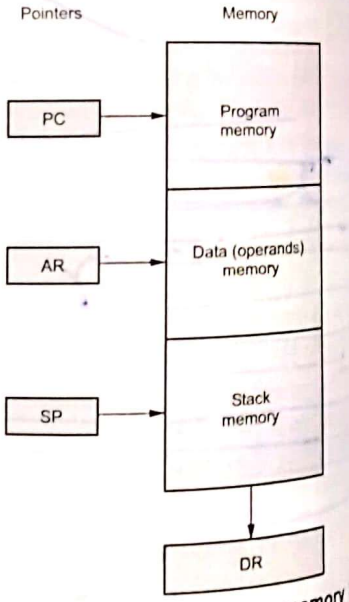


Fig. 6.3.3 Sharing of computer memory by program, data and stack area

Jan Lukasiewicz. Therefore, we can say that stack computer evaluates arithmetic and other expressions using a reverse polish notation format.

In most conventional computers, stack is used to implement subroutine call and return instructions. In some situations, it is better to execute part of a program that is not in sequence (don't confuse with logical sequence) with the main program. For example, there may be a part of a program that must be repeated many times during the execution of the entire program. Rather than writing repeated part of the program again and again, the programmer can write that part only once. This part is written separately. The part of the program which is written separately is called **subroutine**. The Fig. 6.3.4 shows how the main and subroutine program are executed.

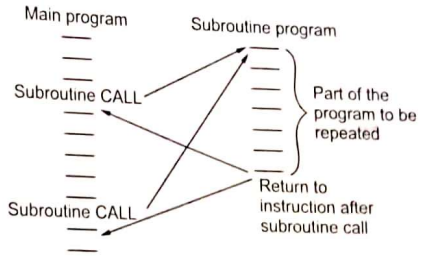


Fig. 6.3.4 Execution of subroutine programs

When the subroutine is called by CALL instruction the current content of PC are pushed on the stack and after execution of subroutine program the content of PC are popped back into the PC to execute the next instruction after the CALL.

6.3.3 Notations

Basically, there are three methods of representing arithmetic expressions. These are

A + B : Infix notation

+ A B : Prefix or polish notation

A B + : Postfix or reverse polish notation

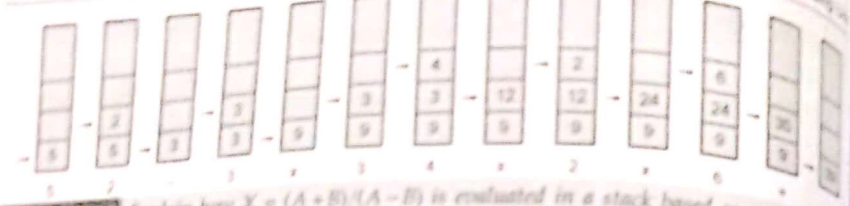
As mentioned earlier, the stack organization uses reverse polish notations to evaluate arithmetic expressions and this is very effective method of evaluating arithmetic expressions on computers.

Example 6.3.1 *Show the stack states during the execution of* $Z = w + (3 \times (x - y)) + 2 (v \times u)$.

Solution : In reverse polish notation, the given expression can be expressed as

$$x\ y - 3 \times v\ u \times 2 \times w + +$$

Let us assume that $x = 5$, $y = 2$, $v = 3$, $u = 4$ and $w = 6$. During the execution of given expression with assumed values, the stack states are as shown below.



Example 6.3.3 *Explain how X = (A + B)/(A − B) is evaluated in a stack based computer.*

Solution : We know that the stack organization uses reverse polish notations. In reverse polish notation, the given expression can be expressed as

AB + AB − /

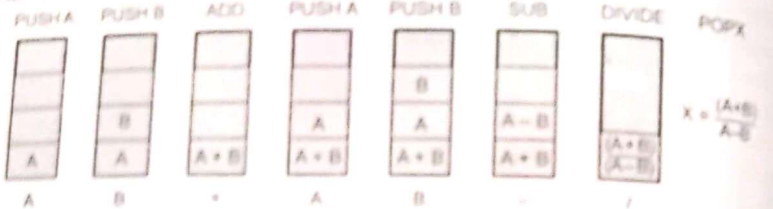


Fig. 6.3.5

Example 6.3.5 *Convert the following into reverse polish notation.*

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

2) A [B + C* (D + E)] / [F/G*(H + I)]*

Solution :

1. Reverse polish notation A + B * [C + D + E * (F + G)] is A B F G + E * C D + * B + +

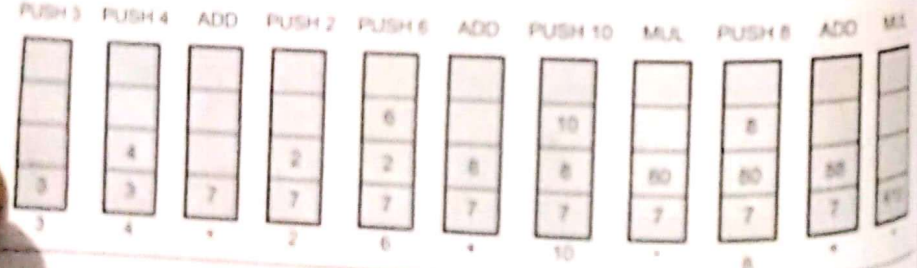
2. Reverse polish notation for A + [B + C * (D + E)]/ [F/G * (H + I)] is A D E + C B + * H I + F G / * /

Example 6.3.4 *Explain stack and evaluate the following expression using stack*

*(3 + 4) * [10 (2 + 6) + 8]*

Solution : We know that the stack organization uses reverse polish notations. In reverse polish notation, the given expression can be expressed as

34 + 26 + 10 + 8 + *



Review Questions

1. *Write short note on stack organization.*
2. *Which is the notation used to represent arithmetic expression in stack organization ?*
3. *Explain the operation of the memory stack.*
4. *What do you mean by reverse polish notation ?*
5. *What is stack ? Give the organization of register stack with all necessary elements and explain the working of push and pop operations.* **GTU : Dec.-09, 12-13, May-13, Marks 7**
6. *Explain register stack and memory stack with block diagram.* **GTU : May-12, Marks 7**
7. *What is a memory stack ? Explain its role in managing subroutines with the help of neat diagrams.* **GTU : May-11, Marks 7**

6.4 Instruction Format

Each instruction of the CPU contain specific information fields, which are required to execute it. These information fields of instructions are called elements of instruction. These are :-

Operation code : The operation code field in the instruction specifies the operation to be performed. The operation is specified by binary code, hence the name operation code or simply opcode for 8085 processor operation code for Add B instruction is 80H.

Source / Destination operand : The source/destination operand field directly specifies the source/destination operand for the instruction. In 8085, the instruction MOV A, B has B register as a source operand and A register as a destination operand, because this instruction copies the contents of register B to register A.

Source operand address : We know that the operation specified by the instruction may require one or more operands. The source operand may be in the CPU register or in the memory. Many times the instruction specifies the address of the source operand so that operand(s) can be accessed and operated by the CPU according to the instruction.

In 8085, the source operand address for instruction Add M is given by HL register pair.

Destination operand address : The operation executed by the CPU may produce result. Most of the times the result is stored in one of the operand. Such operand is known as destination operand. The instruction which produce result specifies the destination operand address. In 8085, the destination operand address for instruction INR M is given by HL register pair. Because INR M instruction increments the contents

of memory location specified by HL register pair and stores the result in the same memory location.

Next instruction address : The next instruction address tells the CPU from where to fetch the next instruction after completion of execution of current instruction. For JUMP and BRANCH instructions the address of the next instruction is specified within the instruction. However, for other instructions, the next instruction to be fetched immediately follows the current instruction. For example, in 8085, instruction after INR follows its. The instruction JMP 2000H specifies the next instruction address as 2000H.

Computers may have instructions of several different lengths containing varying number of addresses. According to address reference there are three address, two address, one address and zero address reference instructions. Let us see examples of each of them.

Three address instructions

The three address instruction can be represented symbolically as

ADD A, B, C

where A, B, C are the variables. These variable names are assigned to distinct locations in the memory. In this instruction operands A and B are called source operands and operand C is called destination operand and ADD is the operation to be performed on the operands. Thus the general instruction format for three address instruction is

Operation Source 1, Source 2, Destination

The number of bits required to represent such instruction include :

1. Bits required to specify the three memory addresses of the three operands. If n-bits are required to specify one memory address, 3n bits are required to specify three memory addresses.
2. Bits required to specify the operation.

Two address instructions

The two address instruction can be represented symbolically as

ADD A, B

This instruction adds the contents of variables A and B and stores the sum in variable B destroying its previous contents. Here, operand A is source operand however operand B serves as both source and destination operand. The general instruction format for two address instruction is

Operation Source, Destination

To represent this instruction less number of bits are required as compared to three address instruction. The number of bits required to represent two address instruction include :

1. Bits required to specify the two memory addresses of the two operands, i.e. 2n bits.
2. Bits required to specify the operation.

One address instruction

The one address instruction can be represented symbolically as

ADD B

This instruction adds the contents of variable A into the processor register called **accumulator** and stores the sum back into the accumulator destroying the previous contents of the accumulator. In this instruction the second operand is assumed implicitly in a unique location accumulator. The general instruction format for one address instruction is

Operation Source

Few more examples of one address instructions are :

LOAD A : This instruction copies the contents of memory location A into the accumulator.

STORE B : This instruction copies the contents of accumulator into memory location B.

In one address instruction, it is important to note that the operand specified in the instruction can be either source operand or destination operand depending on the instruction. For example, in LOAD A instruction, the operand specified in the instruction is a source operand whereas the operand specified in the STORE B instruction is a destination operand. Similarly, in one address instruction the implied operand (accumulator) can be either source or destination depending on the instruction.

Zero address instructions

In these instructions, the locations of all operands are defined implicitly. Such instructions are found in machines that store operands in a structure called a **pushdown stack.**

From above discussion we can easily understand that the instruction with only one address will require less number of bits to represent it, and instruction with three addresses will require more number of bits to represent it. Therefore, to access entire instruction from the memory, the instruction with three addresses requires more memory accesses while instruction with one address requires less memory accesses. The speed of instruction execution is mainly depend on how much memory accesses it requires for the execution. If memory accesses are more, more time is required to

execute the instruction. Therefore, the execution time for three address instructions is more than the execution time for one address instructions.

To have a less execution time we have to use instructions with minimum memory accesses. For this instead of referring the operands from memory it is advised to refer operands from processor registers. When machine level language programs are generated by compilers from high-level languages, the intelligent compilers see that the maximum references to the operands lie in the processor registers.

Example 6.4.1 *Write a program to evaluate the arithmetic statement* $Y = (A+B)*(C+D)$ *using three-address, two-address, one-address and zero-address instructions.*

Solution : Using three address instructions

```
ADD   R1, A, B   ;   R1 ← M[A] + M[B]
ADD   R2, C, D   ;   R2 ← M[C] + M[D]
MUL   Y, R1, R2  ;   M[Y] ← R1 * R2
```

Using two address instructions

```
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 Y, R1    ;   M[Y] ← R1
```

Using one address instruction

```
LOAD   A   ;   AC ← M[A]
ADD    B   ;   AC ← AC + M[B]
STORE  T   ;   M[T] ← AC
LOAD   C   ;   AC ← M[C]
ADD    D   ;   AC ← AC + M[D]
MUL    T   ;   AC ← AC * M[T]
STORE  Y   ;   M[Y] ← AC
```

Using zero address instructions

```
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    Y   ;   M[Y] ← TOS
```


Example 6.4.2 *Write the program to evaluate the expression* $X = \frac{A[B+C(D+E)]}{F(G+H)}$ *using the zero address instructions and one address instructions.*

Solution : Program using zero address instructions

```
PUSH   D   ;   TOS ← D
PUSH   E   ;   TOS ← E
ADD        ;   TOS ← (D + E)
PUSH   C   ;   TOS ← C
MUL        ;   TOS ← C × (D + E)
PUSH   B   ;   TOS ← B
ADD        ;   TOS ← B + C × (D + E)
PUSH   A   ;   TOS ← A
MUL        ;   TOS ← A [B + C × (D + E)]
PUSH   G   ;   TOS ← G
PUSH   H   ;   TOS ← H
ADD        ;   TOS ← G + H
PUSH   F   ;   TOS ← F
MUL        ;   TOS ← F × (G + H)
DIV        ;   TOS ← A[B + C × (D + E)]/F × (G + H)
POP    X   ;   M [X] ← TOS
```

Program using one address instructions

```
LOAD   H   ;   AC ← M[H]
ADD    G   ;   AC ← AC + M[G]
MUL    F   ;   AC ← AC * M[F]
STORE  T   ;   M[T] ← AC
LOAD   D   ;   AC ← M[D]
ADD    E   ;   AC ← AC + M[E]
MUL    C   ;   AC ← AC * M[C]
ADD    B   ;   AC ← AC + M[B]
MUL    A   ;   AC ← AC * M[A]
DIV    T   ;   AC ← AC/M[T]
STORE  X   ;   M[X] ← AC
```

An instruction format defines the layout of the bits of an instruction. It must include opcode, zero or more operands and addressing mode for each operand. The instruction length is usually kept in multiple of the character length or memory transfer length which is usually 8-bits. With this length we will always get an integral number of instructions during a fetch cycle. Once the instruction length is fixed, it is necessary to allocate number of bits for opcode, operand(s) and addressing modes. For an instruction

```
ADD     T   ;   AC ← AC + M[T]
STORE   X   ;   M[X] ← AC
```

Using two address instructions

```
MOV  A, R1   ;   R1 ← M[A]
MUL  B, R1   ;   R1 ← R1 * M[B]
MOV  C, R2   ;   R2 ← M[C]
MUL  C, R2   ;   R2 ← R2 * M[C]
ADD  R2, R1  ;   R1 ← R1 + R2
MOV  R1, X   ;   M[T] ← R1
```

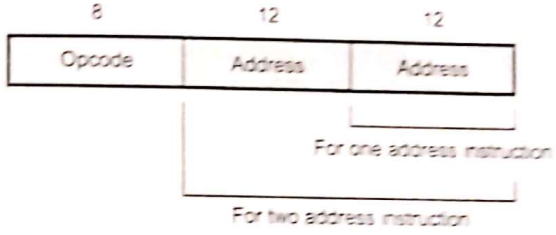
Using three address instructions

```
MUL  A, B, R1   ;   R1 ← M[A] * M[B]
MUL  C, C, R2   ;   R2 ← M[C] + M[C]
ADD  R1, R1, X  ;   M[X] ←R1 * R2
```

Example 6.4.4 *A computer has 32-bit instructions and 12-bit addresses. If there are 250 two-address instructions, how many one address instructions can be formulated ?*

Solution : The given instruction is 32-bit. These 32-bit has following format.

The total number of instruction combinations for two address instructions are



$$2^8 = 256$$

In these 256 combinations, 250 combinations are used for two address instructions. Therefore, remaining 6 combinations are used for one address instructions. Thus, the maximum number of one address instructions

$$= (256 - 250) \times 2^{12} = 6 \times 2^{12}$$

$$= \textbf{24576 instructions}$$

Example 6.4.5 *A machine support 16-bit instruction format, the size of address field is 4-bits. The computer support expanding opcode technique and has 55000 zero-address instruction and 100 one address instructions. Find out the number of two address instructions supported by this machine.*

Solution : A 16-bit instruction can supports 2^{16} = 65536 zero address instruction. It is given that machine supports 55000 zero address instructions. Thus we have

65536 – 5500 = 10536 instructions combinations to in corporate one address and
address instructions. A 100 one address instructions requires 100×2^4 (four bit add
field) = 1600 instruction combinations. Thus 10536 – 1600 = 8936 instruction combinat
left for two-address instructions. Let us consider the number of two address instruc
supported by this machine is x then

$$8936 > x \times 2^8 \text{ (eight bit address field)}$$

$$\therefore \quad x = 34.9 = \mathbf{34 \text{ instructions}}$$

So there are 34 two-address instructions supported by this machine.

Review Questions

1. *What is instruction format ?*
2. *Explain zero-address, one-address, two-address and three-address instructions.*

GTU : June-08, Jan.-10, Dec.-10, Marks 10

3. *Using a stack organized computer with zero address operation instructions. Write a program to evaluate the arithmetic statement.*

$$X = \frac{a+b+c^*(d+e)}{f-g^*h}$$

GTU : Dec.-02, Marks

4. *Explain four types of instruction formats.*

GTU : Feb.-10, Dec.-11, 12, Marks

6.5 Addressing Modes

The address of the memory can be specified directly within the instruction. F
example, MOV [2000H], R1. In this instruction the memory address is fix; it can not
dynamically changed in the program itself. There are some situations where we need
change the memory address dynamically. Let us see the example program. In t
program the contents from the array of data are added to get the total sum of all ar
elements. We know for this we have repeat the add instruction number of times equal
the array elements and each time we have to add a number from a new success
memory location. Every time the address of memory is different. So to change t
address of memory each time when we enter the loop address variable is used. So
addressing is called **indirect addressing**. For example, ADD R1, [R2]. Here, the conte
of R2 register are used as an address of memory location. By incrementing the conte
of register R2 it is possible to change the memory address each time we enter the loop

Note *The instruction used in the program given below specifies first destination operand and then the source operand.*

```
MOV R2, Array-start    ; Load the starting address of the array
MOV R0, Count          ; Initialize the counter
```



```
        MOV R1, 0          ; Result = 0
        ADD R1, [R2]       ; Result ← Result + array element
BACK :  INC R2             ; Increment memory pointer
        DEC R0             ; Decrement count
        JNZ BACK           ; If count ≠ 0, repeat
```

Part of the programming flexibility for each processor is the number and different kind of ways the programmer can refer to data stored in the memory or I/O device. The different ways that a processor can access data are referred to as **addressing schemes** or **addressing modes.**

6.5.1 Implementation of Variables and Constants

The variables and constants are the most commonly used data types on every computer. In assembly language, a variable is initialized by assigning register or a memory location to hold its value. Thus by changing the contents of specified register or memory location it is possible to change the value of the variable. Let us see the addressing modes used for this purpose.

1. **Register mode :** The operand is the contents of processor register. The name of register is specified in the instruction.

Example : MOV R1, R2

This instruction copies the contents of register R2 to register R1.

2. **Absolute mode or direct mode :** The address of the location of the operand is given explicitly as a part of the instruction.

Example : MOVE A, 2000

The above instruction copies the contents of memory location 2000 into the A register. As shown in the instruction, here, address of operand is given explicitly in the instruction.

The constant for address and data can be represented by immediate addressing mode in the assembly language programming. Let us see immediate addressing mode.

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

Example : MOVE A, #20

The above instruction copies operand 20 in the register A. The sign # in front of the value of an operand is used to indicate that this value is an immediate operand.

6.5.2 Indirection and Pointers

Some processor instructions does not provide an operand or its address explicitly. Instead, they provide the information from which the memory address of the operand can be determined. Such address is referred to as **effective address** (EA) of the operand.

The two dimensional array is something which you can compare with the two storied building! Extra space which is arranged in rows and columns. Let me draw a figure which will represent the two dimensional array.

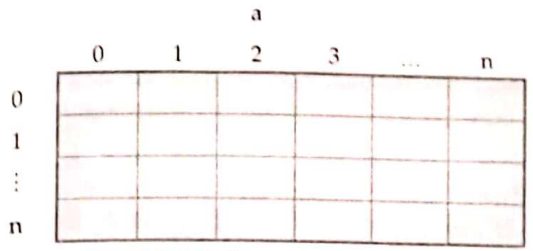


Fig. 6.5.2 2-dimensional array

The two dimensional array elements can be accessed by index addressing mode. Using index addressing mode we can specify the row address in the register and offset gives the column address. For example : MOV R2, 10(R1). In this instruction R1 specifies the row the address and offset 10 gives the column address.

6.5.4 Relative Addressing

We have seen the index addressing mode using general-purpose processor registers. If we use this addressing mode with program counter instead of the general purpose register we get the relative addressing mode.

6. Relative mode : The effective address is determined by the index mode using program counter in place of the general purpose processor register. EA = PC + Address part of instruction

This addressing mode is commonly used to specify the target address in branch instructions. For example,

```
JNZ BACK
```

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

6.5.5 Additional Modes

So far we have discussed the six addressing modes : Register, absolute (direct), immediate, indirect, index and relative addressing modes. In this section we will see two more addressing modes : autoincrement and autodecrement.

7. Autoincrement mode : The effective address of the operand is the contents of a register specified in the instruction. After accessing the operand, the contents of this register are incremented to address the next location.

Fig. 6.5.3

i) Direct address : EA = 500 ∴ AC = 800

ii) Indirect address : EA = [500] = 800 ∴ AC = 300

iii) Relative address : EA = PC + Address part of instruction

= 202 + 500 = 702

∴ AC = 325

iv) Indexed address : EA = XR + Address part of instruction

= 100 + 500 = 600

∴ AC = 900

v) Register indirect address : EA = R1 = 400

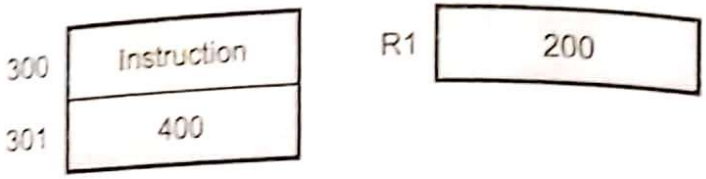
∴ AC = 700

Example 6.5.3 *An instruction is stored at location 300 with its address field at location 301. The address field has the value 400. A processor register R1 contains the number 200. Evaluate the effective address if the addressing mode of the instruction is :*

i) Immediate ii) Direct iii) Register indirect

iv) Relative v) Index with R1 as the index register.

Solution :



i) Immediate mode : In immediate addressing mode, the address bits of instruction code specify the actual operand. Hence for this addressing mode the address of the operand i.e. effective address is 301.

ii) Direct mode : Effective Address (EA) : **400**

iii) Register indirect node : Effective Address (EA) : **200**

iv) Relative mode : Effective Address (EA) : PC + Address part of instruction = 302 + 400 = **702**

v) Index with R1 as the index register : Effective address (EA)

= R1 + Address part of the instruction = 200 + 400

= 600

Review Questions

1. *What is addressing mode ? List and explain all addressing modes used in basic computer.*

Data manipulation ...

- Arithmetic operations
- Logical operations
- Shift and rotate operations

The Table 6.6.2 gives the list of typical arithmetic instructions.

Instruction	Mnemonic	Description
Add	ADD	It adds the contents of two operands.
Subtract	SUB	It subtracts the contents of two operands.
Increment	INC	It adds 1 to the value stored in a register or memory word.
Decrement	DEC	It subtracts 1 from the value stored in a register or memory word.
Multiply	MUL	It multiplies the contents of two operands.
Divide	DIV	It divides the contents of two operands.
Add with carry	ADC	It adds the contents of two operands with carry.
Subtract with borrow	SBB	It subtracts the contents of two operands with borrow.
Negate (2's complement)	NEG	It gives the 2's complement of the specified operand.

Table 6.6.2 List of typical arithmetic instructions

Note Add, subtract, multiply and divide instructions may be available for different data types.

The Table 6.6.3 gives the list of typical logical instructions.

Instruction	Mnemonic	Description
AND	AND	It logically ANDs the individual bits of the operands.
OR	OR	It logically ORs the individual bits of the operands.
Exclusive OR	XOR	It logically Ex-ORs the individual bits of the operands.
Clear	CLR	It causes the specified operand to be replaced by 0s.
Complement	COM	It gives the 1's complement of the specified operand.

Clear carry	CLC	It resets the carry bit to 0.
Set carry	STC	It sets the carry bit to 1.
Complement carry	CMC	It complements (reverse) the carry bit.
Enable interrupt	EI	It enables interrupt facility.
Disable interrupt	DI	It disables interrupt facility.

Table 6.6.3 List of typical logical instructions

The Table 6.6.4 gives list and typical shift and rotate instructions.

Instruction	Mnemonic	Description
Logical shift right	SHR	It shifts the contents of specified register 1-bit position towards right and fills vacant bit (leftmost) with zero.
Logical shift left	SHL	It shifts the contents of specified register 1-bit position towards left and fills vacant bit (rightmost) with zero.
Arithmetic shift right	ASHR	It shifts the contents of specified register 1-bit position towards right and fills vacant bit (leftmost) with previous sign bit.
Arithmetic shift left	ASHL	It shifts the contents of specified register 1-bit position towards left and fills vacant bit (rightmost) with zero.
Rotate right	ROR	It rotates (circular-shifts) right the contents of specified register.
Rotate left	ROL	It rotates (circular-shift) left the contents of specified register.
Rotate right through carry	RORC	It transfers the carry bit into the leftmost bit position of the register, transfers the rightmost bit into the carry, and at the same time, shifts the entire register to the left by 1-bit position.
Rotate left through carry	ROLC	It transfers the carry bit into the rightmost bit position of the register, transfers the leftmost bit into the carry, and at the same time, shifts the entire register to the left by 1-bit position.

Table 6.6.4 List of typical shift and rotate instructions

C-bit : Carry Bit. It is set when the output carry of ALU is 1; otherwise it is 0.

S-bit : Sign Bit. It is set when MSB (Most Significant Bit) of the result is 1; otherwise it is zero. It indicates the sign of the result. When S bit is 1, result is negative; otherwise result is positive.

Z-bit : Zero Bit. It is set when the result is zero; otherwise it is zero.

V-bit : Overflow Bit. In 2's complement arithmetic, most significant bit is used to represent sign and remaining bits are used to represent magnitude of a number. It is set if the result of a signed operation is too large to fit in the number of bits available (7-bits for 8-bit number) to represent it. The overflow can be detected by exclusively - ORing two most significant carry out signals.

Fig. 6.7.1 shows the logic circuit which detects above mentioned conditions and sets the flag bits accordingly.

The collection of all status bits conditions in the CPU is sometimes called a **program status word** (PSW).

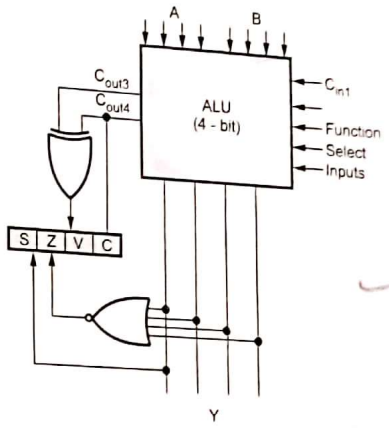


Fig. 6.7.1 Setting bits in a status register

Conditional Branch Instructions

Conditional branch instructions first checks the status of the specified flag and accordingly decides whether to transfer program control to the given address or to execute next instruction in the sequence. The Table 6.7.2 shows some conditional branch instructions and tested conditions.

Mnemonic	Branch condition	Tested condition
BV	Branch if over flow	V = 1
BNV	Branch if no overflow	V = 0
BZ	Branch if zero	Z = 1
BNZ	Branch if no zero	Z = 0
BM	Branch if minus	S = 1
BP	Branch if plus	S = 0
BC	Branch if carry	C = 1

BNC	Branch if no carry	C = 0
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

Table 6.7.2 Conditional branch instructions

rrupt

outine program, the program control is transfer to the interrupt ser sponse to interrupt. There are several sources of interrupts :

errupts : They come from input-output devices or from a timing device

errupt : Internal interrupts arise from illegal or erroneous use of or data.

ifference between internal and external interrupts is that the internal interrupt some exceptional condition caused by the program itself rather than by an extern

No.	Characteristics	CISC	RISC
1.	Instruction size	Varies	Fixed
2.	Instruction length	1, 2, 3 or 4 bytes	4 bytes
3.	Number of Instruction	More	Less
4.	Instruction decoding	Serial (slow) to decode	Easy (quick) to decode
5.	Instruction sementics	Varies from simple to complex ; possibly many dependent operations per instruction	Almost always one simple operation
6.	Addressing Modes	Supports complex addressing modes.	Complex addressing modes are synthesized in software.
7.	Instruction execution speed	Slow (depend on complexity of instruction)	Medium
8.	Instruction execution	By microprogram. Complex instructions taking multiple cycles.	By hardware. Simple instructions taking one cycle.
9.	Registers	Few, may be special purpose	Many, general purpose
10.	Memory references	Combined with operations in many different types of instructions	Not combined with operations, i.e., load/store architecture
11.	Hardware	Complicated	Simple
12.	Hardware design focus	Take the advantage of microcoded implementations	Take the advantage of implementations with one pipeline and no microcode
13.	Memory access	Frequently	Rarely
14.	Instruction format	Field placement varies	Regular, consistent placement of fields

One instruction per cycle

In RISC processors, there is an one instruction per machine cycle. A machine cycle is defined to be the time it takes to fetch two operands from registers, performs an ALU operation, and stores the result in a register. Due to this, RISC machine instructions are not complicated and can execute about as fast as, microinstructions on CISC machines.

Hardwired instructions

With simple, one cycle instruction, there is no need for microinstructions. The machine instructions can be hardwired. These instructions are executed faster than the instructions implemented with microinstructions, since it is not necessary to access a microprogram control memory during instruction execution.

Reduced number of instructions

RISC processor provides limited number of instructions, which simplifies the design of control unit.

Simple addressing modes

RISC processor uses simple addressing modes. Almost all instructions use simple register addressing. Several other addressing modes are also used, such as displacement and PC-relative. It does not support complex-addressing modes. They can be synthesized in software from the simple ones. This architecture of RISC processor simplifies the instruction set and the control unit.

Simple instruction format

RISC processor uses simple instruction formats with fixed instruction length. The instruction length is aligned on word boundaries. Field locations, especially opcodes are fixed.

This architecture provides following benefits :

- With fixed fields, opcode decoding and register operand addressing can occur simultaneously.
- It simplifies the design of control unit.
- Instruction fetching is optimized since word-length units are fetched.

6.8.3 RISC Addressing Modes

The small instruction set of typical RISC processor consists mostly of register-to-register operations, and simple load and store operations for memory access. Each operand is brought into a processor register with a load instruction and results are transferred to memory by means of store instruction. In this architecture almost all

instructions have simple register addressing, and thus it uses only a few addressing modes.

Usually, the RISC processor has three basic addressing modes :

- Register addressing
- Immediate operand and
- Relative to PC addressing for branch instructions

Register addressing : In register addressing, the instruction usually consists of three fields : opcode field which specifies an operation, one or two source register fields and one destination register field. The operation is performed with the data specified in the source register fields and the result is stored in the destination register field. For example : ADD R_1, R_2, R_3 ; $R_3 \leftarrow R_1 + R_2$. In case of memory access one source register specifies the memory address and second source register specifies the offset. For example, LD (R_1) R_2 ; $R_3 \leftarrow M [R_1 + R_2]$.

Immediate operand addressing : In immediate operand addressing mode, the second source is an immediate operand. The operation is performed with the data specified in the source register field and the immediate operand, and the result is stored in the destination register field. For example : ADD R_1, # 100, R_2 ; $R_2 \leftarrow R_1 + 100$

Relative to PC addressing : In relative to PC addressing, the instruction usually consists of three fields : opcode field, condition field and address field. Opcode field specifies the operation. The condition field specifies one of many possible branch conditions, and address field specifies the signed offset which is to be added to the contents of PC to calculate new address when branch condition is satisfied. For example, JMP COND, $R_1(R_2)$; $PC \leftarrow R_1 + R_2$ if condition is satisfied.

6.8.4 CISC and RISC Application Areas

The CISC technologies had established themselves solidly as the technology of choice for the desktop and various other business computing and information management platforms. A CISC microcomputer is equipped with a complex instruction set including a large number of instructions such as bit control and other instructions. Although it is generally difficult to achieve the high operating frequencies of RISC (Reduced Instruction Set Computer) microcomputers, CISC microcomputers have certain advantages such as ease of programming which have led to their use in a variety of products and PC peripherals such as optical storage devices (CD-R/RW, DVD-R/RW, DVD-ROM/RAM, etc.) OA devices and consumer products.

The RISC technology has been gaining momentum in the applications areas where less of a base of legacy applications were available or where performance justified migration. Notably the workstation market that emerged in the 80s became the domain of the RISC architecture, as well as the rapidly growing environment of embedded

applications. It would be overstating the situation to say that RISC is in the process of replacing CISC but it is rue to say that RISC is taking an increasing share of the micro-components market. Typical RISC application areas are :

- Wireless LANs
- Network interface cards (NICs)
- Server farms and Internet appliances
- Traditional portable applications such as PocketPCs and cell phones.
- Telematics and Intelligent Traffic Systems
- Transport logistics
- Warehouse logistics
- Location planning
- Mobile information systems
- Mobile Web Services
- Mobile Frameworks
- Touch based Human Machine Interface (HMI)
- Data collector
- Security Controller

Review Questions

1. *Write a short note on RISC architecture.*
2. *Give the comparison between RISC and CISC architectures.*
3. *Explain the properties of RISC architecture.*
4. *Explain the advantages and disadvantages of RISC architecture.*
5. *Explain the addressing modes of RISC architecture.*
6. *Write a short note on overlapping register window.*
7. *Give the applications of RISC processor.*
8. *Explain the characteristics of RISC and CISC.*

   GTU : Dec.-02,06,12,13, May-12, June-06, Feb.-10, Marks 8
9. *Differentiate between RISC and CISC.* **GTU : Dec.-05, 11, June-08, May-13, Marks 4**
10. *Importance of overlapped window of registers.*

    GTU : June-06, Dec.-12, May-12, Feb.-10, Marks 7
11. *How to speed up procedure call and return using overlapped register window?*

    GTU : June-08, May-13, Marks 4

12. *List the important characteristics of RISC architecture and explain the use of overlapped register windows.*

    GTU : Dec.-09, Marks 7
13. *What is overlapped register window ? How window size and register file size is computed ?*

    GTU : Winter-13, Marks 7
14. *What do you understand by reduced instruction set computers ? What are complex instruction set computers ? List important characteristics of CISC and RISC computers. Also in a tabular from compare their advantages / disadvantages.*

    GTU : May-11, Marks 7

❑❑❑

Syllabus

Flynn's taxonomy, Parallel processing, Pipelining, Arithmetic pipeline, Instruction, Pipeline, RISC pipeline, Vector processing, Array processors

Contents

7.1 Parallel Processing	Dec.-09, Feb.-10,	Marks 7
7.2 Pipeline Processing	Feb.-10, May-12,	Marks 5
7.3 Arithmetic Pipeline 7.4 Instruction Pipeline	Dec.-03,05,09,10,11,12, Feb.-10, May-12,14	Marks 8
7.5 RISC Pipeline 7.6 Vector Processing	May-11,13,	Marks 7
7.7 Array Processors	Dec.-11,12,	Marks 7

7.1 Parallel Processing

GTU : Dec-09, Feb-10

To fulfil increasing demands for higher performance it is necessary to process data concurrently to achieve better throughput instead of processing each instruction sequentially as in a conventional computer. Processing data concurrently is known as **parallel processing**. There are two basic ways by which we can achieve parallelism. These are :

- **Multiple Functional Units** : System may have two or more ALUs so that they can execute two or more instructions at the same time.
- **Multiple Processors** : System may have two or more processors operating concurrently.

We know that, the earlier processors had only one arithmetic and logic unit (ALU) in its CPU. Furthermore, the ALU could only perform one function at a time, resulting quite slow process for executing a long sequence of arithmetic and logical instructions. Now-a-days processors are available with multiple functional units. These multiple functional units are used to execute functions of ALU which can be distributed and operate in parallel. This parallel execution increases the speed of process.

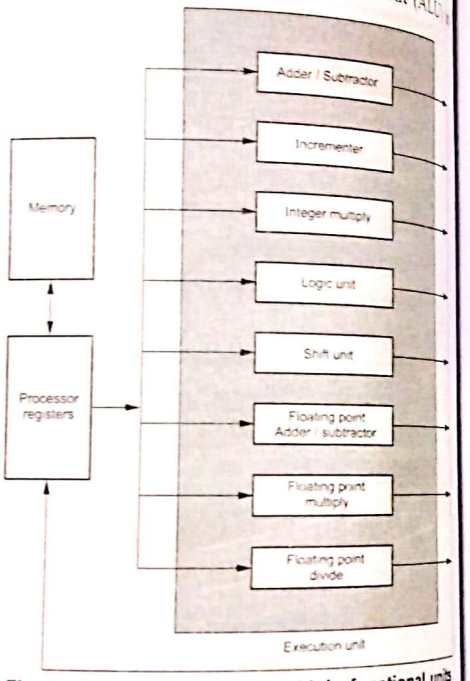


Fig. 7.1.1 Processor with multiple functional units

The Fig. 7.1.1 shows one typical way of separating the execution unit into eight functional units. These functional units can be operated in parallel to achieve faster execution time. All functional units are independent of each other, so two operands can be added while third is being shifted or incremented and so on. These functional units get operands from processor registers and the result of the operation is stored back in the processor register/s as shown in the Fig. 7.1.1.

7.1.1 Flynn's Taxonomy

Parallel processing can be classified in many ways. It can be classified according to internal organisation of processors, according to interconnection structure used between processors or according to flow of information through the system. One such classification is introduced by M. J. Flynn.

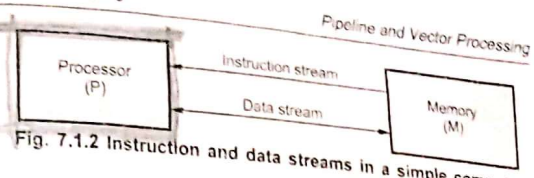


Fig. 7.1.2 Instruction and data streams in a simple computer

We know that a typical processing unit operates by fetching instructions and operands from the main memory, executing the instructions, and placing the results in the main memory. The steps associated with the processing of an instruction form an instruction cycle. The instruction is can be viewed as forming an instruction stream flowing from main memory to the processor, while the operands form another stream, the data stream, flowing to and from the processor, as shown in Fig. 7.1.2.

In 1966, Micheal J. Flynn has made an informal and widely used classification of processor parallelism based on the number of simultaneous instruction and data streams seen by the processor during program execution.

The classification made by Micheal J. Flynn divides computers into four major groups.

- Single instruction stream-single data stream (SISD).
- Single instruction stream-multiple data streams (SIMD).
- Multiple instruction streams-single data stream (MISD).
- Multiple instruction streams-multiple data streams (MIMD).

1. Single instruction stream-single data stream (SISD) : Most conventional machines with one CPU containing a single arithmetic-logic unit capable only of scalar arithmetic fall into this category. SISD computers and sequential computers are thus synonymous. In SISD computers instructions are executed sequentially but may overlap in their execution stages (Pipelining). They may have more than one functional unit, but all functional units are control by a single control unit.

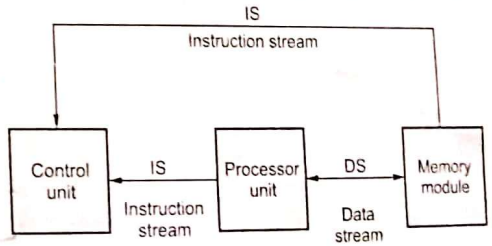


Fig. 7.1.3 SISD computer

2. Single instruction stream-multiple data stream (SIMD) : This category corresponds to array processors. They have multiple processing/execution units and one control unit. Therefore, all processing/execution units are supervised by the single control unit. Here, all processing elements received same instruction from control unit but operate on different data sets from distinct data streams.

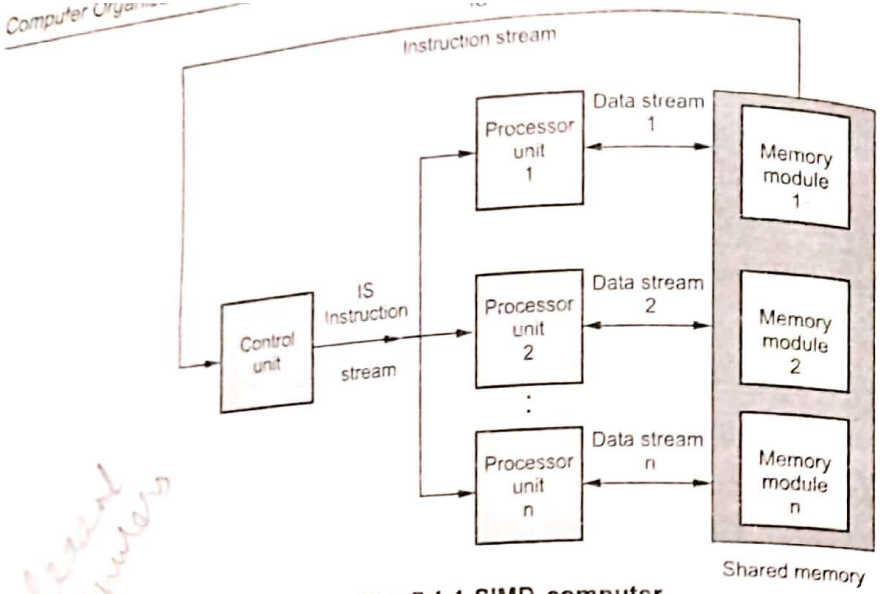
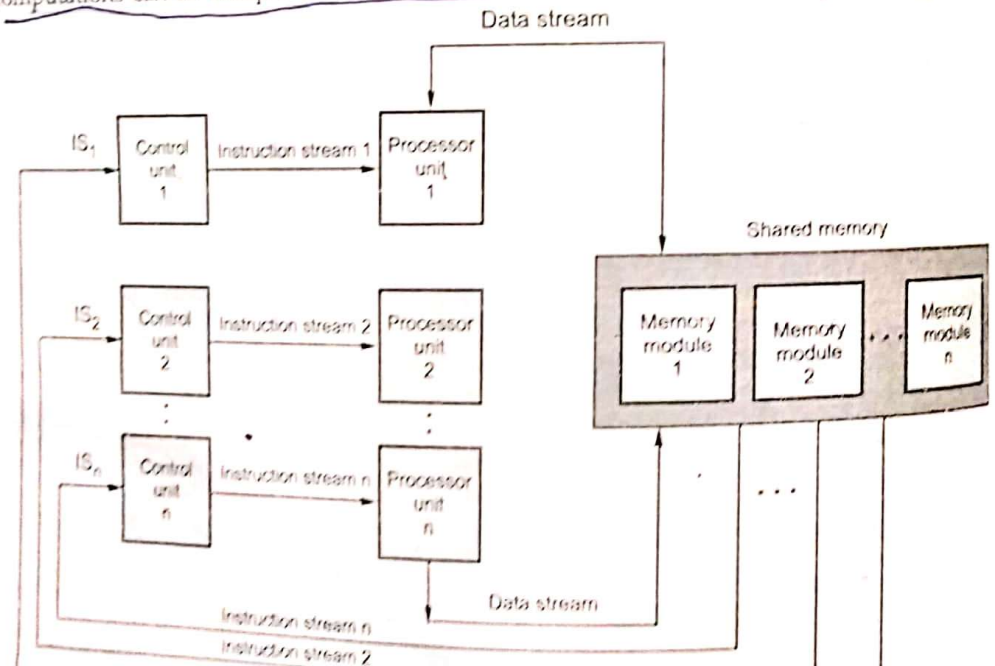


Fig. 7.1.4 SIMD computer

3. Multiple instruction stream-single data stream (MISD) : Not many parallel processors fit well into this category. In MISD, there are n processor units, each receiving distinct instructions operating over the same data stream and its derivatives. The results of one processor become the input of the next processor in the micropipe. The fault-tolerant computers where several processing units process the same data using different programs belongs to the MISD class. The results of such apparently redundant computations can be compared and used to detect and eliminate faulty results.



a hardware architecture is known as **pipeline processor**. Fig. 7.2.1 shows the basic structure of a pipeline processor.

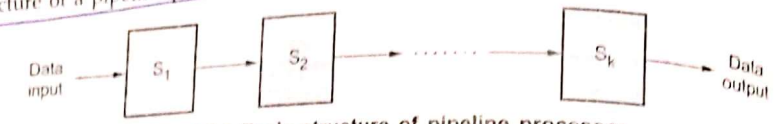


Fig. 7.2.1 Basic structure of pipeline processor

As shown in the Fig. 7.2.1, a pipeline processor is constructed with k processing stages. Data inputs such as operands are fed into the pipeline at the first stage S_1. The processed results are passed from stage S_i to stage S_{i+1}, for all i = 1, 2, 3, . . . , k - 1. The final result emerges from the pipeline at the last stage S_k. A specific control mechanism is used to pass data flow along the pipeline stages. Depending on the control mechanism used we can categorize pipelines into :

- Asynchronous
- Synchronous

Example 7.2.1 *Perform the arithmetic operation (Ai * Bi) + (Ci * Di) with a stream of number. Specify a pipeline configuration to carry out the task. List the contents of all registers in the pipeline for i = 1 through 6.*

Solution : The Fig. 7.2.2 shows the pipeline configuration to carry out the given task. It consists of seven registers that receive new data with every clock pulse, two multipliers and one adder circuit. The suboperations performed in each stage of the pipeline are :

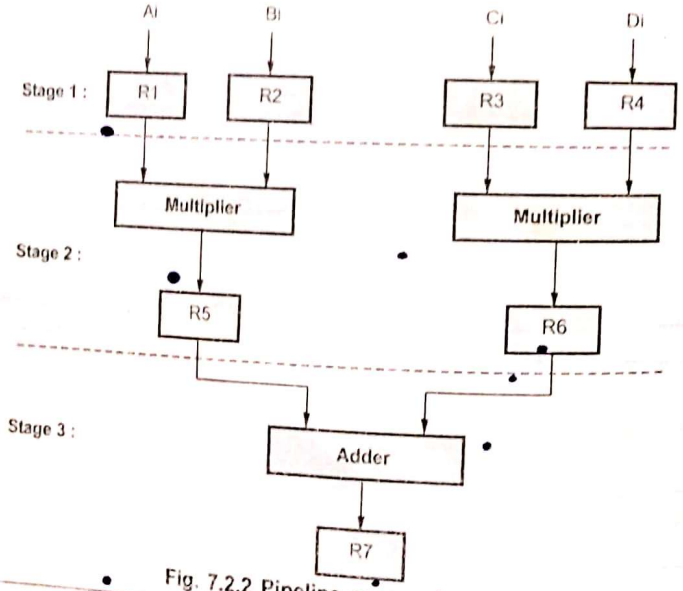


Fig. 7.2.2 Pipeline processing

Stage 1 : R1 ← Ai, R2 ← Bi, R3 ← Ci, R4 ← Di

Stage 2 : R5 ← R1 * R2, R6 ← R3 * R4

Stage 3 : R7 ← R5 + R6

The Table 7.2.1 shows contents of all registers in the pipeline for i = 1 through 6. The first clock pulse transfers A1, B1, C1 and D1 into R1, R2, R3 and R4, respectively. The second clock pulse transfers the product of R1 and R2 into R5 and product of R3 and R4 into R6. The same clock pulse transfers A2, B2, C2 and D2 into R1, R2, R3 and R4 respectively. The third clock pulse operates on all three stages simultaneously. It places A3, B3, C3 and D3 into R1, R2, R3 and R4, respectively, transfers the product of R1 and R2 into R5 and product of R3 and R4 into R6 and places sum of R5 and R6 into R7. It takes three clock pulses to fill up the pipe and retrieve the first output from R7. From there on, each clock produces a new output and moves the data one step down the pipeline. This is continued as long as new input data flow into the system. When no more input data are available, the clock must continue until the last output emerges out of the pipeline.

Clock Pulse Number (i)	Stage 1				Stage 2		Stage 3
	R1	R2	R3	R4	R5	R6	R7
1	A1	B1	C1	D1	–	–	–
2	A2	B2	C2	D2	A1 * B1	C1 * D1	–
3	A3	B3	C3	D3	A2 * B2	C2 * D2	A1 * B1 + C1 * D1
4	A4	B4	C4	D4	A3 * B3	C3 * D3	A2 * B2 + C2 * D2
5	A5	B5	C5	D5	A4 * B4	C4 * D4	A3 * B3 + C3 * D3
6	A6	B6	C6	D6	A5 * B5	C5 * D5	A4 * B4 + C4 * D4
7	–	–	–	–	A6 * B6	C6 * D6	A5 * B5 + C5 * D5
8	–	–	–	–	–	–	A6 * B6 + C6 * D6

Table 7.2.1 Contents of all registers in the pipeline for i = 1 through 6

7.2.1 Asynchronous Pipeline Model

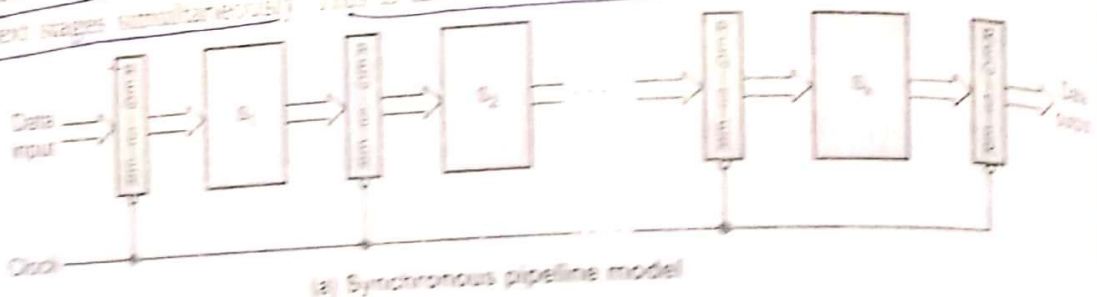
In asynchronous pipeline models, data flow along the pipeline stages is controlled by a handshaking protocol, as shown in the Fig. 7.2.3. When stage S_i is ready to transmit its result, it sends a ready signal to stage S_{i+1}. The result of S_i is an input data for stage S_{i+1}, thus stage S_{i+1} accepts result of S_i as incoming data and returns an acknowledge signal to S_i.



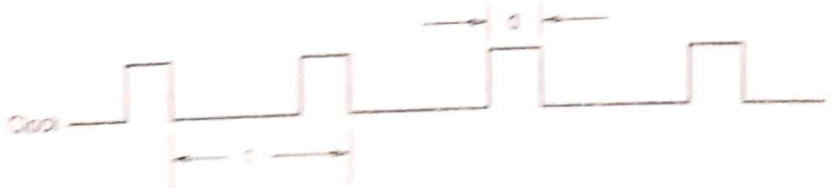
Fig. 7.2.3 Asynchronous pipeline model

7.2.1.2 Synchronous Pipeline Model

In synchronous pipeline model, clocked high speed registers are used to interface between stages. At the falling edge of the clock pulse, all registers transfer data to the next stages simultaneously. This is illustrated in Fig. 7.2.4.



(a) Synchronous pipeline model



(b) Clock signal

Fig. 7.2.4

7.2.2 Performance Parameters of Pipelining

Each pipeline stage is a combinational logic circuit. It requires a specific amount of time in terms of propagation delay to process the input data. The processing time of each stage is known as **stage delay**. It is denoted as t. The delays are also introduced due to interstage transfer data. These time delays are known as **interstage delay** and is denoted as d.

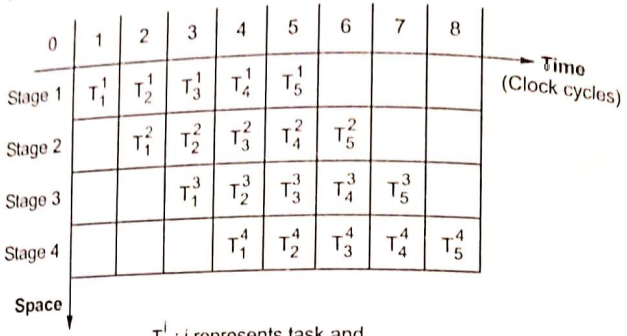
Clock Cycle : To determine the time period for the clock cycle it is necessary to consider stage delay and interstage delay. The interstage delays can be same across the stages. However, stage delays may not be same for different stages. In such case, maximum stage delay, denoted as t_m is considered to determine the time period of the clock cycle. The time period for clock cycle, τ can be given as

$$\tau = \max_i \{t_i\} + d = t_m + d$$

The space-time diagram shown in Fig. 7.2.5 (b) has four stages and five tasks. Therefore, the ideal total time required is

$$T_k = [4 + (5 - 1)]\ T_p = 8 \text{ clock cycles (Refer Fig. 7.2.5 (b))}$$

The amount of time required to perform same number of tasks in a non-pipeline processor is nT_n:



T_j^i : j represents task and
i represents sub task

Fig. 7.2.5 (b) The space-time diagram showing the overlapped operations

Speedup Factor : The speedup factor of a k-stage pipeline over an equivalent non-pipelined processor is defined as

$$S_k = \frac{n\,T_n}{[k + (n-1)]\,T_p}$$

It can be noticed that maximum speedup is $S_k \rightarrow k$ as $n \rightarrow \infty$. However, this maximum speed is very difficult to achieve because of data dependencies between successive task (instructions), program branches, interrupts and other factors.

$$\underset{n\to\infty}{S_k} = \frac{nT_n}{nT_p} = \frac{T_n}{T_p}$$

If we assume that $T_n = kT_p$ we have

$$S_k = \frac{kT_p}{T_p} = k$$

Example 7.2.2 *A non-pipelined system takes 100 ns to process a task. The same task can be processed in a six-segment pipeline with a clock cycle of 20 ns. Determine the speedup ratio of the pipeline for 200 tasks. What is the maximum speedup that can be achieved ?*

Solution : Given : $T_n = 100$ ns, $T_p = 20$ ns, $n = 200$, $k = 6$

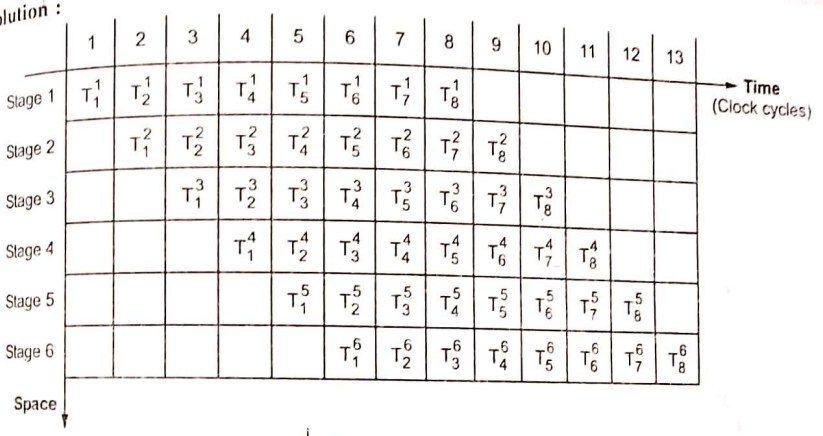
$$\text{Speedup ratio } S_k = \frac{nT_n}{[k+(n-1)]\,t} = \frac{200 \times 100 \times 10^{-9}}{[6+(200-1)] \times 20 \times 10^{-9}} = 4.878$$

$$\text{Maximum speedup} = \underset{n\to\infty}{\text{limit}} \frac{nT_n}{[k+(n-1)]\,T_p} = \frac{nT_n}{n\,T_p}$$

$$= \frac{T_n}{T_p} = \frac{100 \text{ ns}}{20 \text{ ns}} = 5$$

Example 7.2.3 *Draw a space time diagram for a six segment stage pipeline showing the time it takes to process 8 tasks.*

Solution :



T_j^i = j represents task and
i represents sub task

Fig. 7.2.6 The space time diagram

Efficiency : The efficiency of a linear pipeline is defined as a ratio of speedup factor and the number of stages in the pipeline. The efficiency of k-stage pipeline is given as

$$E_k = \frac{S_k}{k} = \frac{nk}{k+(n-1)} \div k = \frac{n}{k+(n-1)}$$

For example, if speedup factor is 2.5 and number of stages are four, the efficiency can be given as

$$E_k = \frac{2.5}{4} = 0.625$$

It can be noticed that the efficiency approaches to unity when $n \rightarrow \infty$. When $n = 1$, efficiency is minimum. It is $1/k$.

Throughput : The pipeline throughput H_k is defined as the number of results (tasks) that can be completed by a pipeline per unit time. It is given as

$$H_k = \frac{n}{[k+(n-1)]\,T_p} = \frac{E_k}{T_p} \qquad \because E_k = \frac{n}{[k+(n-1)]}$$

$$= E_k f \qquad \because f = 1/T_p$$

In the ideal case, $H_k = 1/T_p = f$ when $E_k \to 1$. This means that the maximum throughput of a linear pipeline is equal to its frequency, which corresponds to one output result per clock period. The overall throughput of pipeline is always less than f. This is because usually $E_k < 1$.

Performance/Cost Ratio (PCR) : Let us consider t be the total time required for a non-pipelined sequential program of a given function. To execute the same program on a k-stage pipeline with an equal flow-through delay t, one needs a clock period of $T_p = t/k + d$, where d is the latch delay. Therefore maximum throughput for k-stage pipeline can be given as

$$f = \frac{1}{T_p} = \frac{1}{(t/k+d)}$$

The total pipeline cost is roughly estimated by c + kh, where c represents cost of all logic stages and h represents the cost of each latch. With this information, in 1973, Larson has defined a **pipeline performance/cost ratio** as

$$PCR = \frac{f}{c+kh} = \frac{1}{(t/k+d)(c+kh)}$$

Optimal number of stages : We know that performance of the pipeline can be increased by increasing pipeline stages. Many times pipeline stages are subdivided upto logic gates level called **micropipelining** to increase parallelism. However, increase in pipeline stages increases the cost of pipelining. Therefore, the optimal choice of the number of pipeline stages should be able to maximize a performance/cost ratio. Fig. 7.2.7 shows the graph of PCR Vs k-stages. The peak of curve corresponds to an optimal choice for the number of desired pipeline stages. It is given as

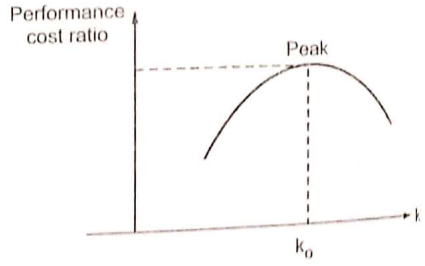


Fig. 7.2.7 Optimal number of pipeline stages

$$k_o = \sqrt{\frac{t \cdot c}{d \cdot h}}$$

where t is the total flow-through delay of the pipeline, c is the total stage cost, d is the latch delay and h is the latch cost.

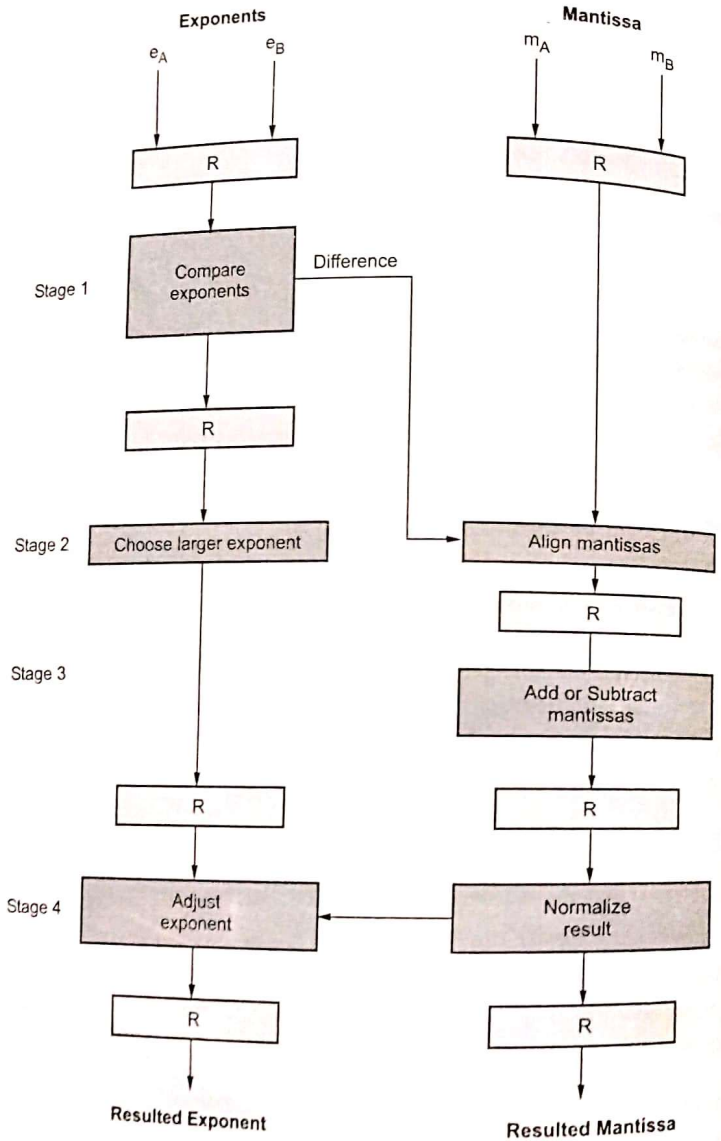


Fig. 7.3.1 Pipeline for floating-point addition and subtraction

These four sub operations are divided into four different stages. In the first stage exponent comparison is carried out by subtracting them. In the second stage the larger exponent is chosen as the exponent of the result. The result of the comparison, i.e. difference in the first stage determines how many times the mantissa associated with the smaller exponent must be shifted right to align the mantissas. This subtask is also performed in second stage. In the third stage, two mantissas are added or subtracted. Finally, the result is normalized in the stage four. If an overflow occurs, the mantissa of the result is shifted right and the exponent is incremented by one. If an underflow occurs, the mantissa is shifted left and exponent is decremented by a number equal to the number of leading zeros in the mantissa. The flow chart in the Fig. 7.3.1 illustrates above discussed pipelined stages. Registers shown in the flowchart stores the result of the suboperations. (See Fig. 7.3.1 on previous page.)

Example 7.3.1 *Let us consider that we have to add number A and number B.*

where $A = m_A \times 2^{e_A} = 0.10110 \times 2^3$ *and*

$B = m_B \times 2^{e_B} = 0.11010 \times 2^4$

Now we see how suboperations are performed in different stages.

Stage 1 : Exponent comparison

$|e_A - e_B| = |3 - 4| = 1$

Stage 2 : Align the mantissas and choose larger exponent

1. Exponent of the result is $= e_B = 4$.
2. Mantissa $m_A = 0.01011$

$A = 0.01011 \times 2^4$ and

$B = 0.11010 \times 2^4$

Stage 3 : Add mantissas

Mantissa of result $m_R = 0.01011 + 0.11010 = 1.00101$

$\therefore$ $R = 1.10101 \times 2^4$

Stage 4 : Normalize the result

$R = 0.100101 \times 2^5$

Mantissa is shifted right by one digit so that it has a fraction with a nonzero first digit, and exponent is incremented by one.

We know that, the comparator, shifter, incrementer, decrementer and add / subtractor used in implementation of floating point addition / subtraction pipeline are the combinational circuits. They introduce time delays to produce output. Suppose that

delay d = 10 ns. The time period for clock is equal to interstage

$$t = \max_i \{t_i\}_1^k + d = 90 \text{ ns} + 10 \text{ ns} = 100 \text{ ns}$$

The amount of time required to perform floating point addition / subtraction using a non-pipeline floating point adder / subtractor will be

$$T_1 = t_1 + t_2 + t_3 + t_4 + d = 50 + 80 + 90 + 80 + 10 = 310 \text{ ns}$$

Therefore, the speed up factor for this particular pipeline is

$$S_4 = \frac{310 \text{ ns}}{100 \text{ ns}} = 3.1$$

Example 7.3.2 *The time delay of the four segment (fetch instruction, decode and calculate effective address, fetch operand, and execute instruction segment) are as follows :*

$t_1 = 40$ *ns,* $t_2 = 25$ *ns,* $t_3 = 85$ *ns,* $t_4 = 55$ *ns*

The interface registers delay time $t_r = 5$ *ns*

i) How long would it take to add 75 pairs of numbers in the pipeline ?

ii) How can we reduce the total time to about one half of the time calculated in part (i) ?

Solution : i) Minimum clock cycle for pipeline

= Maximum delay through a stage + Delay time for interface register

= 85 + 5 = **90 ns**

Number of cycles required to add 75 pairs of numbers

= k + n − 1 = 4 + 75 − 1 = **78**

∴ Time required to execute 75 pairs of numbers

= 78 × 90 = **7020 ns**

ii) One-half time of part (i) = $\frac{7020}{2}$ = **3510 ns**

∴ Minimum clock cycle of pipeline = $\frac{3510}{78}$ = **45 ns**

∴ Maximum delay through stage = 45 − 5 = **40 ns**

Thus keeping maximum delay through stage equal to 40 ns we can reduce the total time to 3510 ns.

In this instruction pipelining four instructions are in progress at any given time. This means that four distinct hardware units are needed, as shown in Fig. 7.4.2. These units are implemented such that they are capable of performing their tasks simultaneously and without interfering with one another. Information from the stage is passed to the next stage with the help of buffers.

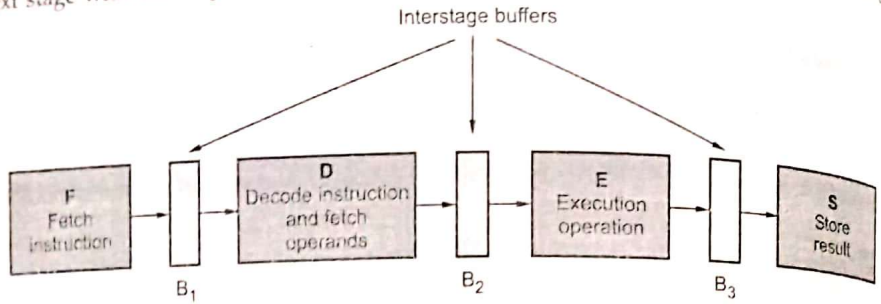


Fig. 7.4.2 Hardware organization for four-stage segment instruction pipeline

The Fig. 7.4.3 shows the four stage CPU pipeline. As shown in the Fig. 7.4.3, the pipeline works properly when there is no branch or interrupt. In case of branch or interrupt it is necessary to update PC with the target address and flush the pipeline. This is indicated by the shaded block.(See Fig. 7.4.3 on next page)

7.4.1 Pipeline Conflicts

The timing diagram for instruction pipeline operation shown in Fig. 7.4.1 completes the processing of one instruction in each clock cycle. This means that the rate of instruction processing is four times that of sequential operation. The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages. However, this increase would be achieved only if pipelined operation shown in Fig. 7.4.1 could be performed without any interruption throughout program execution. Unfortunately, this is not the case.

For many of reasons, one of the pipeline stages may not be able to complete its operation in the allotted time. For example, the stage E (Execute) is responsible for arithmetic and logic operations. In the timing diagram one clock period is allotted for this task. This stage can perform most of the operations in the allotted time. However, operations like divide may take more than one clock cycle and hence require more time to complete execution. The Fig. 7.4.4 shows an example in which the operation specified in instruction 2 requires three cycles to complete, from cycle 4 through cycle 6. Thus, in cycles 5 and 6, the information in buffer B2 must remain intact until the instruction execution stage has completed its operation. This means that stage 2 and in turn, stage 1 are blocked from accepting new instructions because the information in B1 cannot be

overwritten. Thus decode step for instruction and fetch step for instruction 5 must be postponed as shown in the Fig. 7.4.4.

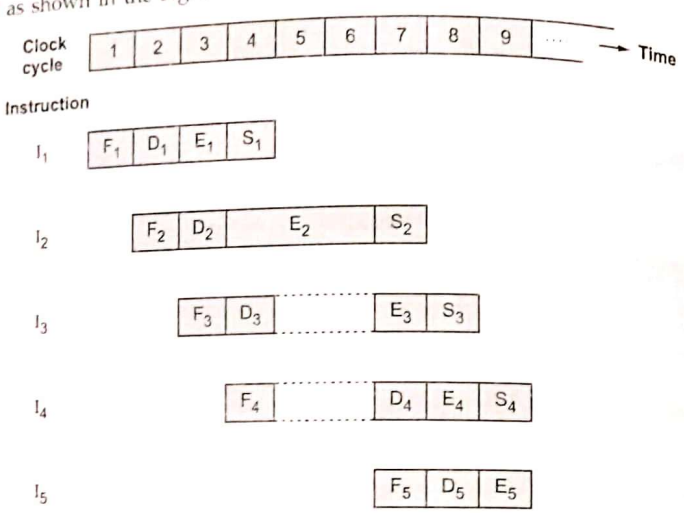


Fig. 7.4.4 Effect on pipeline of an execution operation taking more than one clock cycle

The instruction pipeline shown in Fig. 7.4.4 is said to have been **stalled** for two clock cycles (clock cycles 5 and 6) and normal pipeline operation resumes in clock cycle 7. Any reason that causes the pipeline to stall is called a **hazard** or conflict.

Pipeline conflicts are caused by resources usage conflicts among various instructions in the pipeline. Such conflicts are triggered by inter instruction dependencies. Such sequential instructions overlap their fetch, decode and execution through a pipeline processor, inter instruction dependencies may arise to prevent the sequential data flow in the pipeline. Until the completion of the previous instruction, the current instruction can not be initiated into the pipeline. Another type of conflicts are due to a job scheduling problems. In all there are three types of conflicts.

1. Resource conflicts : These conflicts are because of conflicts due to insufficient resources when even with all possible combination, it may not be possible to overlap the operation.

2. Data or data dependency conflicts : These result when instruction in the pipeline depends on the result of previous instructions which are still in pipeline and not completed.

3. Branch difficulties : They arise while pipelining branch and other instructions that change the contents of program counter. The simplest way to handle these conflicts is to

stall the pipeline. Stalling of the pipeline allows few instructions to proceed to completion while stopping the execution of those which results in conflicts.

Resource conflict

The performance of pipelined processor depends on whether the functional units are pipelined and whether they are multiple execution units to allow all possible combination of instructions in the pipeline. If for some combination, pipeline has to be stalled to avoid the resource conflicts then there is a **structural hazard**. In other words, we can say that when two instructions require the use of a given hardware resource at the same time, the structural hazard occurs. The most common case in which this hazard may arise is in access to memory. One instruction may need to access memory for storage of the result while another instruction or operand needed is being fetched. If instructions and data reside in the same cache unit, only one instruction can proceed and the other instruction is delayed. To avoid such type of conflicts many processors use separate caches for instruction and data.

Data dependency conflict

When either the source or the destination operands of an instruction are not available at the time expected in the pipeline and as a result pipeline is stalled, we say such a situation is a data hazards or **data conflict.**

Consider a program with two instructions, I_1 followed by I_2. When this program is executed in a pipeline, the execution of these two instructions can be performed concurrently. In such case the result of I_1 may not be available for the execution of I_2. If the result of I_2 is dependent on the result of I_1 we may get incorrect result if both are executed concurrently. For example, assume A = 10 in the following two operations :

$$I_1 : A \leftarrow A + 5$$

$$I_2 : B \leftarrow A \times 2$$

When these two operations are performed in the given order, one after the other, we get result = 30. But if they are performed concurrently, the value of A used in computing B would be the original value, 10, leading to an incorrect result. In this case data used in the I_2 depend on the result of I_1. The conflict due to such situation is called **data conflict** or **data dependent conflict.** To avoid incorrect results we have to execute dependent instructions one after the other (in-order). In other words we can say that, the read and write a shared variables by different instructions in a pipeline may lead to different results causing data-dependent conflicts, if these instructions are executed out of order.

7.4.2 Techniques to Avoid Data Conflicts

1. The easiest way to handle data conflicts is to stall the pipeline. Pipeline hardware **interlocks** detect the conflict and stalls the pipeline beginning with the instruction that requires the data till it is made available by earlier instruction. This delay or stall called a **bubble**.

2. The second simple hardware technique which can handle data conflict is called **operand forwarding or register by passing**. In this technique, ALU result is fed back as ALU input instead of transferring it into a destination register. When the forwarding logic detects the previous ALU operation has the operand for current instruction, it forwards ALU output instead of register file. This technique requires additional hardware paths through multiplexers as well as forwarding logic detect circuit.

3. In some cases register forwarding technique also fails leading to pipeline stall.

Example :	LW R1, 60(R5)	IF	ID	EX	MEM	WB		
	ADD R4, R1, R8	IF	ID	EX	MEM	WB		
	OR R7, R3, R8		IF	ID	EX	MEM	WB	
	XOR R5, R2, R7			IF	ID	EX	MEM	WB

Here LW instruction does not have result ready till the memory access is over. While ADD requires it during execution (EX) stage. The simplest way to handle such situation is to stall ADD instruction for one clock. But it will degrade the performance.

Rather than allowing the pipeline to stall, the compiler can rearrange above instructions to avoid a read following a load. It is used in RISC processors. It is called **pipeline scheduling**.

After scheduling :	LW	R1, 60(R5)
	OR	R7, R3, R8
	ADD	R4, R1, R8
	XOR	R5, R2, R7

A load which requires that following instruction does not use its result is said to be **delayed load** and the pipeline slot after load instruction is called **delayed slot**. Some machines like MIPS 2000 rely heavily on software scheduling to avoid data conflict. In these machines there is no support in hardware to detect and stall the pipeline. However, the compiler can insert number of NOP instructions to fill the delayed slot.

Branch Difficulties / Conflicts

These are caused by change in the relative execution timing of the overlapped instructions due to pipelining. Branch conflict can cause greater degradation in

performance than data conflicts. These occurs due to instruction like branch or jump which change flow of instruction which changes content of PC.

Branch instruction (i)	IF	ID	EX	MEM	WB				
		IF	Stall	Stall	IF	ID	EX	MEM	WB
i + 1			Stall	Stall	Stall	IF	ID	EX	MEM
i + 2				Stall	Stall	Stall	IF	ID	EX
i + 3				Stall	Stall	Stall	IF	ID	
i + 4						Stall	Stall	Stall	IF
i + 5									

Fig. 7.4.5

If branch is taken as shown in Fig. 7.4.5 then the PC value is not known till the end of MEM. Next three instructions are fetch even though they are not required. Hence they have to be flushed after branch is taken and new set of instructions have to be fetched from the branch address. In simple pipelines there is no problem of flushing but in complex pipelines, flushing is not simple. One way to handle it is to stall the pipeline after instruction decoded is branch.

For machine with lower CPI even this loss of two cycles due to branching is significant. To tackle it, prediction of whether branch is taken or not early in pipeline has to be done.

Modified pipeline for branch instruction

IF	MAR ← PC	
	IR ← MEM [MAR]	
	PC ← PC + 4	
ID	A ← RS_1, B ← RS_2	Target calculation
	BTA ← PC + Branch offset	done in ID stage
	If cond THEN PC ← BTA	so branch penalty only 1
EX	–	
MEM	–	Not used
WB	–	

Branch Penalty

The penalty due to branches depends upon two factors :

- Instruction set : In RISC penalty is less than complex instruction set computers.
- Depth of pipeline : more deep pipelining more is branch penalty.

The branch penalties can be reduced by proper scheduling through compiler techniques. The basic idea behind these techniques is to fill the 'delay slot' with some useful instruction which in most cases will be executed.

Delayed Branching

There are three ways to fill the delay slot :

1. The delay slot is filled with an independent instruction before branch. In this case performance always improves.
2. The delay slot is filled from the target branch instructions. Performance improves if only branch is taken.
3. The delay slot is filled with an instruction which is one of the fall through instruction. In this case performance improves if branch is not taken.

These above techniques are called **delayed branching**.

Conditional Branching

The conditional branching is a major factor that affects the performance of instruction pipelining. There are several approaches to deal with conditional branching.

These are :

- Multiple Streams
- Prefetch Branch Target
- Loop Buffer
- Branch Prediction

Multiple Streams

We know that, a simple pipeline suffers a penalty for a branch instruction. To avoid this, in this approach they use two streams to store the fetched instruction. One stream stores instructions after the conditional branch instruction and it is used when branch condition is not valid. The other stream stores the instruction from the branch address and it is used when branch condition is valid.

Drawbacks :

- Due to multiple streams there are contention delays for access to registers and to memory.
- Each branch instruction needs additional stream.

Prefetch Branch Target

In this approach, when a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch. This already fetched target is used when branch is valid.

Loop Buffer

A loop buffer is a small very high speed memory. It is used to store recently prefetched instructions in sequence. If conditional branch is valid, the hardware first checks whether the branch target is within the buffer. If so, the next instructions are fetched from the buffer, instead of memory avoiding memory access.

Advantages

- In case of conditional branch, instructions are fetched from the buffer saving memory access time. This is useful for **loop** sequences.
- If a branch occurs to a target just a few locations ahead of the address of the branch instruction, we may find target in the buffer. This is useful for IF-THEN-ELSE-ENDIF sequences.

Branch Prediction

Prediction techniques can be used to check whether a branch will be valid or not valid. The common prediction techniques are :

- Predict Never Taken
- Predict Always Taken
- Predict By Opcode
- Taken/Not Taken Switch
- Branch History Table

The first three approaches are simple. In the first two approaches if prediction is wrong a page fault or protection violation error occurs. The processor then halts prefetching and fetches the instruction from the desired address.

In the third prediction technique, the prediction decision is based on the opcode of the branch instruction. The processor assumes that the branch will be taken from certain branch opcodes and not for others.

The fourth and fifth prediction techniques are dynamic; they depend on the execution history of the previously executed conditional branch instruction.

In Taken / Not taken switch system one or two tag bits are associated with each conditional branch instruction. These bits reflect the recent history of the conditional branch instruction. These bits reflect the recent history of the conditional branch instruction. These are referred to as a taken / not taken switch, that directs the processor to make a particular decision the next time the instruction is encountered. For example, in a single bit tag, 1 is recorded if the last execution of this instruction resulted valid branch and 0 is recorded for no branch. When this instruction is fetched again, fetching unit checks the tag bit. If tag bit is 0 it fetches next instructions in the sequence ; otherwise it fetches instructions from the branch address.

This technique is very useful for loop sequences. In loop sequences N-1 times branching is taken. Therefore, the prediction is wrong twice for each loop : Once on entering the loop, and once on exiting.

The use of history bits just explained, has one drawback : If the decision is made to take the branch, the target instruction cannot be fetched until the branch instruction is decoded. Greater performance could be achieved if the instruction fetch could be initiated as soon as the branch decision is made. For this purpose, it is necessary to store branch address along with the tag bits. In the branch history table, technique, the branch address along with the tag bits is stored in the branch target buffer or branch history table.

Review Questions

1. *What is instruction pipeline ?*
2. *Draw and explain the flowchart of four stage CPU pipeline.*
3. *What do you mean by stalled pipeline ?*
4. *What are conflicts ?*
5. *Explain resource conflicts in the instruction pipelining.*
6. *Explain data conflicts in the instruction pipelining.*
7. *What are branch difficulties in the instruction pipelining ?*
8. *How data conflicts can be prevented in the instruction pipelining ?*
9. *How does conditional branch instruction affect the performance of instruction pipelining ? Explain with the help of timing diagram.*
10. *Explain instruction pipelining with examples.* **GTU : Dec.-03,05,10,12, May-14 Marks 8**
11. *What are the pipeline conflicts ? Explain the hardware techniques to handle the branch instructions.* **GTU : Dec.-09, May-12, Marks 7**
12. *Considering three segment instruction pipeline, illustrate the concepts of delayed load and delayed branch with example.* **GTU : Dec.-09, May-14, Marks 7**
13. *Explain the following terms.*
    1) Delayed load 2) Pipeline conflict. **GTU : Feb.-10, Marks 3**
14. *What is instruction pipeline ? Explain the four-segment instruction pipeline with the suitable diagram.* **GTU : Dec.-11, Marks 7**

7.5 RISC Pipeline

In RISC processor most instructions are register to register. These instructions have the following two phases :

- Instruction Fetch (I)
- Execute (E)

The instruction fetch phase fetches the instruction to be executed from memory and then execute phase performs an ALU operation with register input and output to execute the instruction.

In case of load and store instructions, three phases are required :

- Instruction fetch (I)
- Execute (E)
- Data transfer (D)

Here, also the instruction fetch phase fetches the instruction to be executed. In execution phase address of memory is calculated and in data transfer phase actual data is transferred from register-to-memory or from memory-to-register depending upon the instruction.

The Fig. 7.5.1 shows that how these phases can be overlapped in RISC processors to achieve the pipelining. The Fig. 7.5.1 (a) shows the two-way pipelining and the Fig. 7.5.1 (b) shows the three way pipelining.

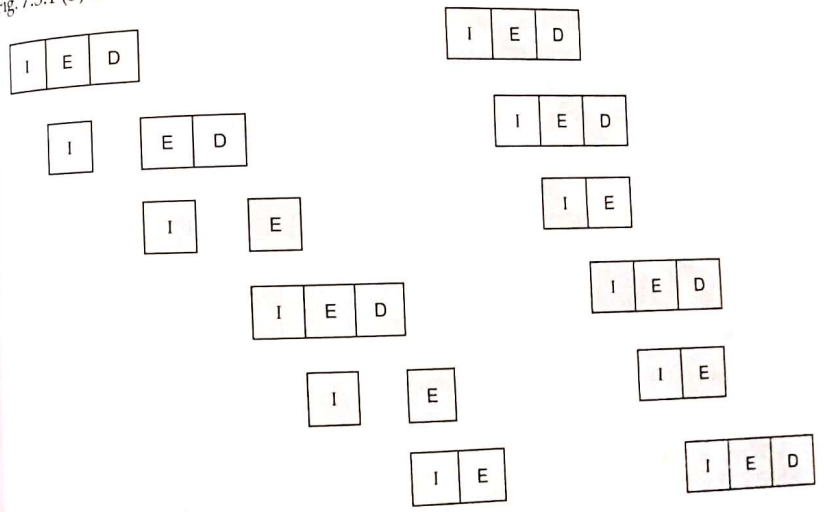


Fig. 7.5.1

In two way instruction pipelining, I and E phases of two different instructions are performed simultaneously. In three way instruction pipelining, three instructions can be overlapped. Therefore, maximum execution rate in two way instruction pipelining is twice the normal execution rate and it is three times the normal execution rate in case of three way instruction pipelining.

It is important to note that, in RISC processors, pipeline conflicts such as data conflicts and branch penalties are handled by the compiler instead of designing hardware. The compiler uses delay load and delayed branch techniques (discussed in previous section) to handle conflict problems.

Review Question

1. *Write a short note on RISC pipeline.*

7.6 Vector Processing

GTU : May-11,13

A **vector processor, or array processor,** is a CPU design where the instruction set includes operations that can perform mathematical operations on multiple data elements simultaneously. This is in contrast to scalar processor which handles one element at a time using multiple instructions. The vast majority of CPUs are scalar (or close to it). Vector processors were common in the scientific computing area, where they formed the basis of most supercomputers.

The need to increase computational power is a never ending requirement. In scientific and research areas, the computations involved are quite extensive and hence high power computers are a must.

The areas like structural engineering, petroleum exploration, aerodynamics, hydrodynamics, nuclear research, tomograph, VLSI design, AI can have data in the form of matrices which suits vector processors to process it at a high speed.

Some examples of it are :

1. In radar and signal processing for detection of space/underwater targets.
2. In remote sensing for earth resources exploration.
3. In computational wind tunnel experiments.
4. In 3D stop action computer assisted tomography.
5. Weather forecasting.
6. Medical diagnosis.

7.6.1 Characteristics of Vector Processing

1. A vector is an ordered set of elements. A vector operand contains an ordered set of n elements, where n is called the length of the vector. Each element in a vector is a scalar quantity, which may be floating point number, an integer, a logical value, or a character (byte).
2. In vector processing, two successive pairs of elements are processed each clock period. The dual vector pipes and the dual sets of vector functional units allow

two pairs of elements to be processed during the same clock period. As each pair of operations is completed, the results are delivered to appropriate elements of the result register. The operation continues until the number of elements processed is equal to the count specified by the vector length register.

For example :

$$C(1:50) = A(1:50) + B(1:50)$$

This vector instruction includes the initial addresses of the two source operands, one destination operand, the length of the vectors and the operation to be performed.

3. Vector instructions are classified into four basic types :

$f_1 : V \rightarrow V$ $\quad$ $f_2 : V \rightarrow S$

$f_3 : V \times V \rightarrow V$ $\quad$ $f_4 : V \times S \rightarrow V$

where V indicates vector operand and S indicates scalar operand. The operations f_1 and f_2 are unary operations such as vector square root, vector sine, vector complement, vector summation and so on. On the other hand, operations f_3 and f_4 are binary operations such as vector add, vector multiply, vector-scalar add and so on.

4. In vector processing, identical processes are repeatedly invoked many times, each of which can be subdivided into subprocesses.
5. In vector processing, successive operands are fed through the pipeline segments and require as few buffers and local controls as possible. This parallel vector processing allow the generation of more than two results per clock period. The parallel vector operations are automatically initiated either when successive vector instructions use different functional units and different vector registers, or when successive vector instructions use the result stream from one vector register as the operand of another operation using a different functional units. This process is known as **chaining**.
6. Because of the startup delay in a pipeline, a vector processors perform better with longer vectors.
7. Vector processing is usually faster and more efficient than scalar processing because it reduces the overhead associated with maintenance of the loop-control variables.

7.6.2 Vector Processing Approach

In order to reduce the amount of time required to access data from memory, most modern CPUs use a technique known as instruction pipelining in which the instructions pass through several sub-units in turn. The first sub-unit reads the address and decodes it, the next "fetches" the values at those addresses, and the next does the math itself.

With pipelining the "trick" is to start decoding the next instruction even before the first has left the CPU, in the fashion of an assembly line so the address decoder is constantly in use. Any particular instruction takes the same amount of time of complete, a time known as the latency, but the CPU can process an entire batch of operations much faster than if it did so one at a time.

Vector processors take this concept one step further. Instead of pipelining just the instructions, they also pipeline the data itself. They are fed instructions that say not just to add A to B, but to add all of the numbers "in the array A to all of the numbers from array B". Instead of constantly having to decode instructions and then fetch the data needed to complete them, it reads a single instruction from memory, and "knows" that the next address will be one larger than the last. This allows for significant savings in decoding time.

To illustrate what a difference this can make, consider the simple task of adding two groups of 10 numbers together. In a normal programming language you would write a "loop" that picked up each of the pairs of numbers in turn, and then added them. To the CPU, this would look something like this :

```
Execute this loop 10 times
read the next instruction and decode it
fetch first number
fetch second number
add them
put the result here
end loop
```

But to vector processor, this task looks considerably different :

```
read instruction and decode it
fetch 10 numbers
fetch 10 numbers
add them
put the results here
```

There are several savings inherent in this approach. For one, only two address translations are needed. Depending on the architecture, this can represent a significant savings by itself. Another savings is fetching and decoding the instruction itself, which only has to be done one time instead of ten. The code itself is also smaller, which can lead to more efficient memory use.

But more than that, the vector processor typically has some form of superscalar implementation, meaning there is not one part of the CPU adding up those 10 numbers, but perhaps two or four of them. Since the output of a vector command does not rely on the input from any other, those two (for instance) parts can each add five of the

numbers, thereby completing the whole operation in half the time. This is parallel processing approach for vector computation.

In the above discussion we have seen the vector computation approach. Let us see the processor organization that can be used to implement vector computation approach. The processor organizations used for vector computation can be categorized as :

- Pipelined ALU
- Parallel ALU
- Parallel Processors

Pipelined ALU

The Fig. 7.6.1 (a) shows the pipelined ALU organization. The complex ALU operations such as floating point operations are decomposed into stages. These stages can be pipelined using pipelined ALU. A pipelined ALU will save time only if it is fed a stream of data from sequential locations. A single, isolated floating point operation is not speeded up by a pipeline. The speedup is achieved when a vector of operands is presented to the ALU.

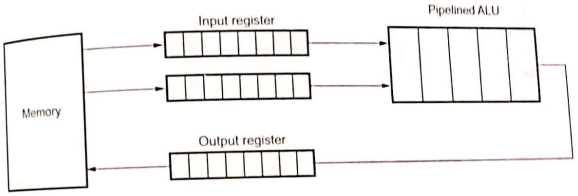


Fig. 7.6.1 (a) pipelined ALU

Parallel ALUs

The Fig. 7.6.1 (b) shows the parallel ALU organization. Here, multiple ALUs in a single processor are used to implement vector processing approach. In this case, the control unit of the processor routes data to ALUs so that they can function in parallel. It is also possible to use pipelining on each of the parallel ALUs to further speedup the operation.

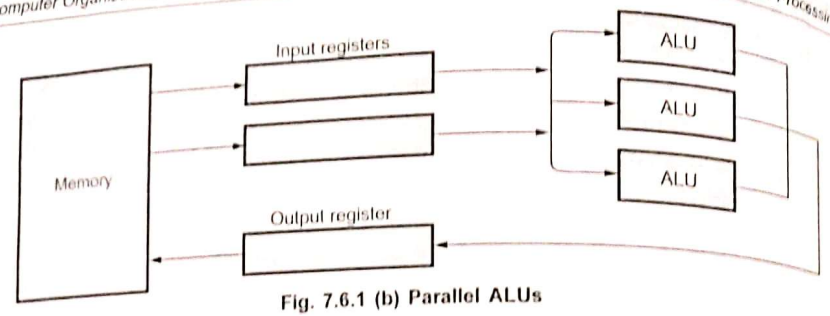


Fig. 7.6.1 (b) Parallel ALUs

Parallel Processors

In the parallel processor organization, the multiple ALUs from multiple processors operate in parallel. The ALUs sometimes called **processing units (PE)** are synchronized to perform the same function at the same time. Such processors are also known as **array processors.** In short, an array processor is a synchronous parallel computer with multiple ALUs. The array processor can handle single instruction and multiple data streams. Because of this reason, array processors are also known as SIMD processors.

Example 7.6.1 *Define vector and vector instruction format. Also explain matrix multiplication with the help of inner product calculation on a pipeline processor.*

Solution : A vector is an ordered set of one-dimensional array of data items. It is represented as a row vector by V = (V1 V2 V3 ... Vn] where V is a vector and n is a length of a vector. It may be represented as a column vector if the data items are listed in a column. A computer capable of vector processing allows operations to be specified with a single vector instruction as follows

C(1 : 100) = A(1 : 100) + B(1 : 100)

This vector instruction adds two vectors A and B of length 100 to produce a vector C.

The Fig. 7.6.2 shows the instruction format of vector instruction. It is a three-address instruction. It includes operation code, the base addresses of the source operands, the base address of the destination operand, and the length of the vectors all in one composite instruction.

Operation code	Base address source 1	Base address source 2	Base address destination	Vector length

Fig. 7.6.2 Format of vector instruction

Inner product is one of the most popular operations performed in computers with vector processors. The inner product consists of the sum of k product terms of the form

$$C = A_1 B_1 + A_2 B_2 + A_3 B_3 + \ldots + A_k B_k$$

The Fig. 7.6.2 (a) shows the inner product calculation on a pipeline processor. Here, the values of A and B vectors are either in memory or in processor registers. A four-segment floating point multiplier and four-segment floating point adder pipelines are used. A_i and B_i pairs are brought in and multiplied at a rate of one pair per cycle. Assuming that k = 100, after the first four cycles, the products begin to be added to the output of the adder (initially cleared). The first addition will take place in the ninth cycle is $A_1 B_1 + A_5 B_5$. The tenth cycle starts the addition $A_2B_2 + A_6B_6$ and so on. This pattern breaks on the summation into four sections as follows :

$$C = A_1B_1 + A_5 B_5 + A_9B_9 + \ldots + A_{97}B_{97}$$
$$A_2B_2 + A_6B_6 + A_{10}B_{10} + \ldots + A_{98}B_{98}$$
$$A_3B_3 + A_7B_7 + A_{11}B_{11} + \ldots + A_{99}B_{99}$$
$$A_4B_4 + A_8B_8 + A_{12}B_{12} + \ldots + A_{100}B_{100}$$

The adder pipeline will then have one partial product in each of its four segments. The four partial sums must then be added in an additional step to form the final sum.

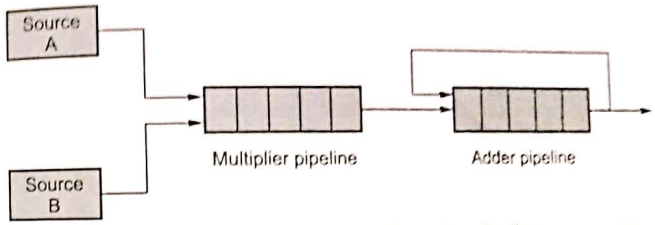


Fig. 7.6.2 (a) Inner product calculation on a pipeline processor

Matrix multiplications is one of the most computational intensive operations within which the inner product is used. Let us consider the 3 × 3 matrices A and B.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$$

The product matrix C is a 3 × 3 matrix whose elements can be determined by the inner product as,

$$c_{ij} = \sum_{k=1}^{3} a_{ik} \times b_{kj}$$

For example, c_{11} can be calculated as,

$$c_{11} = a_{11} b_{11} + a_{12}b_{21} + a_{13}b_{31}$$

Each element is determined by three multiplications and three additions (if c_{ij} is initialized to 0). Thus the total number of multiplications or additions required to

compute the matrix product is $9 \times 3 = 27$, i.e. n^3 where two matrices are $n \times n$. The multiplication of two $n \times n$ matrices requires 9 (n^2) inner products.

7.6.3 Memory Interleaving

Pipeline and vector processors often require simultaneous access to memory from two or more sources. An instruction pipeline may require the fetching of opcode and operand at the same time from two different memory segments. Similarly, arithmetic pipeline may require two or more operands to enter the pipeline at the same time. In such systems, the memory access time is the bottleneck and one way to reduce it is to use cache memory. Alternative technique to reduce memory access time is **memory interleaving**. In this technique, the main memory is divided into a number of memory modules and the addresses are arranged such that the successive words in the address space are placed in different modules. Refer Fig. 7.6.3. Most of the times CPU accesses consecutive memory locations. In such situations, addresses will be to the different modules. Since these modules can be accessed in parallel, the average access time of fetching word from the main memory can be reduced.

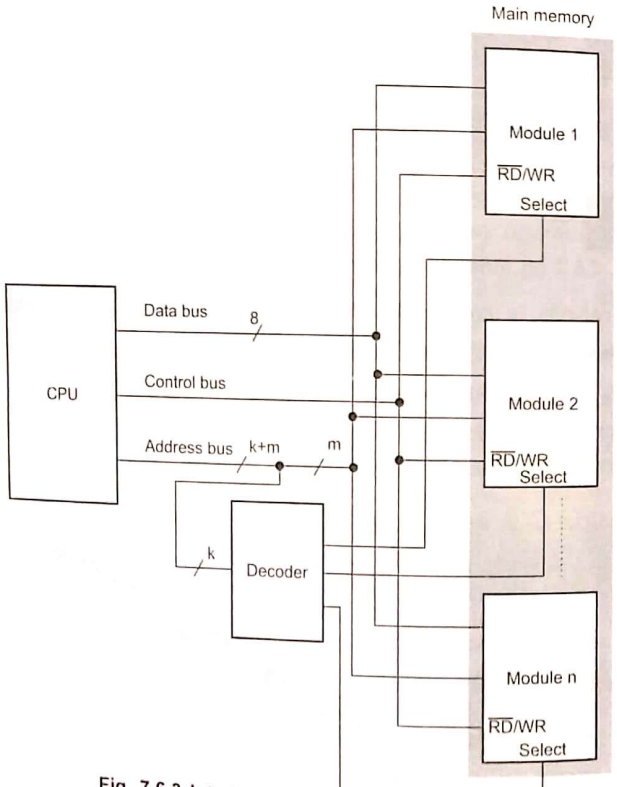


Fig. 7.6.3 Interleaved memory organization

register is 64 words, each word is 64 bits). It allows us to multiply or add(subtract) one vector register by another vector register giving a third vector register.

The Cray 1 consists of –

- 64-bit words
- 8 MB of RAM
- 16-way interleaving on low-order bits
- 50 ns memory cycle
- 12.5 ns clock cycle (80 MHz) to achieve the speed of 160 million floating point operations per second (160 megaflops)
- 12 pipelined functional units
- The Cray 1 has 3 basic data types : addresses (24-bit integer), integers (64-bit), floating point (64-bit, 48-bit mantissa).
- The Cray 1 has 12 functional units. These are divided into four groups :

Group 1 - Vector units

- Vector (integer) Add : 3 stages
- Vector Logical : 2 stages
- Vector Shift : 4 stages

Group 2 - Vector and scalar units

- Floating Add: 6 stages
- Floating Multiply: 7 stages
- Floating Reciprocal Approximation: 14 stages

Group 3 - Scalar units

- Integer Add : 3 stages
- Logical : 1 stage
- Shift : 2 stages
- Scalar population count and leading zero count : 3 stages

Group 4 - Address units

- Add : 2 stages
- Multiply : 6 stages

- Cray 1 instructions are 32 or 16 bits, so from 2 to 4 instructions can be packed into a word. Instructions are thus addressed on 16-bit boundaries while data is addressed on 64-bit boundaries.
- It has 8 megabyte (1 million word) main memory. The memory is divided into 16 banks, with each bank having a 50 ns access time. Therefore, when 16 memory

banks are accessed simultaneously, the memory data transfer rate is 320 million words per second.

Another group within the same company developed the first multiprocessor supercomputer, the Cray X-MP, which was introduced in 1982. Its cycle time is 9.5 ns. it has mainly added a multiport memory system and increased memory to 16 M words, 32-way interleave. In 1988, Cray Research introduced the Cray Y-MP, the world's first supercomputer to sustain over 1 gigaflop on many applications. Its cycle time is reduced to 6 ns. The Cray-2 system appeared in 1985, providing a tenfold increase in performance over the Cray-1. Its effective cycle time is 8.2 ns. It can access up to 256 M words of memory 64 or 128 way interleave depending on configuration.

7.6.4.2 Practical Considerations in Supercomputer Design

To achieve such high speeds, high-power (i.e. hot) drivers are employed, signals are detected with specialized analog circuits, conductors are all shielded and precisely tuned in both impedance and length, and data is encoded with error-correcting so that losses can be recovered.

In addition, the circuits are usually designed to operate in balanced mode so that there is no change in power drawn as drivers switch. As one driver switches from low to high, another switches from high to low, so that the power supply sees a DC load and there is no coupling of switching noise back into the logic via the power supply. In addition, using balanced signal lines can increase the signal to noise ratio by 6 dB, although these are not often used. In a design such as the Cray-1, roughly 40% of the transistors supposedly do nothing but balance the power loading.

Even so, these machines dissipate large amounts of heat. For the dissipation of heat they use special thermal conduction modules in which a multichip substrate is mounted in a carrier with built-in plumbing for a chilled water jacket.

Review Questions

1. *Explain the need of vector processing*
2. *Explain the characteristics of vector processing.*
3. *Explain the general instruction format of the vector processor.*
4. *Write a short note on memory interleaving.* **GTU : May-13, Marks 4**
5. *Write a short note on supercomputers.*
6. *Explain the practical considerations in supercomputer design.*
7. *Explain (i) Vector processing (ii) Vector operations. Explain how matrix multiplication is carried out on a computer supporting vector computations.* **GTU : May-11, Marks 7**

7.7 Array Processors

GTU : Dec.-13, 2

An **array processor** is synchronous parallel computer with multiple arithmetic logic units. These arithmetic logic units are also known as **processing elements** (PE), which can operate in parallel in a lock step fashion. By replication of ALUs the array processors can achieve the spatial parallelism. As PEs are synchronized to perform the same function at the same time, the array processor can handle single instruction and multiple data streams. Because of this reason, array processors are also known as SIMD processors.

7.7.1 SIMD Array Processor

The Fig. 7.7.1 shows the typical functional structure of an SIMD array processor. As shown in the Fig. 7.7.1, all PEs are synchronized and controlled by single control unit. Each PE is an ALU with attached working registers and local memory. The PEs are interconnected by a data-routing network. The control unit (CU) consists of control processor (CP) and control memory (CM). The system and user programs are executed

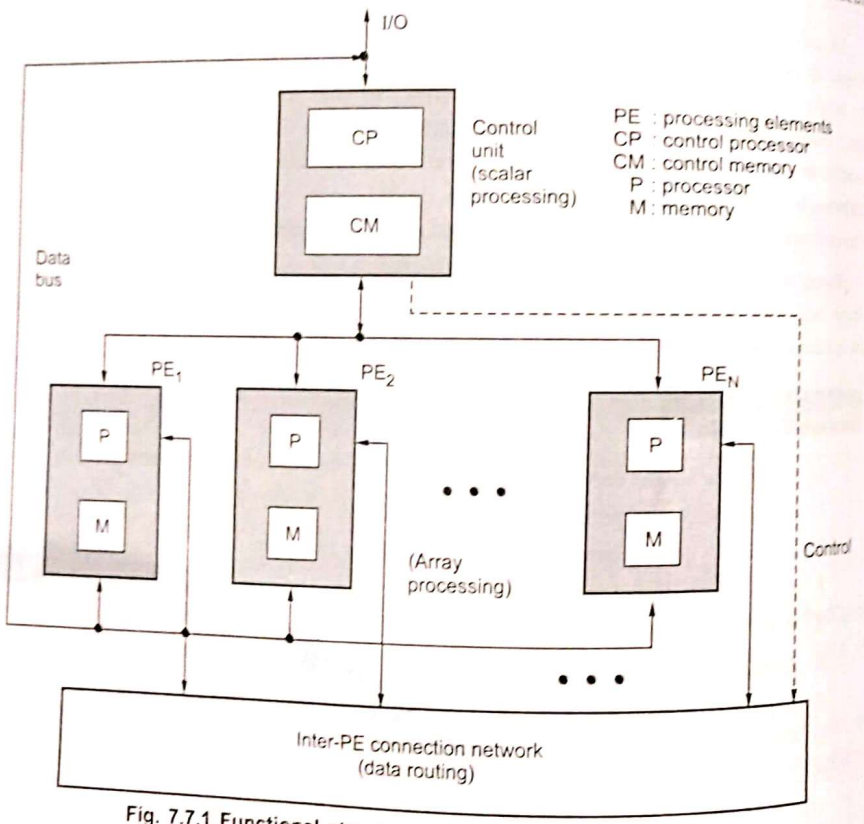


Fig. 7.7.1 Functional structure of an SIMD array processor

under control of CU. The user programs are loaded into CU memory from an external source.

The function of CU is to decode all the instructions and determine where the decoded instructions should be executed.

- **Scalar instructions** - executed directly inside the CU
- **Vector instructions** - they are broadcast to PEs for through duplicate arithmetic units (PEs)

All the PEs perform the same function synchronously in a lock step fashion under the command of CU. Vector operands are distributed to PE memories before parallel execution in the array of PEs. The distributed data can be loaded into PE memories from an external source via system data bus or via CU in a broadcast mode using control bus. Masking schemes are used to control the status of each PE during execution of vector instruction. A masking vector is used to control the status of all PEs. In other words not all PEs need to participate in the execution of vector instruction.

7.7.2 Attached Array Processor

To enhance the performance of conventional computer and to provide vector processing capability for complex scientific applications, the array processor can be attached to a conventional computer as a peripheral. Due to such attachment the performance of conventional computer improves a lot because of parallel processing support given by array processor with multiple functional units.

The Fig. 7.7.2 shows the interface between the attached array processor and the conventional computer. The array processor is connected to the conventional computer through an input output controller. Usually, a local memory is available for array processor. The data for array processor is transferred to local memory from the main memory through high speed bus.

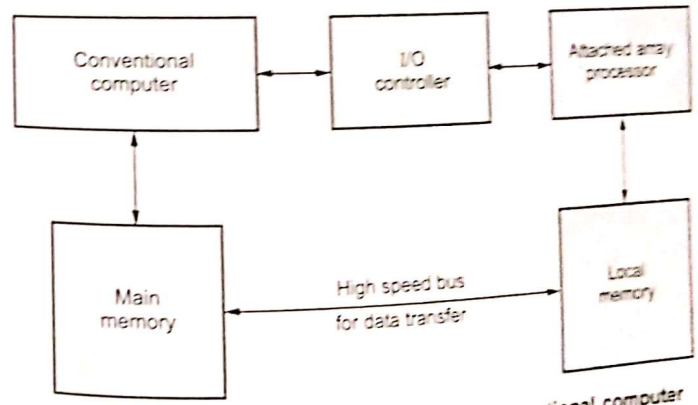


Fig. 7.7.2 Attached array processor with conventional computer

Review Questions

1. What is an array processor ?
2. Draw and explain the typical functional structure of a SIMD array processor.
3. Write a short note on attached array processor.
4. What is an array processor ? Explain the different types of array processor.

GTU : Dec.-11, Marks 7

5. What is array processors ? Explain attached array processors and SIMD array processors.

GTU : Dec.-12, Marks 7

❑❑❑

8 Computer Arithmetic

Syllabus

Introduction, Addition and subtraction, Multiplication algorithms (booth multiplication algorithm), Division algorithms, Floating point arithmetic operations, Decimal arithmetic unit

Contents

8.1 Introduction
8.2 Addition and Subtraction of Signed Numbers. . **Nov.-13,** ..... Marks 13
8.3 Multiplication Algorithms
8.4 Signed Operand Multiplication
8.5 Fast Multiplication
8.6 Division Algorithms
8.7 Floating Point Arithmetic Operations
8.8 Decimal Arithmetic Unit ..... **May-12,** ..... Marks 7
8.9 Decimal Arithmetic Operations

8.1 Introduction

- Based on the number system two basic data types are implemented in the computer system : **fixed point numbers** and **floating point numbers**. Representing numbers in such data types is commonly known as fixed point representation and floating point representation, respectively.
- In binary number system, a number can be represented as an integer or a fraction.
- Depending on the design, the hardware can interpret number as an integer or fraction.
- The radix point is never explicitly specified. It is implicited in the design and the hardware interprets it accordingly.
- In integer numbers, radix point is fixed and assumed to be to the right of the right most digit.
- As radix point is fixed, the number system is referred to as **fixed point number system**. With fixed point number system we can represent positive or negative integer numbers.
- Floating point number system allows the representation of numbers having both integer part and fractional part.

Review Question

1. *What are fixed point and floating point numbers ?*

8.2 Addition and Subtraction of Signed Numbers

GTU : Nov.-18

- We can relate addition and subtraction operations of numbers by the following relationship :

  $(\pm A) - (+B) = (\pm A) + (-B)$ and

  $(\pm A) - (-B) = (\pm A) + (+B)$
- Therefore, we can change subtraction operation to an addition operation by changing the sign of the subtrahend.

8.2.1 1's Complement Representation

The 1's complement of a binary number is the number that results when we change all 1's to zeros and the zeros to ones.

Example 8.2.1 *Find 1's complement of $(11010100)_2$.*

Solution :

	1	1	0	1	0	1	0	0	Number
	▽	▽	▽	▽	▽	▽	▽	▽	NOT operation
	0	0	1	0	1	0	1	1	1's complement of number

8.2.2 2's Complement Representation

The 2's complement is the binary number that results when we add 1 to the 1's complement. It is given as

2's complement = 1's complement + 1

The 2's complement form is used to represent negative numbers.

Example 8.2.2 *Find 2's complement of $(11000100)_2$.*

Solution :

	1	1	0	0	0	1	0	0	Number
						1	1		Carry
	0	0	1	1	1	0	1	1	1's complement of number
+								1	Add 1
	0	0	1	1	1	1	0	0	2's complement of number

8.2.3 Subtraction of Binary Numbers using 2's Complement Method

In a 2's complement subtraction, negative number is represented in the 2's complement form and actual addition is performed to get the desired result. For example, operation A – B is performed using following steps :

1. Take 2's complement of B.
2. Result ← A + 2's complement of B.
3. If carry is generated then the result is positive and in the true form. In this case, carry is ignored.
4. If carry is not generated then the result is negative and in the 2's complement form.

Example 8.2.3 *Perform $(28)_{10} - (15)_{10}$ using 6-bit 2's complement representation.*

Solution : $(28)_{10} = (011100)_2$

$(15)_{10} = (001111)_2$

	0	0	1	1	1	1

$(15)_{10}$

Carry

	1	1	0	0	0	0
+						1
	1	1	0	0	0	1

1's complement of $(15)_{10}$

Add 1

2's complement of 15, i.e., $(-15)_{10}$

		1	1				
		0	1	1	1	0	0
+		1	1	0	0	0	1
	X	0	0	1	1	0	1

Sign Extension →

Ignore Carry →

Carry

Binary equivalent of $(28)_{10}$

2's complement of 15, i.e. $(-15)_{10}$

Result : Binary equivalent of $(13)_{10}$

$(28)_{10}$ + $(-15)_{10}$ = $(13)_{10}$

Example 8.2.4 *Perform $(15)_{10} - (28)_{10}$ using 6-bit 2's complement representation.*

Solution : $(15)_{10} = (001111)_2$

$(28)_{10} = (011100)_2$

	0	1	1	1	0	0

Binary equivalent of $(28)_{10}$

		1	1			
	1	0	0	0	1	1
+						1
	1	0	0	1	0	0

Carry

1's complement of $(28)_{10}$

Add 1

2's complement of $(28)_{10}$, i.e., $(-28)_{10}$

		1	1			
	0	0	1	1	1	1
+	1	0	0	1	0	0
	1	1	0	0	1	1

Carry

Binary equivalent of $(15)_{10}$

2's complement of $(28)_{10}$

Result = Binary equivalent of $(-13)_{10}$

$(15)_{10}$ + $(-28)_{10}$ = $(-13)_{10}$

Verification

		0	0	1	1	0	0
+							1
	0	0	1	1	0	1

1's complement of result

Add 1

Result = Binary equivalent of $(13)_{10}$

Example 8.2.5 *Perform (–35) + (–40) in binary with negative numbers in 2's complement format.*

GTU : Nov.-15, Marks 3

Solution : $(35)_{10} = (0100011)_2$

$(40)_{10} = (0101000)_2$

	0	1	0	0	0	1	1
	1	0	1	1	1	0	0
+							1
	1	0	1	1	1	0	1

$(35)_{10}$

1's complement of $(35)_{10}$

Add 1

2's complement of $(35)_{10} = (-35)_{10}$

	0	1	0	1	0	0	0
	1	0	1	0	1	1	1
+							1
	1	0	1	1	0	0	0

$(40)_{10}$

1's complement of $(40)_{10}$

Add 1

2's complement of $(40)_{10} = (-40)_{10}$

1		1	1				
	1	0	1	1	1	0	1
+	1	0	1	1	0	0	0
1	0	1	1	0	1	0	1

Carry

$(-35)_{10}$

$(-40)_{10}$

Result = $(-75)_{10}$

Fig. 8.2.1

8.2.4 Adders

- Digital computers perform various arithmetic operations.
- The most basic operation, no doubt, is the addition of two binary digits. This simple addition consists of four possible elementary operations, namely,

  $0 + 0 = 0$

  $0 + 1 = 1$

  $1 + 0 = 1$

  $1 + 1 = 10_2$

- The first three operations produce a sum whose length is one digit, but when the last operation is performed sum is two digits.
- The higher significant bit of this result is called a **carry,** and lower significant bit is called **sum**.
- The logic circuit which performs this operation is called a **half-adder**.
- The circuit which performs addition of three bits (two significant bits and a previous carry) is a **full-adder**.

8.2.4.1 Half Adders

- The half-adder operation needs two binary inputs : augend and addend bits; and two binary outputs : sum and carry.
- The truth table shown in Table 8.2.1 gives the relation between input and output variables for half-adder operation.

Inputs		Outputs	
A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Table 8.2.1 Truth table for half-adder

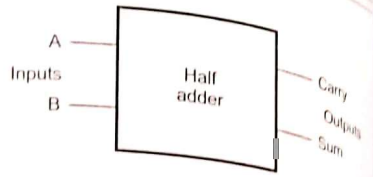
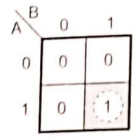


Fig. 8.2.2 Block schematic of half-adder

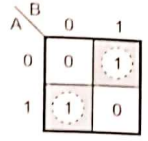
K-map simplification for carry and sum

For carry



Carry = AB

For sum



Sum = $A\overline{B} + \overline{A}B$
= $A \oplus B$

Fig. 8.2.3 Maps for half-adder

Logic diagram

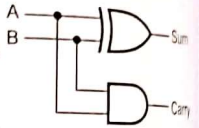


Fig. 8.2.4 Logic diagram for half-adder

Limitations of Half-Adder :

- In multidigit addition we have to add two bits along with the carry of previous digit addition.
- Effectively such addition requires addition of three bits. This is not possible with half-adder. Hence half-adders are not used in practice.

Example 8.2.6 *Draw half adder using NAND gates.*

Solution : For half adder :

$$\text{Sum} = A\overline{B} + \overline{A}B \qquad C_{out} = AB$$

$$= \overline{\overline{A}\overline{B} + \overline{A}B} \qquad = \overline{\overline{A}\overline{B}}$$

$$= \overline{\overline{A}\overline{B}} = \overline{A\overline{B}} \cdot \overline{\overline{A}B}$$

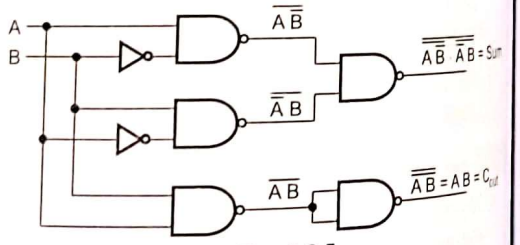


Fig. 8.2.5

8.2.2 Full Adders

- A full-adder is a combinational circuit that forms the arithmetic sum of three input bits.
- It consists of three inputs and two outputs.

- Two of the input variables, denoted by A and B, represent the two significant bits to be added.
- The third input C_{in}, represents the carry from the previous lower significant position.
- The truth table for full-adder is shown in Table 8.2.2.

Inputs			Outputs	
A	B	C_{in}	Carry	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Table 8.2.2 Truth table for full-adder

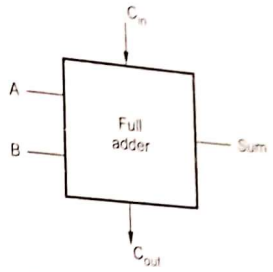
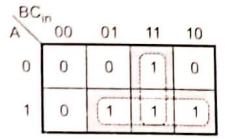


Fig. 8.2.6 Block schematic of full-adder

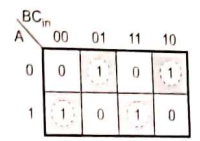
K-map simplification for carry and sum

For carry (C_{out})



$C_{out} = AB + A\,C_{in} + B\,C_{in}$

For sum



Sum = $\overline{A}\,\overline{B}C_{in} + \overline{A}B\overline{C}_{in} + A\overline{B}\,\overline{C}_{in} + ABC_{in}$

Fig. 8.2.7 Maps for full-adder

Logic diagram

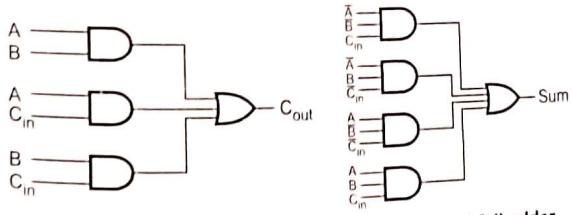


Fig. 8.2.8 Sum of product implementation of full-adder

- The Boolean function for sum can be further simplified as follows :

$$\text{Sum} = \overline{A}\,\overline{B}\,C_{in} + \overline{A}\,B\,\overline{C}_{in} + A\,\overline{B}\,\overline{C}_{in} + A\,B\,C_{in}$$

$$= C_{in}(A\bar{B} + \bar{A}B) + \bar{C}_{in}(\bar{A}B + A\bar{B}) = C_{in}(A \odot B) + \bar{C}_{in}(A \oplus B)$$

$$= C_{in}(\overline{A \oplus B}) + \bar{C}_{in}(A \oplus B) = C_{in} \oplus (A \oplus B)$$

- With this simplified Boolean function circuit for full-adder can be implemented as shown in the Fig. 8.2.9.
- A full-adder can also be implemented with two half-adders and one OR gate, as shown in the Fig. 8.2.10.
- The sum output from the second half-adder is the exclusive-OR of C_{in} and the output of the first half-adder, giving

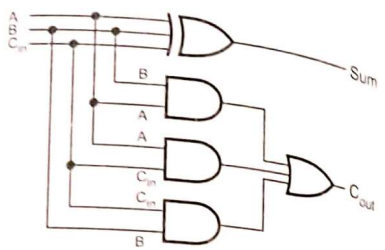


Fig. 8.2.9 Implementation of full-adder

$$C_{out} = AB + AC_{in} + BC_{in} = AB + AC_{in}(B + \bar{B}) + BC_{in}(A + \bar{A})$$

$$= AB + ABC_{in} + A\bar{B}C_{in} + ABC_{in} + \bar{A}BC_{in}$$

$$= AB(1 + C_{in} + C_{in}) + A\bar{B}C_{in} + \bar{A}BC_{in}$$

$$= AB + A\bar{B}C_{in} + \bar{A}BC_{in} = AB + C_{in}(A\bar{B} + \bar{A}B) = AB + C_{in}(A \oplus B)$$

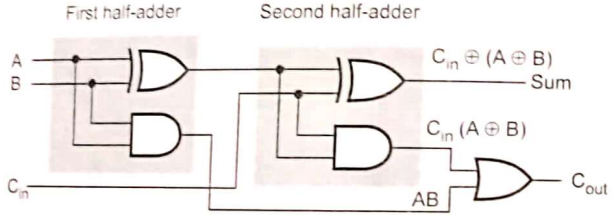


Fig. 8.2.10 Implementation of a full-adder with two half-adders and an OR gate

8.2.5 Parallel Adder

- A single full-adder is capable of adding two one-bit numbers and an input carry. In order to add binary numbers with more than one bit, additional full adders must be employed.
- A n-bit, parallel adder can be constructed using number of full adder circuits connected in parallel.

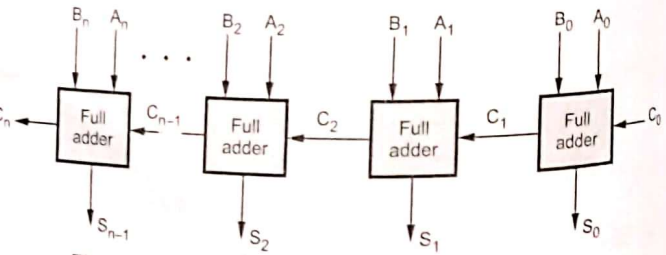


Fig. 8.2.11 Block diagram of n-bit parallel adder

- Fig. 8.2.11 shows the block diagram of n-bit parallel adder using n number of full-adder circuits connected in cascade, i.e. the carry output of each adder is connected to the carry input of the next higher-order adder.
- It should be noted that either a half-adder can be used for the least significant position or the carry input of a full-adder is made 0 because there is no carry into the least significant bit position.

8.2.6 Parallel 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. The 2's complement can be obtained by taking the 1's complement and adding one to the least significant pair of bits.
- The 1's complement can be implemented with inverters and a one can be added to the sum through the input carry to get 2's complement, as shown in the Fig. 8.2.12.

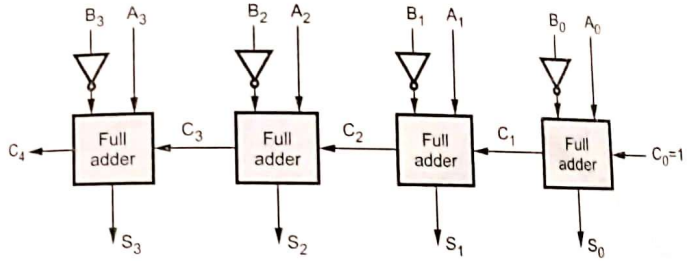


Fig. 8.2.12 4-bit parallel subtractor

8.2.7 Addition / Subtraction Logic Unit

- Fig. 8.2.13 shows hardware to implement integer addition and subtraction. It consists of n-bit adder, 2's complement circuit, overflow detector logic circuit and AVF (overflow flag).
- Number a and number b are the two inputs for n-bit adder.
- For subtraction, the subtrahend (number from B register) is converted into its 2's complement form by making Add/Subtract control signal to the logic one.
- When Add/Subtract control signal is one, all bits of number b are complemented and carry zero (C_0) is set to one.
- Therefore n-bit adder gives result as $R = a + \bar{b} + 1$, where $\bar{b}+1$ represents 2's complement of number b.

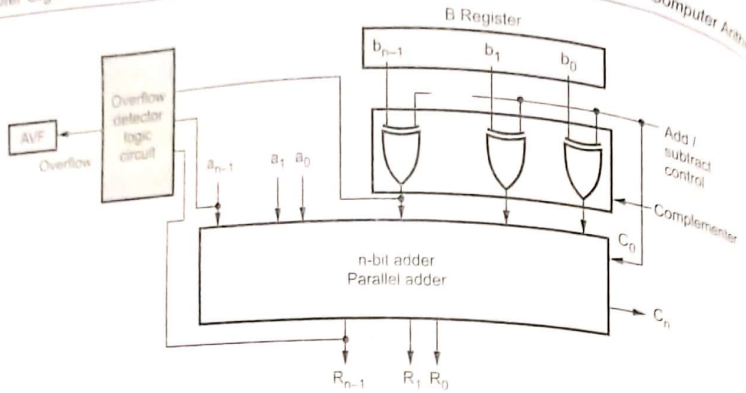


Fig. 8.2.13 Hardware for integer addition and subtraction

8.2.8 Overflow in Integer Arithmetic

- When adding signed numbers, a carry bit beyond the end of the word does not serve as the overflow indicator.
- If we add the numbers + 7 and + 3 in a 4-bit adder, the output is 1010, which is the code of – 6, a wrong result. In this case, carry bit from the MSB position is 0.

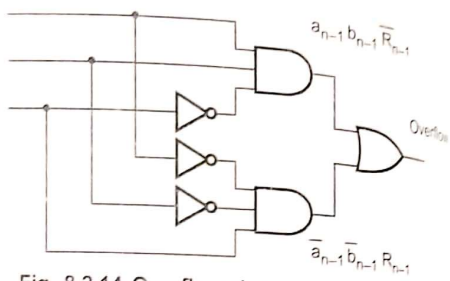


Fig. 8.2.14 Overflow detector logic circuit

- Similarly, if we add – 5 and – 6, we get output = + 5, another error. In this case carry bit from the MSB position is 1.
- One thing we can surely say that, the addition of numbers with different signs cannot cause overflow, because the absolute value of the sum is always smaller than the absolute value of one of the two operands.
- From above discussion we can conclude following points :
  1. Overflow can occur only when adding two numbers that have the same sign.
  2. The carry bit from the MSB position is not a sufficient indicator of overflow when adding signed numbers.
  3. When both operands a and b have the same sign, an overflow occurs when the sign of result does not agree with the signs of a and b. The logical expression to detect overflow can be given as

$$\text{Overflow} = a_{n-1}\, b_{n-1}\, \overline{R_{n-1}} + \overline{a_{n-1}}\, \overline{b_{n-1}}\, R_{n-1}$$

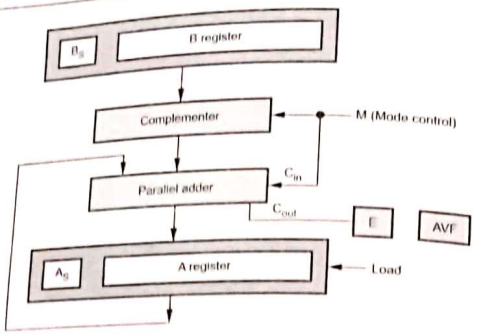


Fig. 8.2.17 (a) Hardware for signed-magnitude addition and subtraction

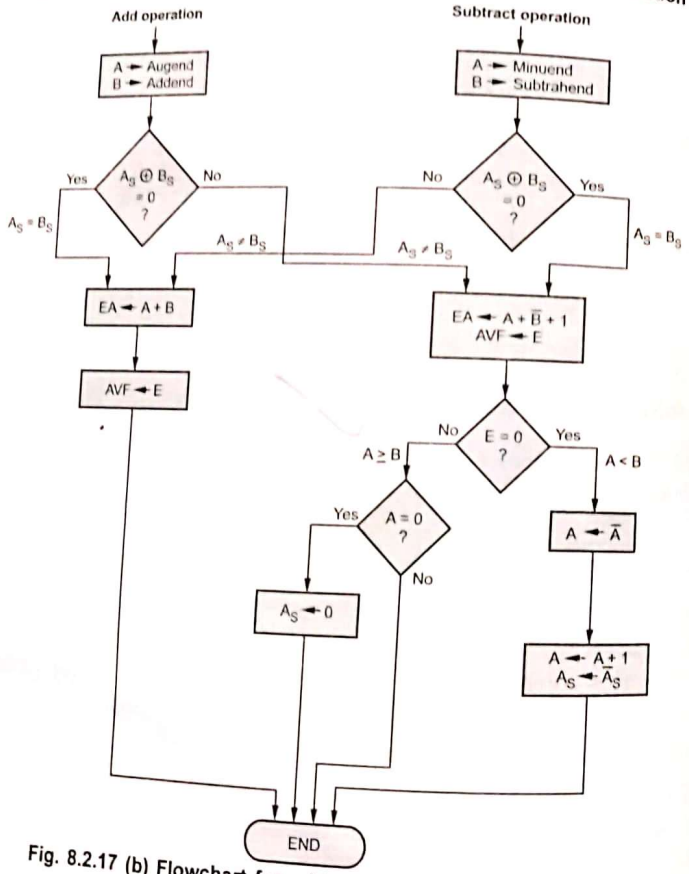


Fig. 8.2.17 (b) Flowchart for add and subtract operations

- A 1 in E indicates that A ≥ B and the number in A is the true result. If result is zero, the sign A_s is made positive to avoid a negative zero.
- A 0 in E indicates that A < B and the result is in A is not true. Here, it is necessary to take 2's complement of the result. The 2's complement of result can be performed using micro-operation $A \leftarrow \overline{A} + 1$.
- The micro-operation $A \leftarrow \overline{A} + 1$ is carried out using operations : $A \leftarrow \overline{A}$, $A \leftarrow A + 1$, $A_s \leftarrow \overline{A}_s$

Review Questions

1. *Write rule for addition of two numbers.*
2. *Perform subtraction of binary numbers in 2's complement method.*
3. *What is half adder? Design a half adder as a two-level AND-OR circuit and show how to implement a full adder using two half adders and a external logic gate.*
4. *Draw the half adder circuit.*
5. *Write the logic equations of a binary half a adder.*
6. *Draw the symbolic representation of the full-adder and give the expression for the sum.*
7. *Draw a full-adder circuit and give the truth table.*
8. *Draw the full adder circuit using two half adders.*
9. *Design the full adder circuit.*
10. *Draw and explain a block diagram of parallel adder.*
11. *What is a ripple carry adder ?*
12. *Draw and explain the block diagram of parallel subtractor.*
13. *Explain the working of parallel subtractor.*
14. *Indicate how an overflow is detected ?*
15. *Define overflow rule in addition.*
16. *Give the block diagram of the hardware implementation of addition and subtraction of signed number and explain the operations with flowchart.*
17. *Explain the procedure for addition and subtraction with signed-magnitude data with the help of flowchart.*
18. *A half-adder is a combinational logic circuit that has two inputs, x and y and two outputs, s and c, that are the sum and carry-out respectively, resulting from the binary addition of x and y*
    i) Design a half-adder as a two-level AND-OR circuit.
    ii) Show how to implement a full-adder using two half-adder and external logic gates as necessary.
    iii) Compare the longest logic delay path through the network derived in part ii) to that of the logic delay of the adder network implemented using basic gates.
19. *Explain the hardware for signed-magnitude addition subtraction with block diagram.*
20. *Design a 4-bit binary adder/subtractor and explain its functions.*

8.3 Multiplication Algorithms

Dec.-11

- The multiplication is a complex operation than addition and subtraction. It can be performed in **hardware** or **software**.
- Fig. 8.3.1 shows the usual algorithm for multiplying positive numbers by hand.
- Looking at this algorithms we can note following points :
  - Multiplication process involves generation of partial products, one for each digit in the multiplier. These partial products are then summed to produce the final product.
  - In the binary system the partial products are easily defined. When the multiplier bit is 0, the partial product is 0, and when the multiplier is 1, the partial product is the multiplicand.
  - The final product is produced by summing the partial products. Before summing operation each successive partial product is shifted one position to the left relative to the preceding partial product, as shown in the Fig. 8.3.1.

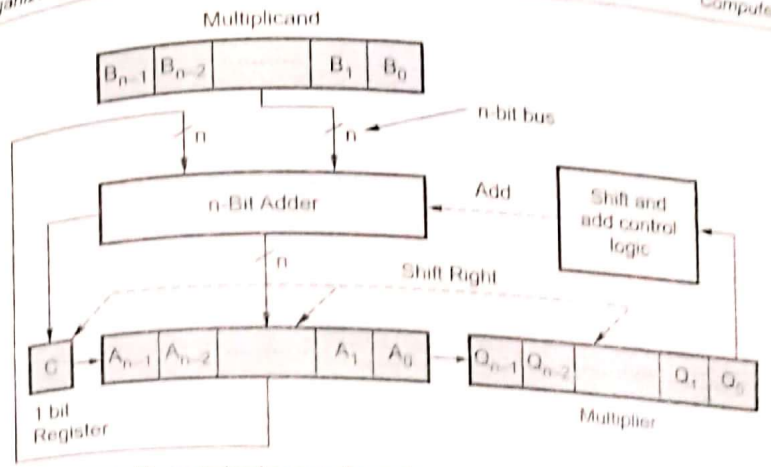
```
    1 1 0 1      (13)   Multiplicand
  × 1 0 0 1      (9)    Multiplier
  -----------
    1 1 0 1
  0 0 0 0
0 0 0 0                 Partial products
1 1 0 1
-----------------
1 1 1 0 1 0 1           Final product (117)
```

Fig. 8.3.1 Manual multiplication algorithm

  - The product of two n-digit numbers can be accommodated in 2n digits, so the product of the two 4 bit numbers in fits into 8-bits.
- Fig. 8.3.2 shows the implementation of manual multiplication approach. It consists of n-bit binary adder, shift and add control logic and four registers, A, B, C and Q.
- As shown in the Fig. 8.3.2 multiplier and multiplicand are loaded into register Q and register B, respectively, and C are initially set to 0.

Multiplication Operation Steps

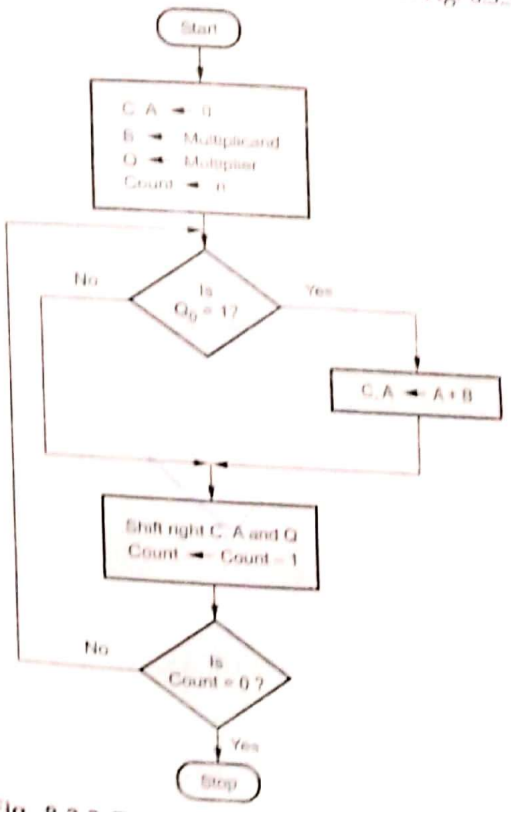
1. Bit 0 of multiplier operand (Q_0 of Q register) is checked.
2. If bit 0 (Q_0) is one then multiplicand and partial product are added and all bits of C, A and Q registers are shifted to the right one bit, so that the C bit goes into A_{n-1}, A_0 goes into Q_{n-1}, and Q_0 is lost. If bit 0 (Q_0) is 0, then no addition is performed, only shift operation is carried out.
3. Steps 1 and 2 are repeated n times to get the desired result in the A and Q registers.



Note : Dotted lines indicate control signals

Fig. 8.3.2 Hardware implementation of unsigned binary multiplication

- A flowchart for multiplication operation is shown in Fig. 8.3.3.



- Let us see one example.

Consider 4-bit multiplier and multiplicand :

Multiplicand = 1 1 0 1 and Multiplier = 1 0 1 1

Fig. 8.3.4 shows operations involved and their results in the multiplication process

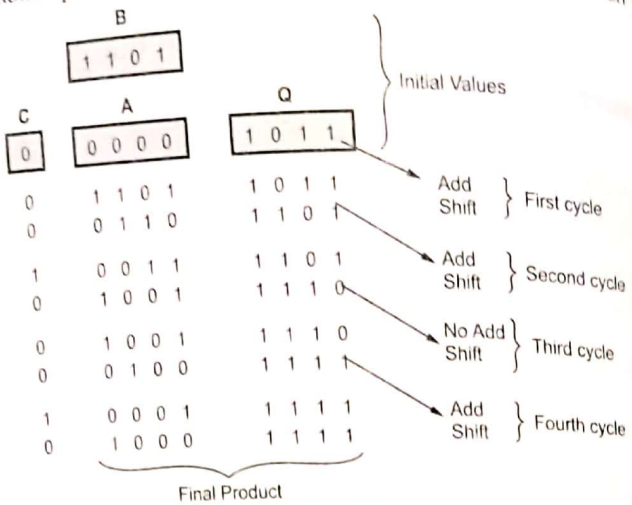


Fig. 8.3.4 Multiplication process

Review Questions

1. *Explain an algorithm to multiply two positive numbers. Also discuss the realization of a multiplier to implement the same.*
2. *Design a multiplier that multiplies two 4-bit numbers.*
3. *Explain the operation of sequential circuit binary multiplier with*
   Multiplicand 1101
   Multiplier 1011.

8.4 Signed Operand Multiplication

- To accumulate partial products by adding versions of multiplicand (either positive or negative) as selected by the multiplier bit.
- It is important to note that multiplicand when we add negative multiplicand we must extend the sign-bit value of the multiplicand to the left as far as product will extend.
- Refer Fig. 8.4.1. The sign extension of the multiplicand is shown bold.

8.4.1 Booth's Algorithm

- A powerful algorithm for signed-number multiplication is a Booth's algorithm, which generates a 2n-bit product and treats both positive and negative numbers uniformly.

	0	1	0	0	0	0	(16)
−	0	0	0	0	1	0	(2)
	0	0	1	1	1	0	(14)

- This algorithm suggest that we can reduce the number of operations required for multiplication by representing multiplier as a difference between two numbers. For example, multiplier 0 0 1 1 1 0 (14) can be represented as follows.
- Therefore, the product can be computed by adding 2^4 times the multiplicand to the 2's complement of 2^1 times the multiplicand. In simple notations, we can describe the sequence of required operations by **recoding** the preceding multiplier as

  0 + 1 0 0 − 1 0

- In general, for Booth's algorithm recoding scheme can be given as

−1 times the shifted multiplicand is selected when moving from 0 to 1, +1 times the shifted multiplicand is selected when moving from 1 to 0, and 0 times the shifted multiplicand is selected for none of the above case, as multiplier is scanned from right to left.

- We have to assume an implied 0 to right of the multiplier LSB. This is illustrated in the following examples.

Example 8.4.1 *Recode the multiplier 1 0 1 1 0 0 for Booth's multiplication.*

Solution :

							Implied zero
1	0	1	1	0	0	0	Multiplier
−1	+1	0	−1	0	0		Recoded multiplier

Example 8.4.2 *Recode the multiplier 0 1 1 0 0 1 for Booth's multiplication.*

Solution :

							Implied zero
0	1	1	0	0	1	0	Multiplier
+1	0	−1	0	+1	−1		Recoded multiplier

- The Fig. 8.4.1 shows the Booth's multiplication. As shown in the Fig. 8.4.1, whenever multiplicand is multiplied by −1, its 2's complement is taken as a partial result.

Multiplier 0 0 1 1 0 0 Multiplicand : 0 1 0 0 1 1

Recoded multiplier 0 + 1 0 – 1 0 0

						0	1	0	0	1	1	Multiplicand
					×	0	+1	0	–1	0	0	Multiplier
	0	0	0	0	0	0	0	0	0	0	0	0	
+	0	0	0	0	0	0	0	0	0	0	0		
+	1	1	1	1	1	0	1	1	0	1			← 2's complement of the multiplicand
+	0	0	0	0	0	0	0	0	0				
+	0	0	0	1	0	0	1	1					
+	0	0	0	0	0	0	0						
	0	0	0	0	1	1	1	0	0	1	0	0	

Note Shaded portion indicates sign extensions

Fig. 8.4.1 Booth's multiplication

- The same algorithm can be used for negative multiplier. This is illustrated in the following example.

Example 8.4.3 *Multiply 0 1 1 1 0 (+14) and 1 1 0 1 1 (–5).*

Solution :

0	1	1	1	0	(+14)	Multiplicand
1	1	0	1	1	(–5)	Multiplier
0	–1	1	0	–1		Recoded multiplier

Multiplication :

					0	1	1	1	0		Multiplicand
				×	0	–1	+1	0	–1		Recoded multiplier
	1	1	1	1	1	1	0	0	1	0	← 2's complement of the multiplicand
+	0	0	0	0	0	0	0	0	0		
+	0	0	0	0	1	1	1	0			
+	1	1	1	0	0	1	0				← 2's complement of the multiplicand
+	0	0	0	0	0	0					
	1	1	1	0	1	1	1	0	1	0	(–70)

Note Shaded portion indicates sign extensions

- The same algorithm also can be used for negative multiplier and negative multiplicand. This is illustrated in the following example.

Example 8.4.4 *Explain the following pair of signed 2's complement numbers.*
Multiplicand : 1 1 0 0 1 1 (–13)
Multiplier : 1 0 1 1 0 0 (–20)

Solution :

1	0	1	1	0	0	Multiplier
–1	+1	0	–1	0	0	Recoded multiplier

Multiplication :

						1	1	0	0	1	1	Multiplicand
					×	–1	+1	0	–1	0	0	Recoded multiplier
	0	0	0	0	0	0	0	0	0	0	0	0	
+	0	0	0	0	0	0	0	0	0	0	0		
+	0	0	0	0	0	0	1	1	0	1			← 2's complement of the multiplicand
+	0	0	0	0	0	0	0	0	0				
+	1	1	1	1	0	0	1	1					
+	0	0	0	1	1	0	1						← 2's complement of the multiplicand
	0	0	0	1	0	0	0	0	1	0	0		(260)

Note Shaded portion indicates sign extensions

Example 8.4.5 *Using Booth's algorithm multiply the following :*
Multiplicand = + 12 Multiplier = – 5

PU : May-13

Solution : Multiplicand = + 12 (0 1 1 0 0)

Multiplier = – 5 (1 1 0 1 1)

0	1	1	0	0	(+12)	Multiplicand
1	1	0	1	1	0 (–5)	Multiplier
0	–1	+1	0	–1		Recoded multiplier
						Implied zero

Multiplication :

						0	1	1	0	0	Multiplicand
					×	0	–1	+1	0	–1	Recoded multiplier
	1	1	1	1	1	1	0	1	0	0	← 2's complement of multiplicand
+	0	0	0	0	0	0	0	0	0		
+	0	0	0	0	1	1	0	0			
+	1	1	1	0	1	0	0				
+	0	0	0	0	0	0					
1	1	1	1	1	0	0	0	1	0	0	= (– 60)

Example 8.4.6 *Multiply (– 7) and (+ 3) using Booth's algorithm. Register size is 5 bits.* PU : May-11

Solution : Multiplicand = – 7 (1 1 0 0 1)

Multiplier = + 3 (0 0 0 1 1)

1	1	0	0	1		(–7) Multiplicand
0	0	0	1	1	0	(+3) Multiplier
0	0	1	0	–1		Recoded multiplier
						Implied zero

Multiplication :

					1	1	0	0	1	Multiplicand
				×	0	0	+1	0	–1	Recoded multiplier
	0	0	0	0	0	0	0	1	1	1	2's complement of multiplicand
+	0	0	0	0	0	0	0	0	0	
+	1	1	1	1	1	0	0	1		
+	0	0	0	0	0	0	0			
+	0	0	0	0	0	0				
	1	1	1	1	1	0	1	0	1	1	= (– 21)

Example 8.4.7 *Using Booth's algorithm multiply the following :*
Multiplicand = – 13 Multiplier = + 11 PU : Dec-11

Solution : Multiplicand = – 13 (1 0 0 1 1)

Multiplier = + 11 (0 1 0 1 1)

1	0	0	1	1		(–13) Multiplicand
0	1	0	1	1	0	(+11) Multiplier
1	–1	1	0	–1		Recoded multiplier
						Implied zero

Multiplication :

					1	0	0	1	1	Multiplicand
				×	+1	–1	+1	0	–1	Recoded multiplier
	0	0	0	0	0	1	1	0	1	
+	0	0	0	0	0	0	0	0	0	
+	1	1	1	1	0	0	1	1		
+	0	0	0	1	1	0	1			
+	1	1	0	0	1	1				
	1	1	0	1	1	1	0	0	0	1	= (– 143)

8.4.2 Hardware Implementation of Booth's Algorithm

- The Booth's algorithm can be implemented as shown in the Fig. 8.4.2.

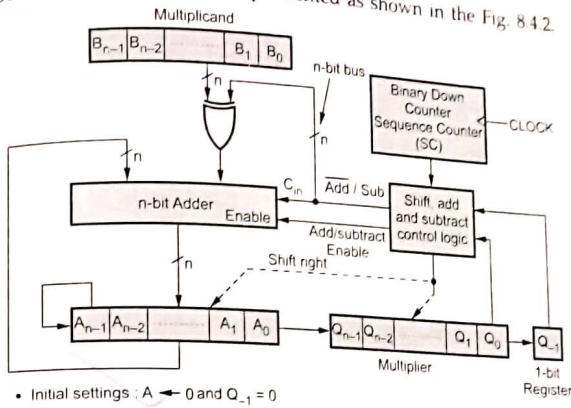


Fig. 8.4.2 Hardware implementation of signed binary multiplication

- It consists of n-bit adder, shift, add subtract control logic and four registers, A, B, Q and Q_{-1}.
- As shown in the Fig. 8.4.2 multiplier and multiplicand are loaded into register Q and register B, respectively, and register A and Q_{-1} are initially set to 0. The sequence counter, SC is set to a number n equal to the number of bits in the multiplier.
- The n-bit adder performs addition of two inputs. One input is the A register and other input is multiplicand. In case of addition, $\overline{\text{Add}}$/sub line is 0, therefore $C_{in} = 0$ and multiplicand is directly applied as a second input to the n-bit adder.
- In case of subtraction, $\overline{\text{Add}}$/sub line is 1, therefore $C_{in} = 1$ and multiplicand is complemented and then applied to the n-bit adder. As a result, the 2's complement of multiplicand is added in the A register.
- The shift, add and subtract control logic scans bits Q_0 and Q_{-1} one at a time and generates the control signals as shown in the Table 8.4.1.

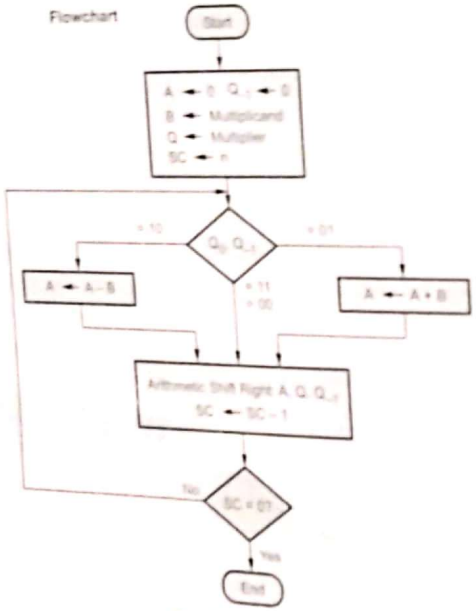
Q_0	Q_{-1}	$\overline{\text{Add}}$/Sub	Add/Subtract Enable	Shift
0	0	×	0	1
0	1	0	1	1
1	0	1	1	1
1	1	×	0	1

Table 8.4.1 Truth table for shift, add and subtract control logic

- If the two bits are same (1 – 1 or 0 – 0), then all of the bits of the A, Q, and Q_{-1} registers are shifted to right 1-bit without addition or subtraction (Add/subtract Enable = 0).
- If the two bits are differ, then the multiplicand (B-register) is added to or subtracted from the A register, depending on the status of bits.
- If bits are $Q_0 = 0$ and $Q_{-1} = 1$ then multiplicand is added and if bits are $Q_0 = 1$ and $Q_{-1} = 0$ then multiplicand is subtracted.
- After addition or subtraction right shift occurs such that the leftmost bit in A (A_{n-1}) is not only shifted into A_{n-2} but also remains in A_{n-1}. This is necessary to preserve the sign of the number in A and Q. It is known as an **arithmetic shift** since it preserves the sign bit.
- After arithmetic shift, the sequence counter is decremented by 1 and if it is not zero, computational loop is repeated. That is, in all computational loop is repeated n-times.

8.4.3 Flowchart of Booth's Multiplication Algorithm

The sequence of events in Booth's algorithm can be explained with the help of flowchart and algorithm shown in Fig. 8.4.3.



Algorithm

Step 1 Load A = 0, Q_{-1} = 0
B = Multiplicand
Q = Multiplier
SC = n

Step 2 Check the status of Q_0Q_{-1}
if Q_0Q_{-1} = 10 perform A ← A-B
if Q_0Q_{-1} = 01 perform A ← A+B

Step 3 Arithmetic shift right A, Q, Q_{-1}

Step 4 Decrement sequence counter
if not zero, repeat step 2 through 4

Step 5 Stop

Fig. 8.4.3 Booth's algorithm for signed multiplication

Let us see the multiplication of 4-bit numbers, 5 and 4 with all possible combinations.

Case 1 : Both Positive (5 × 4)

Multiplicand (B) = 0 1 0 1 (5) and Multiplier (Q) = 0 1 0 0 (4)											
SC			A				Q				Operation
			A_3	A_2	A_1	A_0	Q_3	Q_2	Q_1	Q_0	Q_{-1}	
1 0 0			0	0	0	0	0	1	0	0	0	Initial
0 1 1			0	0	0	0	0	0	1	0	0	Arithmetic shift right
0 1 0			0	0	0	0	0	0	0	1	0	Arithmetic shift right
			0	0	0	0	A					
		-	1	0	1	1	2's complement of B					$Q_0 Q_{-1} = 1\ 0$
			1	0	1	1	0	0	0	1	0	A ← A - B
0 0 1			1	1	0	1	1	0	0	0	1	Arithmetic shift right
			1	1	0	1	A					
		+	0	1	0	1	B					$Q_0 Q_{-1} = 0\ 1$
			0	0	1	0	1	0	0	0	1	A ← A + B
0 0 0			0	0	0	1	0	1	0	0	0	Arithmetic shift right
Result	=		0	0	0	1	0	1	0	0		= + $(20)_{10}$

Case 2 : Negative Multiplier (5 × – 4)

Multiplicand (B) = 0 1 0 1 (5) and Multiplier (Q) = 1 1 0 0 (– 4)											
SC			A				Q				Operation
			A_3	A_2	A_1	A_0	Q_3	Q_2	Q_1	Q_0	Q_{-1}	
1 0 0			0	0	0	0	1	1	0	0	0	Initial
0 1 1			0	0	0	0	0	1	1	0	0	Arithmetic shift right
0 1 0			0	0	0	0	0	0	1	1	0	Arithmetic shift right
			0	0	0	0	A					
		-	1	0	1	1	2's complement of B					$Q_0 Q_{-1} = 1\ 0$
			1	0	1	1	0	0	1	1	0	A ← A - B
0 0 1			1	1	0	1	1	0	0	1	1	Arithmetic shift right
0 0 0			1	1	1	0	1	1	0	0	1	Arithmetic shift right
Result	=		1	1	1	0	1	1	0	0		= – 20 (2's complement of 20)

Case 3 : Negative Multiplicand (– 5 × 4)

Multiplicand (B) = 1 0 1 1 (–5) and Multiplier (Q) = 0 1 0 0 (4)												
SC		A				Q					Operation
		A_3	A_2	A_1	A_0	Q_3	Q_2	Q_1	Q_0	Q_{-1}	
1 0 0		0	0	0	0	0	1	0	0	0	Initial
0 1 1		0	0	0	0	0	0	1	0	0	Arithmetic shift right
0 1 0		0	0	0	0	0	0	0	1	0	Arithmetic shift right
		0	0	0	0	A					$Q_0 Q_{-1}$ = 1 0
	+	0	1	0	1	2's complement of B					∴ A ← A – B
		0	**1**	**0**	**1**	0	0	0	1	0	
0 0 1		0	0	1	0	1	0	0	0	1	Arithmetic shift right
		0	0	1	0	A					$Q_0 Q_{-1}$ = 0 1
	+	1	0	1	1	B					∴ A ← A + B
		1	**1**	**0**	**1**	1	0	0	0	1	
0 0 0		1	1	1	0	1	1	0	0	0	Arithmetic shift right
Result : =		**1**	**1**	**1**	**0**	**1**	**1**	**0**	**0**		= – 20 (2's complement of 20)

Case 4 : Both Negative (– 5 × – 4)

Multiplicand (B) = 1 0 1 1 (–5) and Multiplier (Q) = 1 1 0 0 (– 4)											
SC		A				Q					Operation
		A_3	A_2	A_1	A_0	Q_3	Q_2	Q_1	Q_0	Q_{-1}	
1 0 0		0	0	0	0	1	1	0	0	0	Initial
0 1 1		0	0	0	0	0	1	1	0	0	Arithmetic shift right
0 1 0		0	0	0	0	0	0	1	1	0	Arithmetic shift right
		0	0	0	0	A					$Q_0 Q_{-1}$ = 1 0
	+	0	1	0	1	2's complement of B					∴ A ← A – B
		0	**1**	**0**	**1**	0	0	1	1	0	
0 0 1		0	0	1	0	1	0	0	1	1	Arithmetic shift right
0 0 0		0	0	0	1	0	1	0	0	1	Arithmetic shift right
Result : =		**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**		= + 20

Examples for Practice

Example 8.4.8 : Using Booth's algorithm multiply the following :
Multiplicand = + 13 Multiplier = – 5.

Example 8.4.9 : Using Booth's algorithm multiply
Multiplicand = – 13 Multiplier = + 15.

8.5 Fast Multiplication

- There are two techniques for speeding up the multiplication process. In first technique the maximum number of summands are reduced to n/2 for n-bit operands.
- The second technique, called the carry save addition reduces the time needed to add the summand.

8.5.1 Bit-Pair Recoding of Multipliers

- To speed-up the multiplication process in the Booth's algorithm a technique called **bit-pair recoding** is used. It is also called **modified Booth's algorithm**.
- It halves the maximum number of summands.
- In this technique, the Booth-recoded multiplier bits are grouped in pairs. Then each pair is represented by its equivalent single bit multiplier reducing total number of multiplier bits to half.
- For example pair (+ 1 –1) is equivalent to the pair (0 +1). That is, instead of adding –1 times multiplicand at shifted position i to +1 times the multiplicand at position i + 1, the same result is obtained by adding +1 times multiplicand at position i. Similarly, (+1 0) is equivalent to (0 +2), (–1 +1) is equivalent to (0 –1) and so on.
- By replacing pairs with their equivalents we can get bit-pair recoded multiplier.
- But instead of deriving bit-pair recoded multiplier from Booth recoded multiplier one can directly derive it from original multiplier.
- The bit-pair recoding of multiplier can be directly derived from Table 8.5.1. The Table 8.5.1 shows the bit-pair code for all possible multiplier bit options.

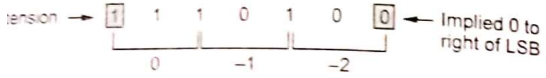
Multiplier bit-pair		Multiplier bit on the right	Bit-pair recoded multiplier bit at position i
i + 1	i	i – 1	
0	0	0	0

0	0	1	+ 1
0	1	0	+ 1
0	1	1	+ 2
1	0	0	- 2
1	0	1	- 1
1	1	0	- 1
1	1	1	0

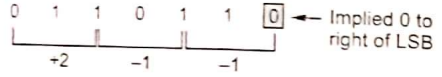
Table 8.5.1

he bit-pair code for multiplier.

ng table we can derive bit-pair code as follows :



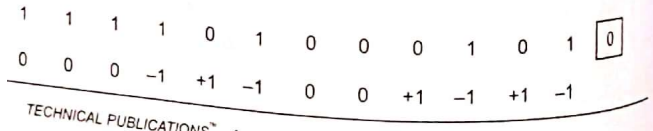
nly given signed 2's complement numbers using bit-pair recoding
multiplicand (–11)
multiplier (+27)
the bit-pair code for multiplier.



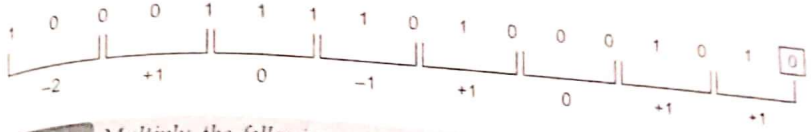
		1	1	0	1	0	1
	×		+2		−1		−1
0	0	0	0	1	0	1	1
0	0	1	0	1	1		
1	0	1	0				
1	1	0	1	0	1	1	1

Multiplicand
Multiplier
← 2's complement of the multiplicand
← 2's complement of the multiplicand
← Multiplicand × (+2)
(–297)

Booth's recoding and bit-pair recoding of the number.
) 0 0 1 0 1.
ling



Bit-Pair recoding



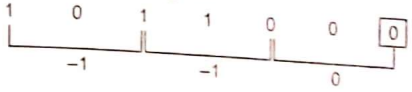
Example 8.5.4 *Multiply the following pair of signed 2's complement numbers using bit-pair recoding of the multipliers : A = 010111, B = 101100.*

Solution :

A = 0 1 0 1 1 1 Multiplicand (+ 23)

B = 1 0 1 1 0 0 Multiplier (– 20)

Let us find the bit-pair code for multiplier



Multiplication

					0	1	0	1	1	1
		×						−1	−1	0
	0	0	0	0	0	0	0	0	0	0
+	1	1	1	0	1	0	0	1		
+	1	0	1	0	0	1				
	1	0	0	0	1	1	0	1	0	0

2's complement of the multiplicand
2's complement of the multiplicand
(–460)

Example 8.5.5 *Explain Booth's algorithm to multiply the following pair of signed two's complement numbers :*
A = 110011 multiplicand
B = 101100 multiplier
Also, implement the above using bit-pair Recoding and explain how it achieves faster multiplication.

Solution :

1	0	1	1	0	1	0
−1	+1	0	−1	0	0	

Multiplier
Recoded multiplier

1	1	0	0	1	1	Multiplier
0	0	1	1	0	1	2's complement of multiplicand

						1	1	0	0	1	1
						−1	+1	0	−1	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	
0	0	0	0	0	1	1	0	1			
0	0	0	0	0	0	0	0				
1	1	1	0	0	1	1					
0	0	1	1	0	1						
0	0	0	1	0	0	0	0	1	0	0	

Multiplicand
Recoded multiplier

2's complement of multiplicand

Multiplicand
2's complement of multiplicand
(+260)

Implementation with Bit-Pair Recoding

Bit-pair Recoding for Multiplier

Multiplication

						1	1	0	0	1	1
					×			−1	−1	0	
	0	0	0	0	0	0	0	0	0	0	0
+	0	0	0	0	0	1	1	0	1		
+	0	0	0	1	1	0	1				
	0	0	1	0	0	0	0	0	1	0	0

Multiplicand × 0
2's complement of the multiplicand
2's complement of the multiplicand
(+260)

The Booth's algorithm may need the summation at each step and number of steps required in Booth's algorithm are equal to length of multiplier in bits. The bit-pair recoding halves the maximum number of summations. Hence it achieves faster multiplication.

Example 8.5.6 *Show the multiplication of (+13) and (– 6) using multiplier bit pair recoding technique.*

Solution : + 13 = 0 1 1 0 1

– 6 = 1 0 1 0 2's complement of 6

Bit pair recoding of – 6

Sign extension → **1** 1 0 1 0 **0** ← Implied 0 to right of LSB

−1 −2

Multiplication

		0	1	1	0	1
0	1	1	0	1	0
1	0	0	1	1	0

13
13 × 2 by shift left 1-bit
2's complement of 13 × 2

			1	1	0	1
		×		−1		−2
	1	1	0	0	1	1	0
+	1	0	0	1	1		
1	0	1	1	0	0	1	0

2's complement of 13 × 2
2's complement of 13
2's complement of 78

$\therefore (+13) \times (-6) = (10110010)_2 = (-78)_{10}$

Example for Practice

Example 8.5.7 : *Illustrate multiplication of signed 2's complement numbers 01101 and 11010 using bit-pairing of the multipliers.* **PU : Dec.-11**

8.5.2 Array Multiplier

- The multiplication process for binary numbers is similar to the decimal numbers.
- Actually binary multiplication is simple than decimal multiplication since it involves only 1s and 0s.

Rules for binary multiplication

- $0 \times 0 = 0$
- $1 \times 0 = 0$
- $0 \times 1 = 0$
- $1 \times 1 = 1$

Example 8.5.8 *Multiply 011_2 by 110_2 using binary multiplication method.*

Solution :

			0	1	1
		×	1	1	0
			0	0	0
+		0	1	1	0
+	0	1	1	0	0
	1	0	0	1	0

← Shift left
← Shift left

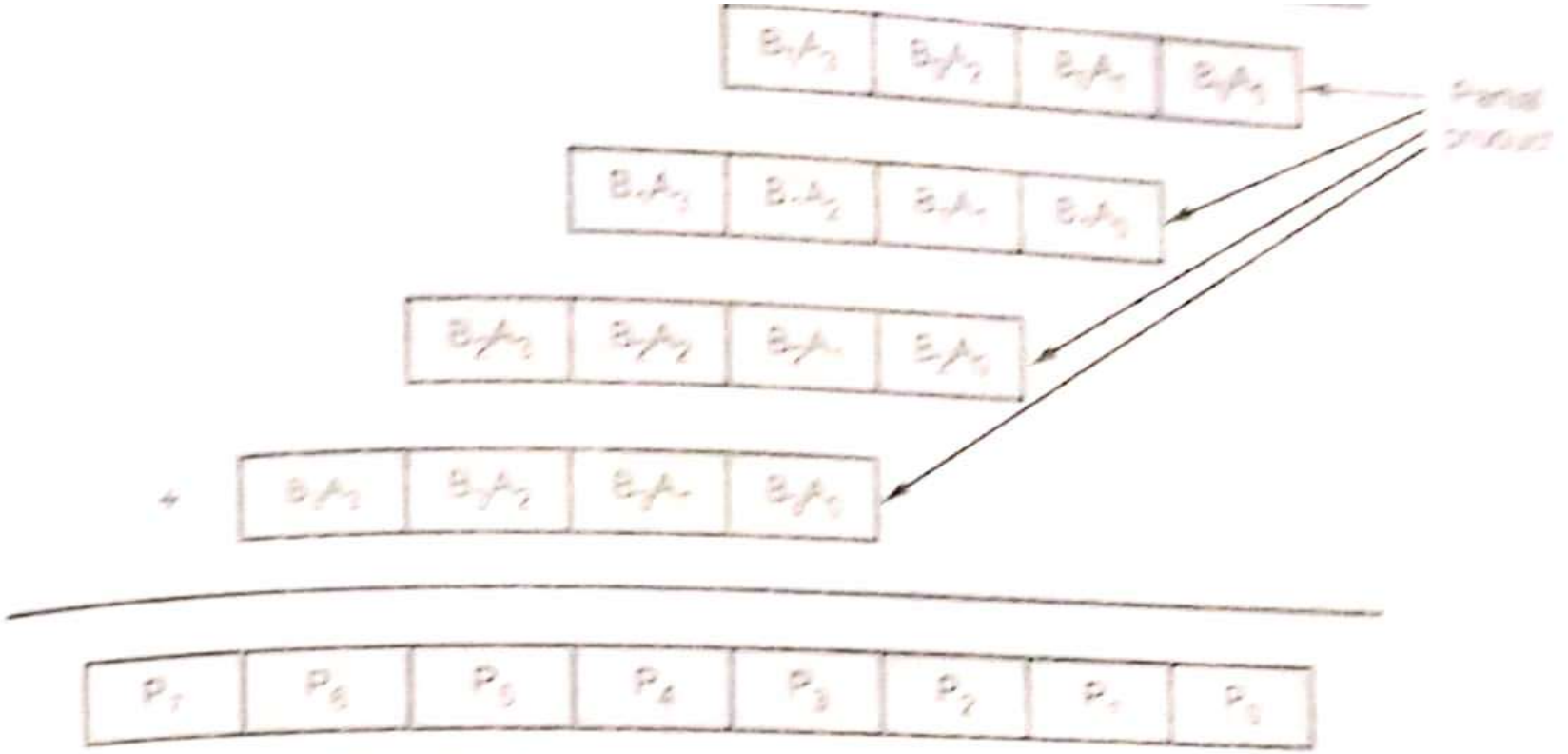


Fig. 8.5.3 4 bit by 4-bit binary multiplication

- The first partial product is formed by multiplying B_0 by A_3 A_2 A_1 A_0. The second partial product is formed by multiplying B_1 by A_3 A_2 A_1 A_0. The third partial is formed by multiplying B_2 by A_3 A_2 A_1 A_0 and the fourth partial product is formed by multiplying B_3 by A_3 A_2 A_1 A_0.
- The multiplication of two bits such as B_j and A_i produces a 1 if both bits are 1, otherwise, it produces a 0. This is identical to AND operation. Therefore, the partial products can be implemented with AND gates as shown in Fig. 8.5.4.

rectangular box.

- The final 8-bit product is obtained by adding all partial products.

Fig. 8.5.5 Multiplication process of 4 × 4 multiplier

- The Fig. 8.5.6 shows the circuit to add the product components. Here, the product components are separated to make the space, and each `+' box is a full adder.

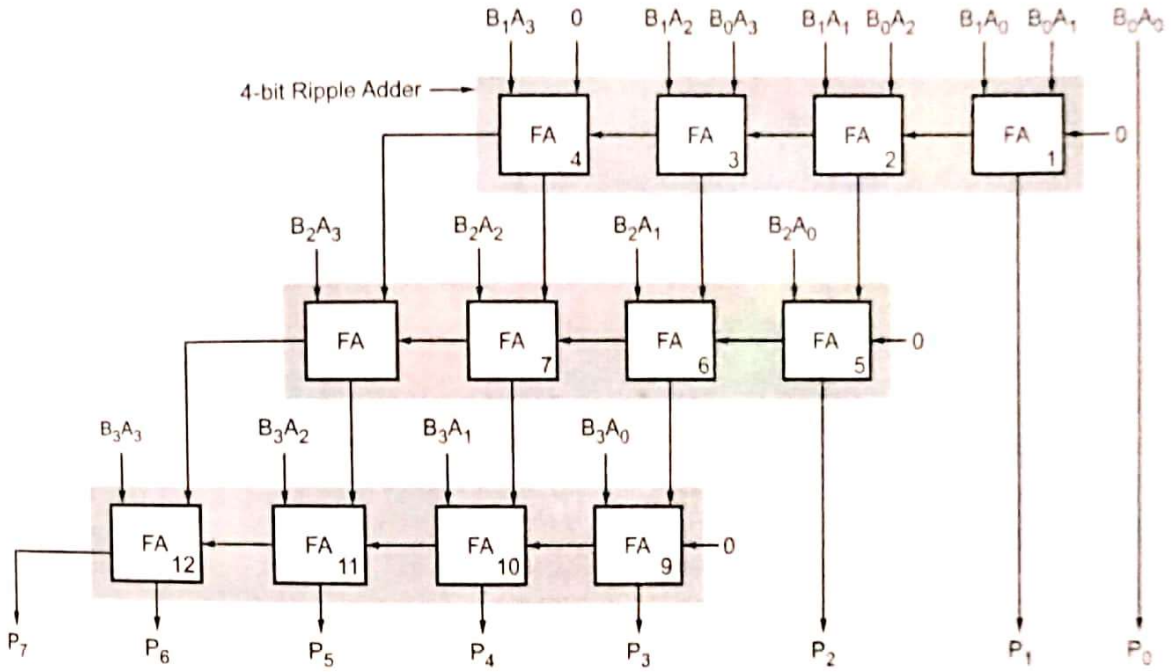


Fig. 8.5.6 4 × 4 combinational multiplier

- The carries in each partial product row of full adders are connected to make an 4-bit ripple adder. Thus, the first 4-bit ripple adder adds the first two rows of product components to produce the first partial product.
- The carry output generated is propagated to the most significant product component used to produce the next partial product.

- The subsequent adders adds each partial product with the next product component.
- In the above multiplier, the least significant adder is full-adder 1 and the adder which gives the MSB of the product (P_7) is full-adder 12.
- If we assume for simplicity that the delays from any input to any output of a full adder are equal, say t_{pd}, then the worst-case path goes through 8 full-adders (1, 2, 3, 4, 7, 8, 11 and 12) and its delay is $8\ t_{pd}$.
- To increase the speed of the addition process many times technique called **carry-save addition** is used. In this technique, the carry output from bit i during step j is applied to carry input for bit i + 1 during the next step, j + 1. After addition of product components in the last row, one more step is required in which the carries are allowed to ripple from the least to the most significant bit.
- The Fig. 8.5.7 shows 4 × 4 combinational multiplier using carry-save addition technique. In this the carry out of each full adder in the first seven row of full adders are connected to an input of an adder below it. Carries in the fourth row of full adders are connected to create a conventional ripple adder. This technique does not save any hardware but it reduces the propagation delay substantially.

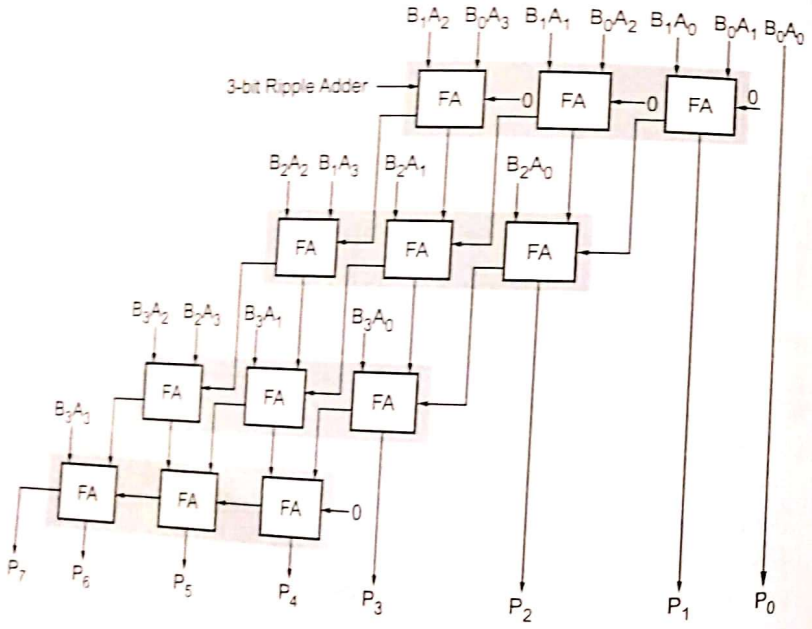


Fig. 8.5.7 4 × 4 combinational multiplier with carry save addition

Review Questions

1. Discuss the role of Booth's algorithm in the design of fast multipliers.
2. Explain the modified Booth's algorithm.
3. Design an array multiplier that multipliers two 4 bit numbers. Use AND gates and binary address.
4. Draw and explain 2-bit by 2-bit array multiplier.
5. Perform signed multiplication of numbers –12 and –11 using booth multiplication algorithm. Represent the numbers in 5-bits including sign bit. Give booth multiplier recoding table that is used in the above multiplication.
6. Explain the concept of carry save addition for the multiplication operation, M×Q = P for 4-bit operands, with diagram and suitable example.
7. Carry out bit pair recoding of following multipliers
   1 1 0 1 0
   0 1 1 0 1.

8.6 Division Algorithms

- The division process for binary numbers is similar to the decimal numbers. The Fig. 8.6.1 shows binary division process.

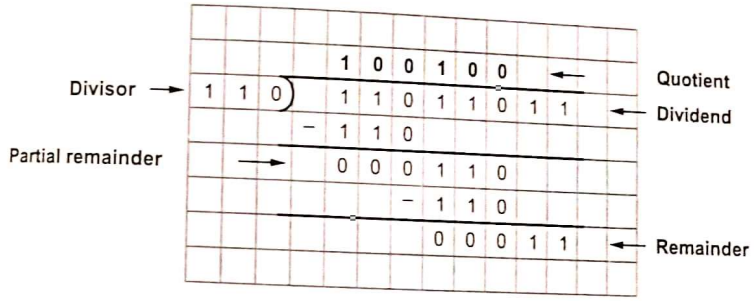


Fig. 8.6.1 Division examples

- In the division process, first the bits of the dividend are examined from left to right, until the set of bits examined represents a number greater than or equal to the divisor.
- Until this condition occurs, 0's are placed in the quotient from left to right.
- When the condition is satisfied, a 1 is placed in the quotient and the divisor is subtracted from the partial dividend. The result is referred to as a **partial remainder.**
- From this point onwards, the division process is required. In each repetition cycle, additional bits from the dividend are brought down to the partial remainder until

the result is greater than or equal to the divisor and the divisor is subtracted from the result to produce a new partial remainder.

- The process continues until all the bits of the dividend are brought down and result is still less than the divisor.

8.6.1 Restoring Division Algorithm

- Fig. 8.6.2 shows the hardware for implementation of restoring division.
- It consists of n+1-bit binary adder, shift, add and subtract control logic and registers A, B, and Q.
- As shown in the Fig. 8.6.2 divisor and dividend are loaded into register B and register Q, respectively. Register A is initially set to zero. The division operation is then carried out.
- After the division is complete, the n-bit quotient is in register Q and the remainder is in register A.

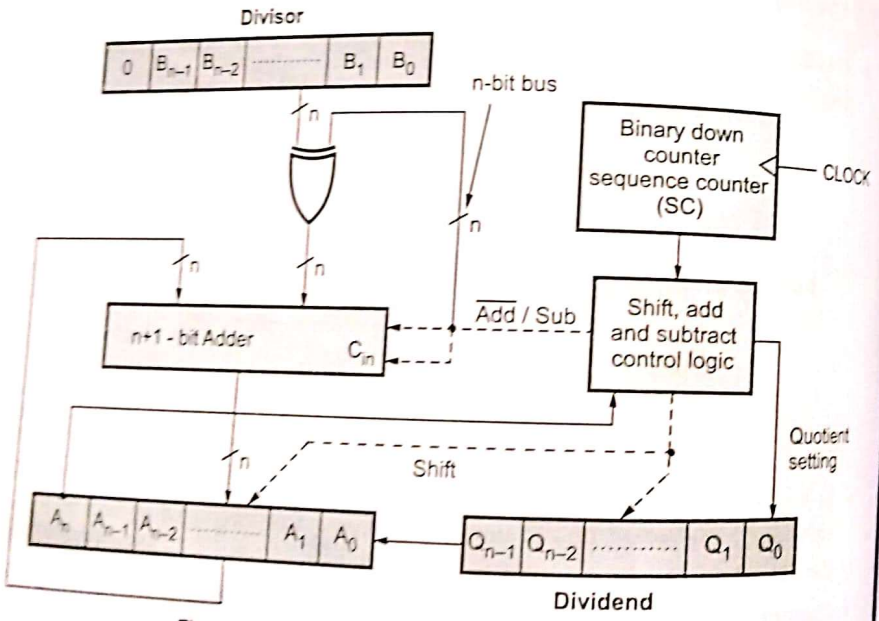


Fig. 8.6.2 Hardware to implement binary division

Division operation steps :

1. Shift A and Q left one binary position.
2. Subtract divisor (i.e. add 2's complement of divisor (B)) from A and place answer back in A ($A \leftarrow A - B$).

3. If the sign bit of A is 1, set Q_0 to 0 and add divisor back to A (that is, restore A); Otherwise, set Q_0 to 1.
4. Repeat steps 1, 2, and 3 n times.

A flowchart for division operation is as shown in Fig. 8.6.3.

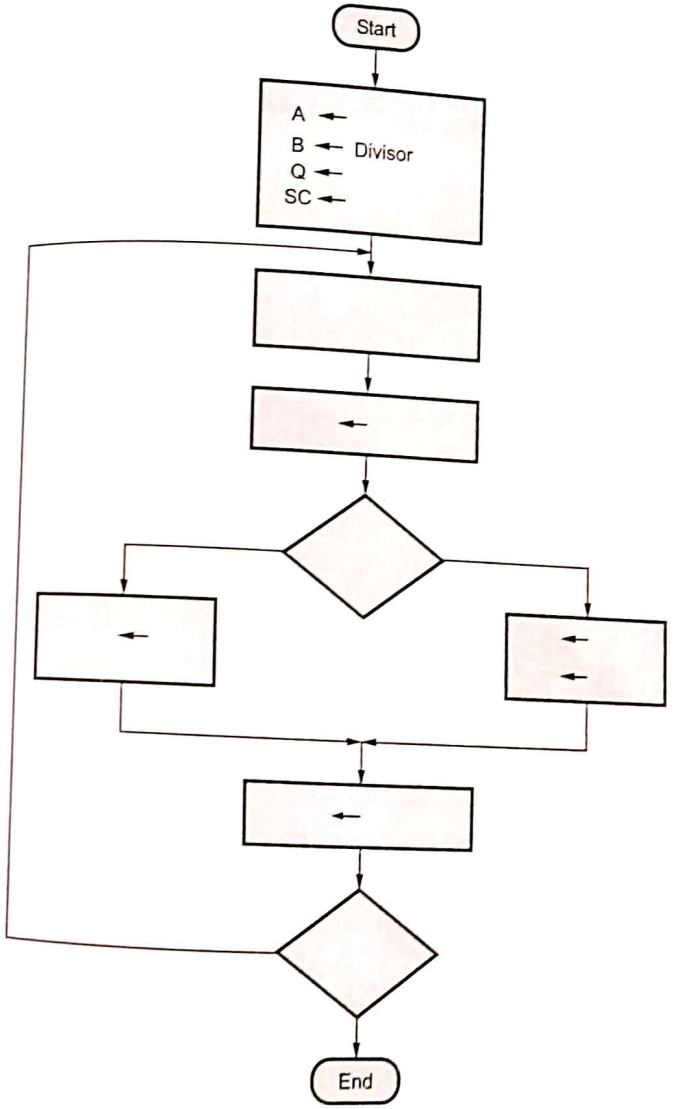


Fig. 8.6.3 Flowchart for restoring division operation

Example 8.6.1 *Perform the division of following numbers using restoring division algorithm :*
Dividend = 1 0 1 0
Divisor = 0 0 1 1

PU : Dec.-12

Solution : Fig. 8.6.2 shows steps involved in the above binary division.

	0	0	0	1	1
	1	1	1	0	0
+					1
	1	1	1	0	1

Divisor
1's complement
Add 1
2's complement of divisor

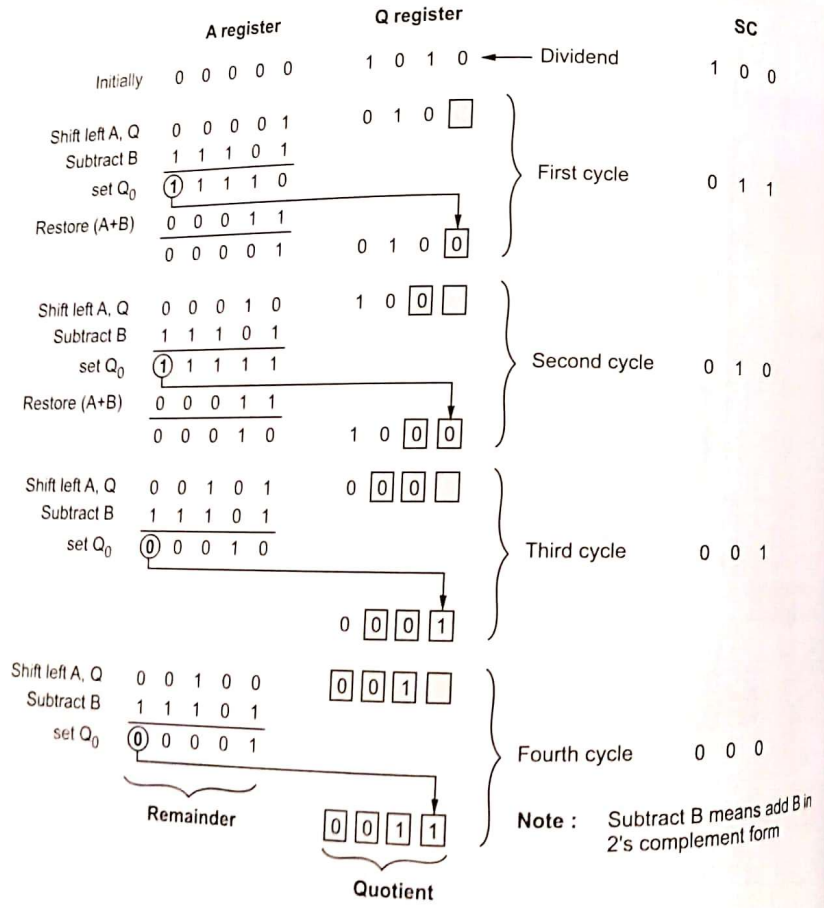


Fig. 8.6.4 A restoring division example

Example 8.6.2 *Divide the following unsigned numbers using restoring division method*
Dividend = 1000, Divisor = 11.

PU : Dec.-09, May-10

Solution :

	0	0	0	1	1
	1	1	1	0	0
+					1
	1	1	1	0	1

Divisor
1's complement
Add 1
2's complement

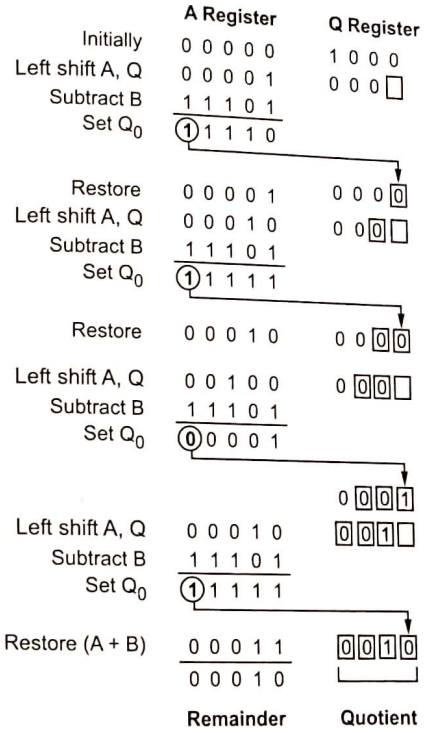


Fig. 8.6.5

Example 8.6.3 *Perform the following division using restoring division algorithm :*
Dividend = 1001
Divisor = 0101.

PU : Dec.-10

Solution :

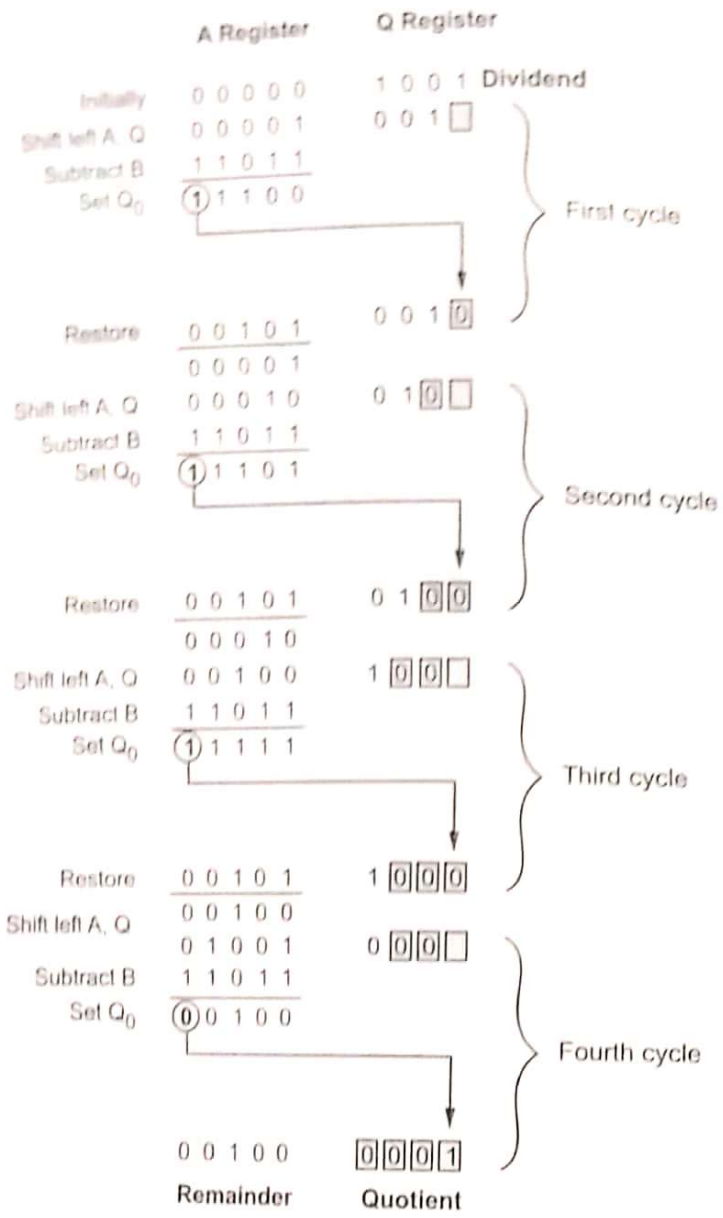


Fig. 8.6.6

Example 8.6.4 *Perform 1100 ÷ 11 using restoring algorithm .*

PU : May-05, 08, 12

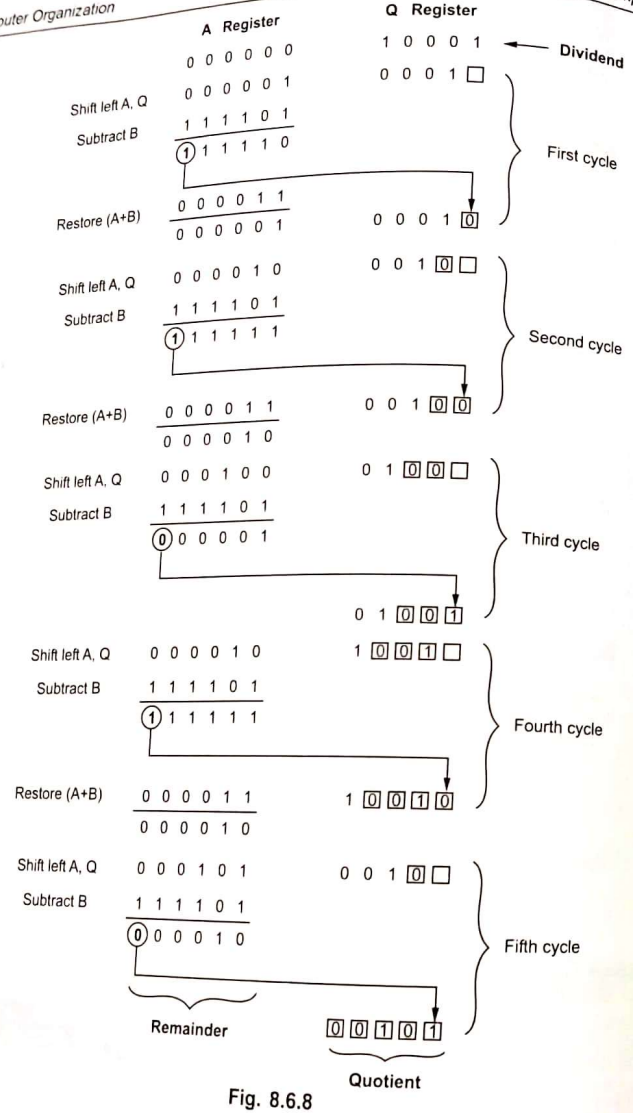


Fig. 8.6.8

- The division algorithm just discussed needs restoring register A after each unsuccessful subtraction. (Subtraction is said to be unsuccessful if the result is negative). Therefore it is referred to as **restoring division algorithm.**

Example 8.6.6 *Perform restoring division when divisor V = 3 and dividend D = 7. Also give the results for all possible combinations of signs of D and V.*

Solution : The restoring division algorithm assumes positive signs of D and V.

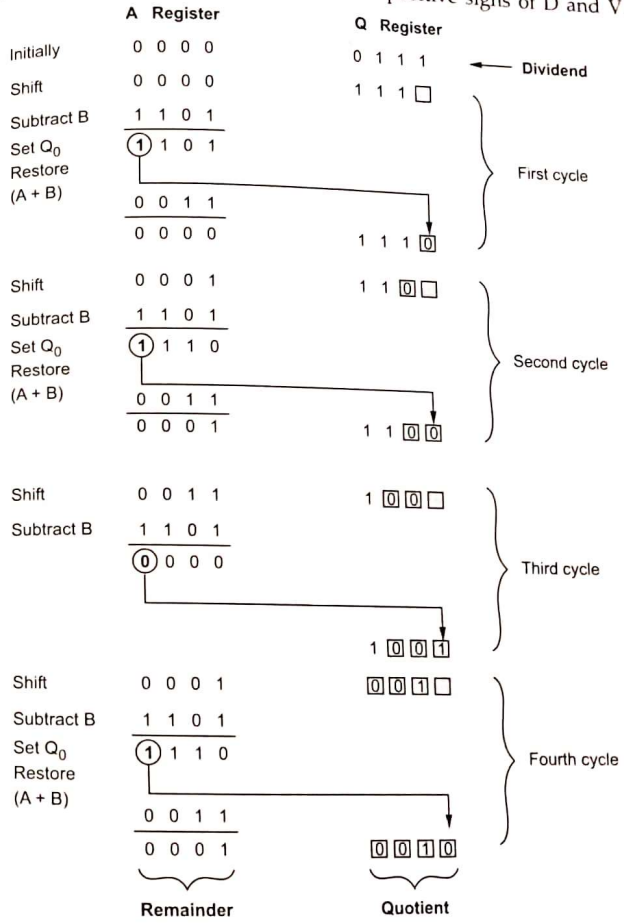


Fig. 8.6.9

∴ Remainder = $(0001)_2 = 1$ and Quotient = $(0010)_2 = 2$

In the division operation, the magnitudes of quotient (Q) and remainder (R) are unaffected by the input signs. The signs of Q and R are easily derived from the signs of D and V using expression : $D = Q \times V + R$

The signs of Q and R for all possible combinations of signs of D and V are as follows :

$D = 7, \quad V = 3 \quad = \quad Q = 2, \quad R = 1$
$D = 7, \quad V = -3 \quad = \quad Q = -2, \quad R = 1$
$D = -7, \quad V = 3 \quad = \quad Q = -2, \quad R = -1$
$D = -7, \quad V = -3 \quad = \quad Q = 2, \quad R = -1$

8.6.2 Non-Restoring Division

- Consider the sequence of operations that takes place after the subtraction operation in the restoring algorithm.

If A is positive	If A is negative
Shift left and subtract divisor → 2A – B	Restore → A + B Shift left and subtract divisor → 2 (A + B) – B = 2A + B

- Looking at the above operations we can write following steps for non-restoring algorithm.

Step 1 : If the sign of A is 0, shift A and Q left one bit position and subtract divisor from A, otherwise, shift A and Q left and add divisor to A. If the sign of A is 0, set Q_0 to 1; otherwise, set Q_0 to 0.

Step 2 : Repeat steps 1 and 2 for n times.

Step 3 : If the sign of A is 1, add divisor to A.

Note : Step 3 is required to leave the proper positive remainder in A at the end of n cycles.

- A flowchart for non-restoring division operation is as shown in Fig. 8.6.10

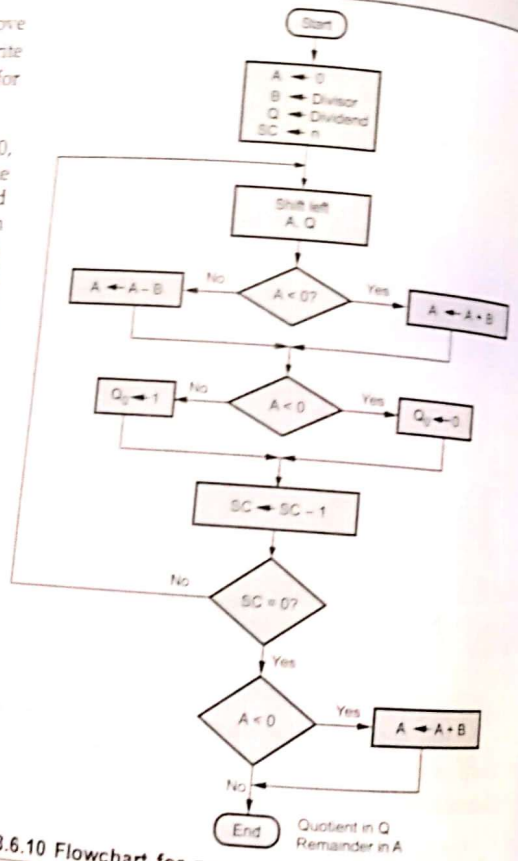


Fig. 8.6.10 Flowchart for non-restoring division operation

Example 8.6.7 *Perform the division of following numbers using non-restoring division algorithm :*

Dividend = 1 0 1 0

Divisor = 0 0 1 1.

Solution : Fig. 8.6.11 shows steps involved in the non-restoring binary division.

PU - Dec.-12

	A register	Q register	SC	
Initially	0 0 0 0 0	1 0 1 0		← Dividend
Shift left A, Q	0 0 0 0 1	0 1 0	1 0 0	
Subtract B	1 1 1 0 1			
set Q_0	(1) 1 1 1 0	0 1 0 0		First cycle
Shift left A, Q	1 1 1 0 0	1 0 0	0 1 1	
Add B	0 0 0 1 1			
set Q_0	(1) 1 1 1 1	1 0 0 0		Second cycle
Shift left A, Q	1 1 1 1 1	0 0 0	0 1 0	
Add B	0 0 0 1 1			
set Q_0	(0) 0 0 1 0	0 0 0 1		Third cycle
Shift left A, Q	0 0 1 0 0	0 0 1	0 0 1	
Subtract B	1 1 1 0 1			
	(0) 0 0 0 1	0 0 1 1		Fourth cycle
	Remainder	Quotient	0 0 0	

Note : Subtract B means add B in 2's complement form

Fig. 8.6.11 A non-restoring division example

In this example, after 4 cycles A is positive and hence step 3 is not required.

Example 8.6.8 *Perform the division of following numbers using non-restoring division algorithm :*

Dividend = 1 0 1 1

Divisor = 0 1 0 1.

Solution :

0	0	1	0	1	Divisor
1	1	0	1	0	1's complement
				1	Add 1
1	1	0	1	1	2's complement

- Fig. 8.6.12 shows steps involved in the non-restoring binary division.

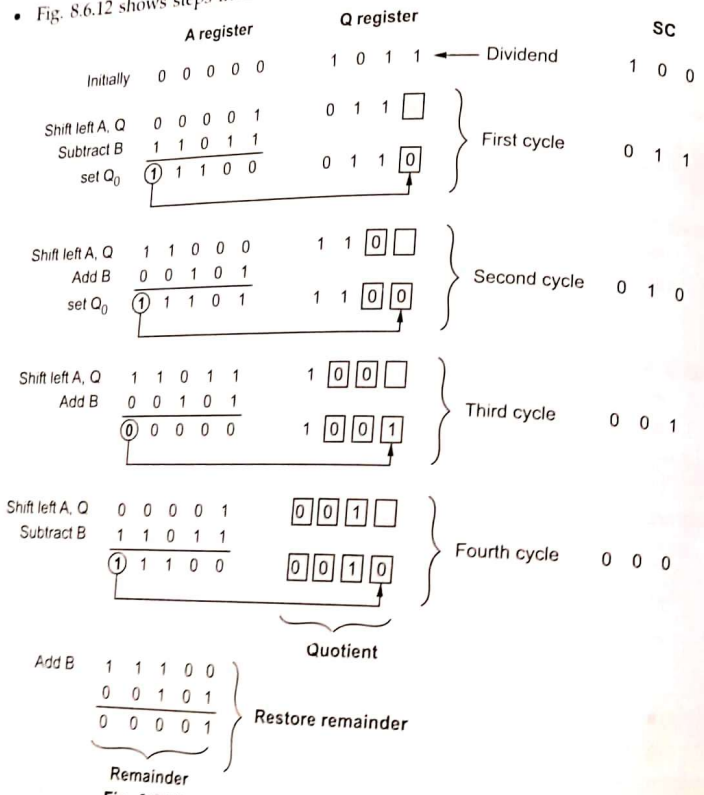


Fig. 8.6.12 A non-restoring division example

- The hardware shown in Fig. 8.6.2 can also be used to perform non-restoring algorithm. There is no simple algorithm for signed division. In signed division, the operands are pre-processed to transform them into positive values. Then using one of the algorithms just discussed quotients and remainders are calculated. The quotients and remainders are then transformed to the correct signed values.

Example 8.6.9 *Divide the following unsigned numbers using non-restoring division method. Dividend = $(15)_{10}$, Divisor = $(2)_{10}$* **PU : May-11, Dec.-11**

Solution : Dividend = $(15)_{10} = (01111)_2$

Divisor = $(2)_{10} = (00010)_2$

0	0	0	1	0	Divisor
1	1	1	0	1	1's complement
				1	Add 1
1	1	1	1	0	2's complement

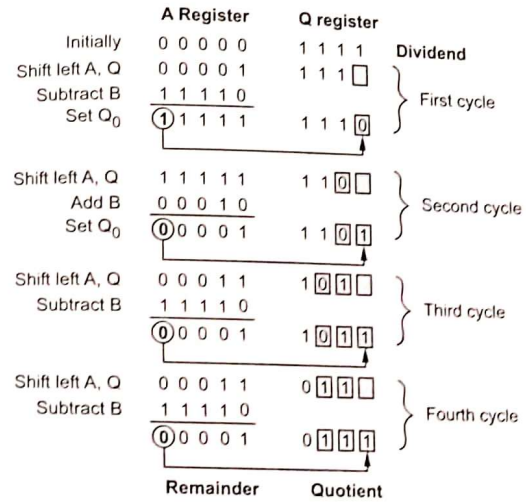


Fig. 8.6.13

Example 8.6.10 *Perform division operation on the following numbers using the non-restoring division algorithm. Dividend = 1101 Divisor = 0100* **PU : Dec.-07, 12**

Solution :

0	0	1	0	0	Divisor (B)
1	1	0	1	1	1's complement
				1	Add 1
1	1	1	0	0	2's complement

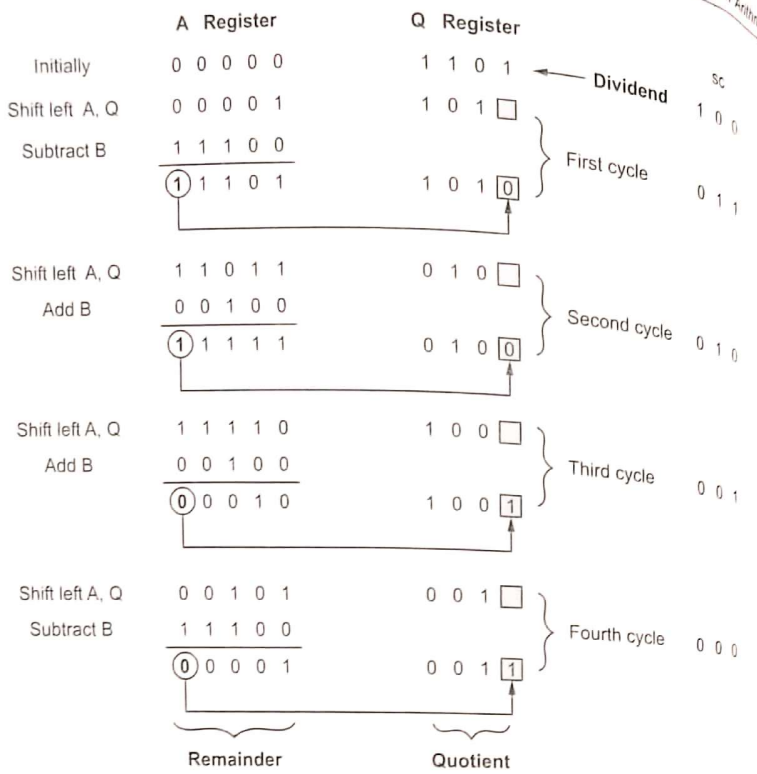


Fig. 8.6.14

Example 8.6.11 *Perform 1100 ÷ 11 using non-restoring algorithm.* **PU : May-05,**

Solution : Non-Restoring Division : Q ← Dividend (1100), B ← Divisor (11).

Complement and divisor (B) = 11101.

(See Fig. 8.6.15 on next page)

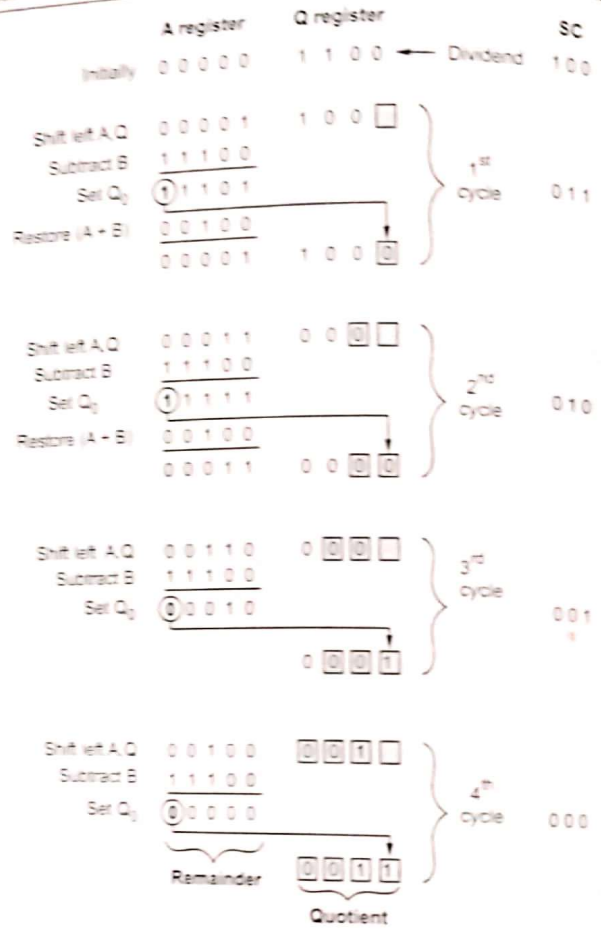


Fig. 8.6.16

Note *Subtract B means add B in 2's complement form.*

A = 1100 B = 0100 (Using non restoring division)

0	1	0	0	Divisor
1	0	1	1	1's complement
			1	Add 1
1	1	0	0	2's complement

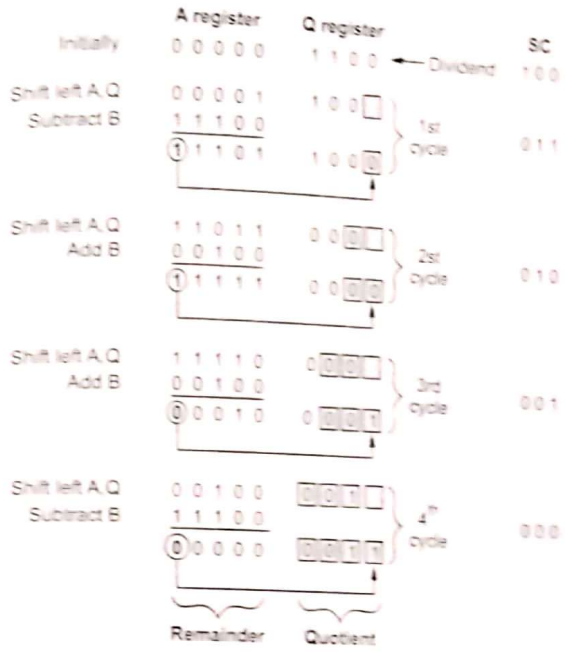


Fig. 8.6.17

8.6.3 Comparison between Restoring and Non-Restoring Division Algorithm

Sr. No.	Restoring	Non-restoring
1.	Needs restoring of register A if the result of subtraction is negative.	Does not need restoring.
2.	In each cycle content of register A is first shifted left and then divisor is subtracted from it.	In each cycle content of register A is first shifted left and then divisor is added or subtracted with the content of register A depending on the sign of A.
3.	Does not need restoring of remainder.	Needs restoring of remainder if remainder is negative.
4.	Slower algorithm.	Faster algorithm.

Examples for Practice

Example 8.6.13 *Perform the following division using non-restoring division algorithm. Dividend = 1101, Divisor = 11.*

Example 8.6.14 *Perform the following division using restoring division algorithm. Dividend = 1100 Divisor = 0100.*

Example 8.6.15 *Perform the following division using non-restoring division algorithm. Dividend = 1000, Divisor = 11.*

Example 8.6.16 *Compare restoring and non-restoring division algorithm. Perform the division using restoring division algorithm. Dividend = 17; Divisor = 03.*

Example 8.6.17 *Draw flowchart of Booth's algorithm for non-restoring unsigned division and divide the following unsigned numbers and justify your answer. Dividend = 1011 ; Divisor = 0011.*

Review Questions

1. *Draw the flowchart for restoring division algorithm.*
2. *Draw the flowchart for non-restoring division algorithm.*
3. *Compare restoring and non-restoring division algorithm.*
4. *Write restoring unsigned division algorithm.*
5. *Perform the following division using restoring and nonrestoring division algorithm. Dividend = 1100 Divisor = 0011*
6. *Perform following division using restoring and non-restoring algorithm. Dividend = 1010 and Divisor = 0011.*
7. *Perform division of the following numbers using restoring and non restoring division algorithm. A : 1100 and B : 0100*

8.7 Floating Point Arithmetic Operations

In this section we are going to see general procedures for addition, subtraction, multiplication and division of floating point numbers.

8.7.1 Addition and Subtraction

Consider two floating point numbers :

$A = m_1 \cdot r^{e_1}$ and

$B = m_2 \cdot r^{e_2}$ Assume : $e_1 \geq e_2$

Let us see the rules for addition and subtraction.

Step 1 : Select the number with a smaller exponent and shift its mantissa right, a number of steps equal to the difference in exponents $|e_2 - e_1|$. For examples, if the numbers are 1.75×10^2 and 6.8×10^4, then the number 1.75×10^2 is selected and converted to 0.0175×10^4.

Step 2 : Set the exponent of the result equal to the larger exponent.

Step 3 : Perform addition/subtraction on the mantissas and determine the sign of the result.

Step 4 : Normalize the result, if necessary.

Example 8.7.1 *Perform addition and subtraction of single precision floating point numbers A and B, where A = 44900000H and B = 42A00000H*

Solution : Step 1 : Represent numbers in single precision format

A = 0 1 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0

B = 0 1 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0

Exponent for A = 1 0 0 0 1 0 0 1 = 137

∴ Actual exponent = 137 – 127 (Bias) = 10

Exponent for B = 1 0 0 0 0 1 0 1 = 133

∴ Actual exponent = 133 – 127 (Bias) = 6

∴ Number B has smaller exponent with difference 4. Hence its mantissa is shifted right by 4-bits as shown below.

Step 2 : Shift mantissa : Shifted mantissa of B = 0 0 0 0 0 1 0 0 0

Step 3 : Add mantissas :

Mantissa of A = 0 0 1 0 0 0 0 0 0

Mantissa of B = 0 0 0 0 0 1 0 0 0

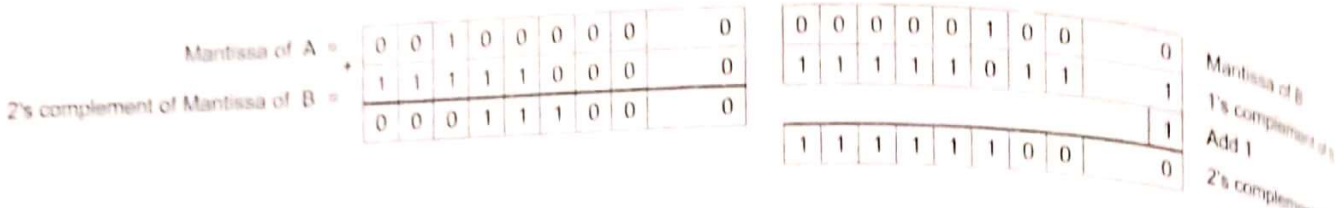
Mantissa of result = 0 0 1 0 0 1 0 0 0

As both numbers are positive, sign of the result is positive.

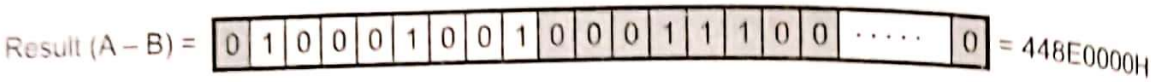
Result (A + B) = 0 1 0 0 0 1 0 0 1 0 0 1 0 1 0 0 ..... 0 = 44920000 H

- In subtraction, two mantissas are subtracted instead of addition and the sign of greater mantissa is assigned to the result.

Step 4 : Subtract mantissa



Mantissa for A is greater than mantissa for B therefore sign of result is sign of A



8.7.1.1 Overflows and Underflows in Mantissa and Exponent

Mantissa overflow : The addition of two mantissas of the same sign may result in a carryout of the most significant bit. If so, the mantissa is shifted right and the exponent is incremented.

Mantissa underflow : In the process of aligning mantissas, digits may flow off the right end of the mantissa. In such case truncation methods such as chopping, rounding are used.

Exponent overflow : Exponent overflow occurs when a positive exponent exceeds the maximum possible exponent value. In some systems this may be designated as $+\infty$ or $-\infty$.

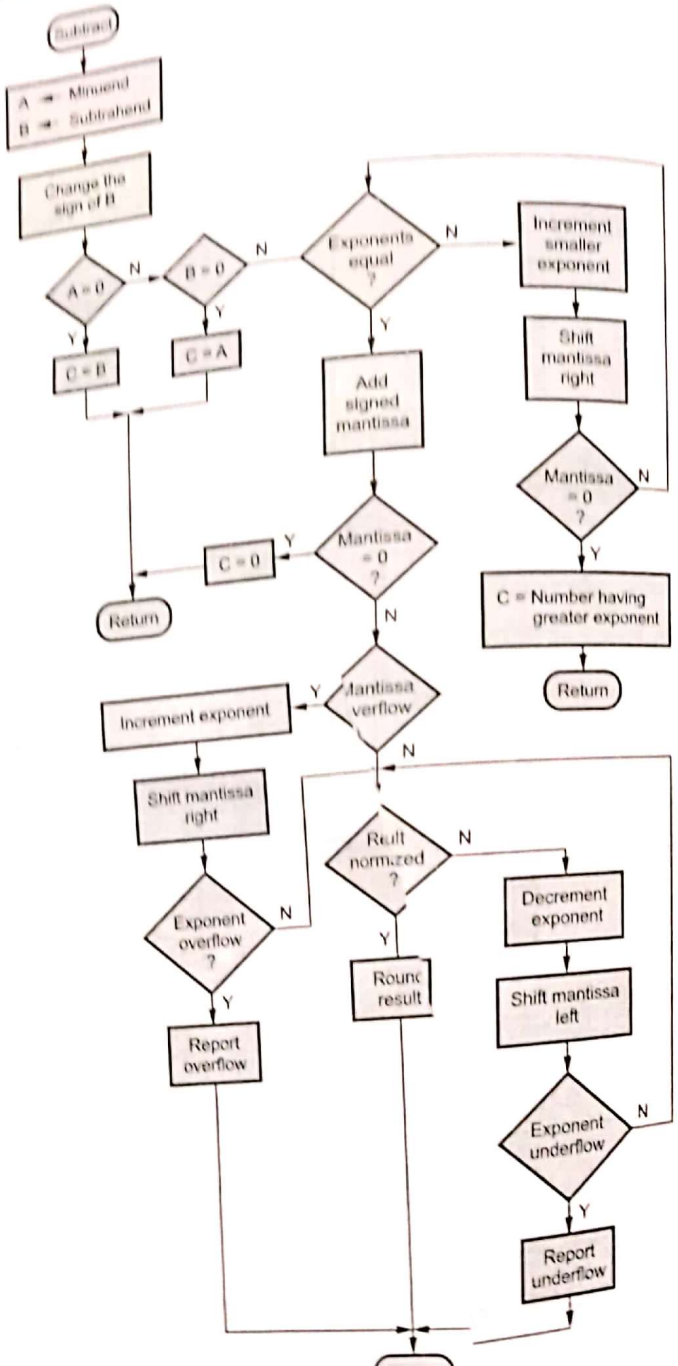
Exponent underflow : Exponent underflow occurs when a negative exponent exceeds the maximum possible exponent value. In such cases, the number is designated as zero.

8.7.1.2 Flowchart for Floating Point Addition and Subtraction

The Fig. 8.7.1 (a) and (b) shows the flowcharts for floating point addition and subtraction.

Phase 1 : Changing sign of B for subtraction and zero check.

- Addition and subtraction are identical except for a sign change in case of subtraction.
- If either operand is zero, the other is reported as the result.



Phase 2 : Align mantissa

- The mantissa of the smaller exponent is shift right a number of times equal to the difference in exponents.
- In this process, if mantissa is 0 then the number having higher exponent is reported as the result.

Phase 3 : Addition

- Two mantissas are added together, taking into account their sign. Because the signs may differ and result may be 0.
- There is also the possibility of mantissa overflow by 1 digit. If so, the mantissa of the result is shifted right and exponent is incremented by 1.
- In case of exponent overflow, it is reported and operation is halted.

Phase 4 : Normalization

- Normalization is achieved by shifting mantissa left until the most significant digit is non zero.
- Each shift causes a decrement of the exponent and thus could cause an exponent underflow.

8.7.1.3 Hardware Implementation of Floating Point Addition/Subtraction

- The Fig. 8.7.2 shows the hardware implementation for the addition and subtraction of 32-bit floating point operands that have the single precision format, i.e. 1-bit for sign, 8-bits or signed exponent and 23-bits for mantissa.
- To find the difference (shift count n) between two exponents, exponents are subtracted using 8-bit subtractor.
- This difference, is sent to the SHIFTER unit. The sign of the difference that results from comparing exponents determines which mantissa is to be shifted. If the sign is 0, then $E_A \geq E_B$ and input to SWAP network is 0. This disables swapping and mantissa M_B is sent to the SHIFTER.
- If the sign is 1, the $E_A - E_B$ and input to SWAP network is 1. In this case swapping is enabled and mantissa M_A is sent to the SHIFTER.
- The SHIFTER unit shifts the given mantissa n positions to the right.
- The two way multiplexer is used to set the exponent of the result (E) equal to the larger exponent, based on the sign of the difference. The output of multiplexer is

$$E = E_A \quad \text{if} \quad E_A \geq E_B$$

or

$$E = E_B \qquad E_A < E_B$$

mantissa adder/subtractor, sign from the exponent comparison, signs of the operands and the operation to be performed.

- The result of the mantissa (M) is normalized. The normalized value is truncated to generate the 23-bit mantissa, MR, of the result. The leading zeros detector determines the number of bit shifts, X, to be applied to mantissa (M). The value X is then subtracted from the tentative resulted exponent E to generate the true result exponent, E_R.

8.7.2 Multiplication and Division

- In floating point arithmetic, multiplication and division are somewhat easier than addition and subtraction because an alignment of mantissas is not required in multiplication and division.

Rules of multiplication

1. If either of the is 0, report 0 as result.
2. Add the exponents and subtract bias. (127 in case of single precision numbers and 1023 in case of double precision numbers). The result may cause exponent overflow or underflow. If so report accordingly.
3. Multiply the mantissas and determine the sign of the result.
4. Normalize the result.

Fig. 8.7.3 shows the flowchart for floating point multiplication.

(See Fig. 8.7.3 on next page).

Rules of division

1. If the dividend is zero, report result as zero.
2. If the divider is zero, report result as ∞.
3. Subtract exponents and add bias (127 in case of single precision numbers and 1023 in case of double precision numbers).
4. Divide the mantissas and determine the sign of the result.
5. Normalize the result.

Fig. 8.7.4 shows the flowchart for floating point division. (See Fig. 8.7.4 on page 8-61)

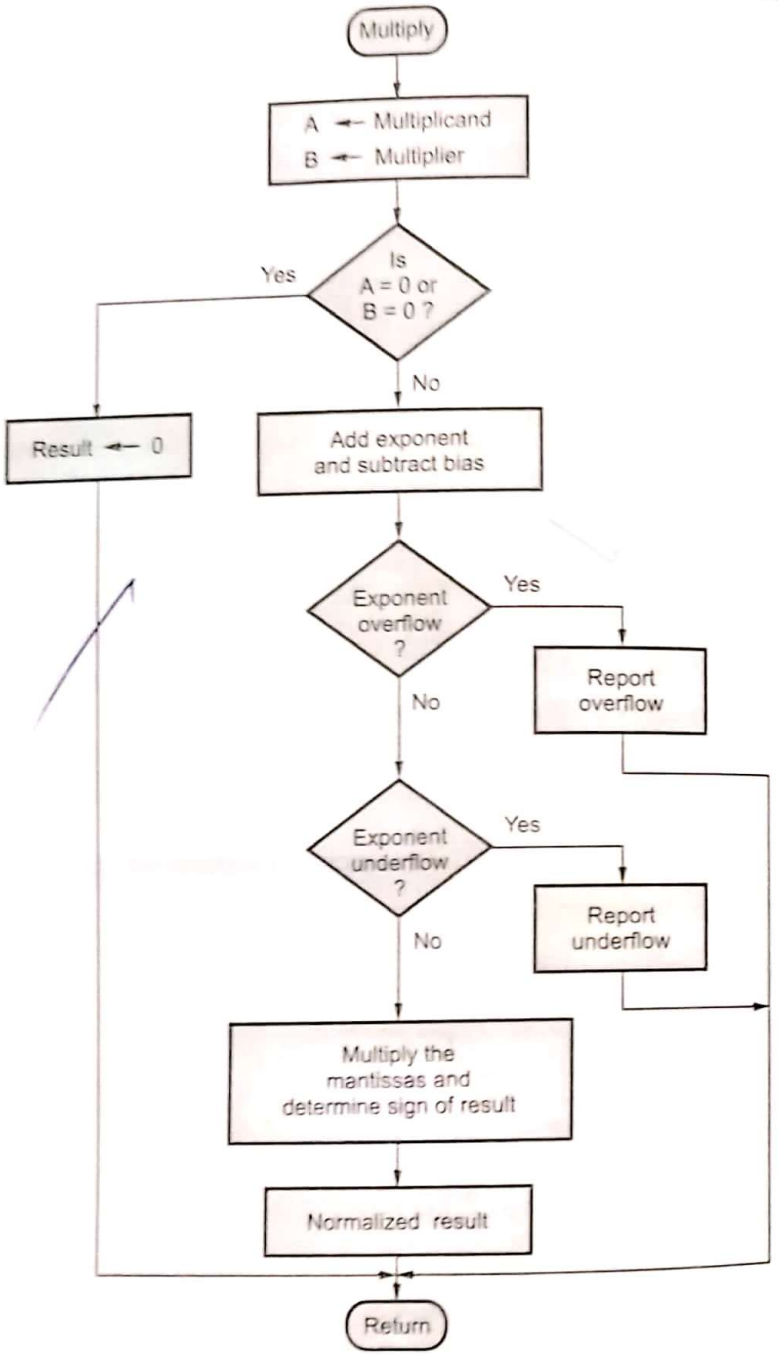


Fig. 8.7.3 Flowchart for floating point multiplication

Review Questions

1. *Define overflow and underflow in mantissa and exponent.*
2. *Draw the flowchart for floating point addition and explain.*
3. *Draw and explain the flowchart for floating point subtraction.*
4. *Draw and explain the hardware implementation of floating point adder/subtractor.*
5. *List the rules for floating point multiplication and division.*
6. *Explain floating point multiplication with the help of flowchart as well as algorithm.*
7. *With the help of flow chart explain floating point multiplication operation.*

8.8 Decimal Arithmetic Unit

GTU : May-12

The computer users are more comfortable with decimal numbers. They give input to computer in decimal numbers and expect result of the operations in the decimal form. However, arithmetic and logic unit of processor can perform operations with binary data. Therefore, it is necessary to convert the input in decimal number to binary, to perform all calculations with binary numbers, and to convert the results again into decimal. This may be convenient when data input/output is less compared to calculations performed on it. But when data input/output is large compared to calculations performed on it, it becomes convenient to do the internal arithmetic using decimal arithmetic unit. The decimal arithmetic unit accepts numbers in binary coded decimal (BCD) form and gives result also in the BCD form. In this section, we see how the basic BCD operations such as addition and subtraction can be implemented using BCD adder and BCD subtractor.

8.8.1 BCD Number

BCD is an abbreviation for binary coded decimal. BCD is a numeric code in which each digit of a decimal number is represented by a separate group of bits. The most common BCD code is 8-4-2-1 BCD, in which each decimal digit is represented by a 4-bit binary number. It is called 8-4-2-1 BCD because the weights associated with 4 bits are 8-4-2-1 from left to right. This means that, bit-3 has weight 8, bit-2 has weight 4 bit-1 has weight 2 and bit-0 has weight 1.

The Table 8.8.1 shows the 4-bit 8-4-2-1 BCD code used to represent a single decimal digit. The 8-4-2-1 BCD code is so widely used that it is common practice to refer to it simply as BCD.

Decimal Digit	BCD code 8	4	2	1
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1

Table 8.8.1 8-4-2-1 BCD code

In multi digit coding, each decimal digit is individually coded with 8-4-2-1 BCD cod For example, 58 in decimal can be encoded in 8-4-2-1 BCD as :

Decimal	5				8			
8-4-2-1 BCD	0	1	0	1	1	0	0	0

As seen from the above example, in multi digit coding of 8-4-2-1 BCD numbers require 4-bits per decimal digit. Therefore, total 8-bits are required to encode 58_{10} 8-4-2-1 BCD. When we represent the same number (58) in binary : 111010_2, we requi only 6 digits. This means that, for representing numbers, 8-4-2-1 BCD is less efficie than pure binary number system. The advantage of a BCD code is that it is easy convert between it and decimal. The principle disadvantage of a BCD, besides its lo efficiency, is that arithmetic operations are more complex than they are in pure binar Let us see the arithmetic operations using 8-4-2-1 BCD.

8.8.2 BCD Addition

The addition of two BCD numbers can be best understood by considering the thr cases that occur when two BCD digits are added.

Case 1 : Sum equals 9 or less with carry 0

Let us consider additions of 3 and 6 in BCD.

	1	1			Carry
+ 6	0	1	1	0	BCD for 6
+ 3	0	0	1	1	BCD for 3
9	0	1	0	0	1 BCD for 9
Carry					

Valid BCD numbers

	0	1	0	1	BCD for 5
+ 2	0	0	1	0	BCD for 2
7	0	0	1	1	1 BCD for 7
Carry					

The addition is carried out as in normal binary addition and the sums are 1 0 0 1 and 0 1 1 1 which are valid BCD codes.

Case 2 : Sum greater than 9 with carry 0

Let us consider addition of 6 and 8 in BCD.

+ 6		0	1	1	0 BCD for 6
+ 8		1	0	0	0 BCD for 3
14	0	1	1	1	0 Invalid BCD number (1 1 1 0) > 9
Carry					

The sum 1 1 1 0 is an invalid BCD number. This has occurred because the sum of the two digits exceeds 9. Whenever this occurs the sum has to be corrected by the addition of six (0110) in the invalid BCD number, as shown below.

+ 6		0	1	1	0 BCD for 6
+ 8		1	0	0	0 BCD for 8
14		1	1		Carry
		1	1	1	0 Invalid BCD number
	+	0	1	1	0 Add 6 for correction
Carry	1	0	1	0	0

0	0	0	1	0	1	0	0	BCD for 14
1				4				Decimal

After addition of 6 carry is produced into the second decimal position.

Case 3 : Sum equals 9 or less with carry 1

Let us consider addition of 8 and 9 in BCD.

+ 8	1				Carry
+ 9		1	0	0	0 BCD for 8
17		1	0	0	1 BCD for 9
	1	0	0	0	1
Carry					Incorrect BCD result

0	0	0	1	0	0	0	1	
1				1				

In this case, result (0001 0001) is valid BCD number, but it is incorrect. To get the correct BCD result correction factor of 6 has to be added to the least significant digit sum, as shown .

8		1	0	0	0 BCD for 8
+ 9		1	0	0	1 BCD for 9
17	0	0	0	1	0 0 0 1 Incorrect BCD result
+	0	0	0	0	0 1 1 0 Add 6 for correction
	0	0	0	1	0 1 1 1 BCD for 17
	1				7

Procedure of BCD addition

Going through these three cases of BCD addition we can summarize the BCD addition procedure as follows :

1. Add two BCD numbers using ordinary binary addition.
2. If four-bit sum is equal to or less than 9, no correction is needed. The sum is in proper BCD form.
3. If the four-bit sum is greater than 9 or if a carry is generated from the four-bit sum, the sum is invalid.
4. To correct the invalid sum, add 6 (0110_2) to the four-bit sum. If a carry results from this addition, add it to the next higher-order BCD digit.

Example 8.8.1 *Perform the code conversion : $(137)_{10}$ = (?) NBCD.*

Solution : NBCD = 8421 BCD

$\therefore \quad (137)_{10} = (0001\ \ 0011\ \ 0111)_{NBCD}$

Example 8.8.2 *Perform each of the following decimal additions in 8-4-2-1 BCD.*

a) $+\begin{matrix}24\\18\end{matrix}$ *b)* $+\begin{matrix}48\\58\end{matrix}$

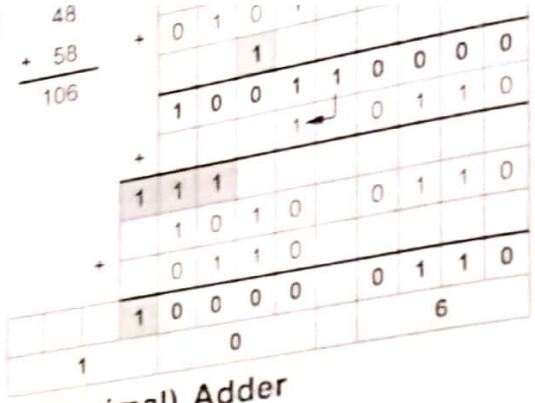
Solution :

a)

$$\begin{matrix}24\\+\ 18\\\hline 42\end{matrix}$$

	0	0	1	0		0	1	0	0
+	0	0	0	1		1	0	0	0
					1	1	1		Carry
	0	0	1	1		1	1	0	0 Invalid BCD number 1 1 0 0 > 9
				+	0	1	1	0	Add 6 for correction
	0	0	1	1	1	0	0	1	0
+				1					Propagate carry to next higher digit
	0	1	0	0		0	0	1	0 BCD for 42
	4					2			

b) 48
+ 58
106



8.8.3 BCD (Decimal) Adder

The digital systems handles the decimal number in the form of binary coded decimal numbers (BCD). A BCD adder is a circuit that adds two BCD digits and produces a sum digit also in BCD.

Here, we will see the implementation of BCD arithmetic.

To implement BCD adder we require :

- 4-bit binary adder for initial addition
- Logic circuit to detect sum greater than 9 and
- One more 4-bit adder to add 0110_2 in the sum if sum is greater than 9 or carry is 1.

The logic circuit to detect sum greater than 9 can be determined by simplifying the Boolean expression of given truth table.

Inputs				Output
S_3	S_2	S_1	S_0	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0			

As shown in the Fig. 8.8.2, the two BCD numbers, together with input carry, are first added in the top 4-bit binary adder to produce a binary sum. When the output carry is equal to zero (i.e. when sum ≤ 9 and C_{out} = 0) nothing (zero) is added to the binary sum. When it is equal to one (i.e. when sum > 9 or C_{out} = 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 can be ignored, since it supplies information already available at the output-carry terminal.

8.8.4 BCD Subtraction using 9's Complement Method

Subtraction of BCD numbers can be performed by representing negative numbers in the 9's complement form. The 9's complement can be used to perform subtraction by adding the minuend to the 9's complement of the subtrahend and adding the carry, if any.

The 9's complement of a decimal number is found by subtracting each digit in the number from 9. The 9's complement of each of the decimal digits is as follows :

9	9	9	9	9	9	9	9	9	9
0	1	2	3	4	5	6	7	8	9
9	8	7	6	5	4	3	2	1	0

Decimal digit
9's complement

Table 8.8.3 9's complement

Example 8.8.3 *Find the 9's complement of $(3497)_{BCD}$*

Solution :

	9	9	9	9
−	3	4	9	7
	6	**5**	**0**	**2**

$(3497)_{BCD}$
9's complement of $(3497)_{BCD}$

Steps for subtraction of BCD numbers using 9's complement method are as follows :

Step 1 : Find the 9's complement of a negative number.

Step 2 : Add two numbers using BCD addition (Minuend + 9's complement of subtrahend).

Step 3 : If carry is not generated result is negative and find the 9's complement of the result, otherwise result is positive and add carry to the result.

Example 8.8.4 *Perform $(46)_{10} - (22)_{10}$ in BCD using 9's complement.*

Solution : Step 1 : Find 9's complement of 22.

9's complement of 22 = (99 − 22) = 77

Step 2 : Add 46 and 9's complement of 22.

	1				**1**	**1**			Carry
	0	1	0	0	0	1	1	0	(46) BCD
+	0	1	1	1	0	1	1	1	(77) 9's complement of (22) BCD
	1	**0**	**1**	**1**	**1**	**1**	**0**	**1**	
	1	**1**	**1**	**1**	**1**	**1**			
	1	0	1	1	1	1	0	1	Invalid BCD numbers
+	0	1	1	0	0	1	1	0	Add 6 in each digit
	1	**0**	**0**	**1**	**0**	**0**	**0**	**1**	**1**
+								→	**1** Add end around carry
	0	**0**	**1**	**0**	**0**	**1**	**0**	**0**	
	2				**4**				Result

Fig. 8.8.3

Since there is a carry the result is positive and true.

∴ $(46)_{BCD} - (22)_{BCD} = (24)_{BCD}$

Example 8.8.5 *Perform $(24)_{10} - (56)_{10}$ in BCD using 9's complement.*

Solution : Step 1 : Find 9's complement of 56

9's complement of 56 = 99 − 56 = 43

Step 2 : Add $(24)_{10}$ and 9's complement of 56.

									Carry
	0	0	1	0	0	1	0	0	(24) BCD
+	0	1	0	0	0	0	1	1	9's complement of (56) BCD
Carry	**0**	**0**	**1**	**1**	**0**	**0**	**1**	**1**	**1** (67) BCD

Fig. 8.8.4

Since carry is 0 the answer is negative.

Step 3 : Take 9's complement of answer.

9's complement of 67 = 99 − 67 = 32

∴ $(24)_{10} - (56)_{10} = (-32)_{10}$

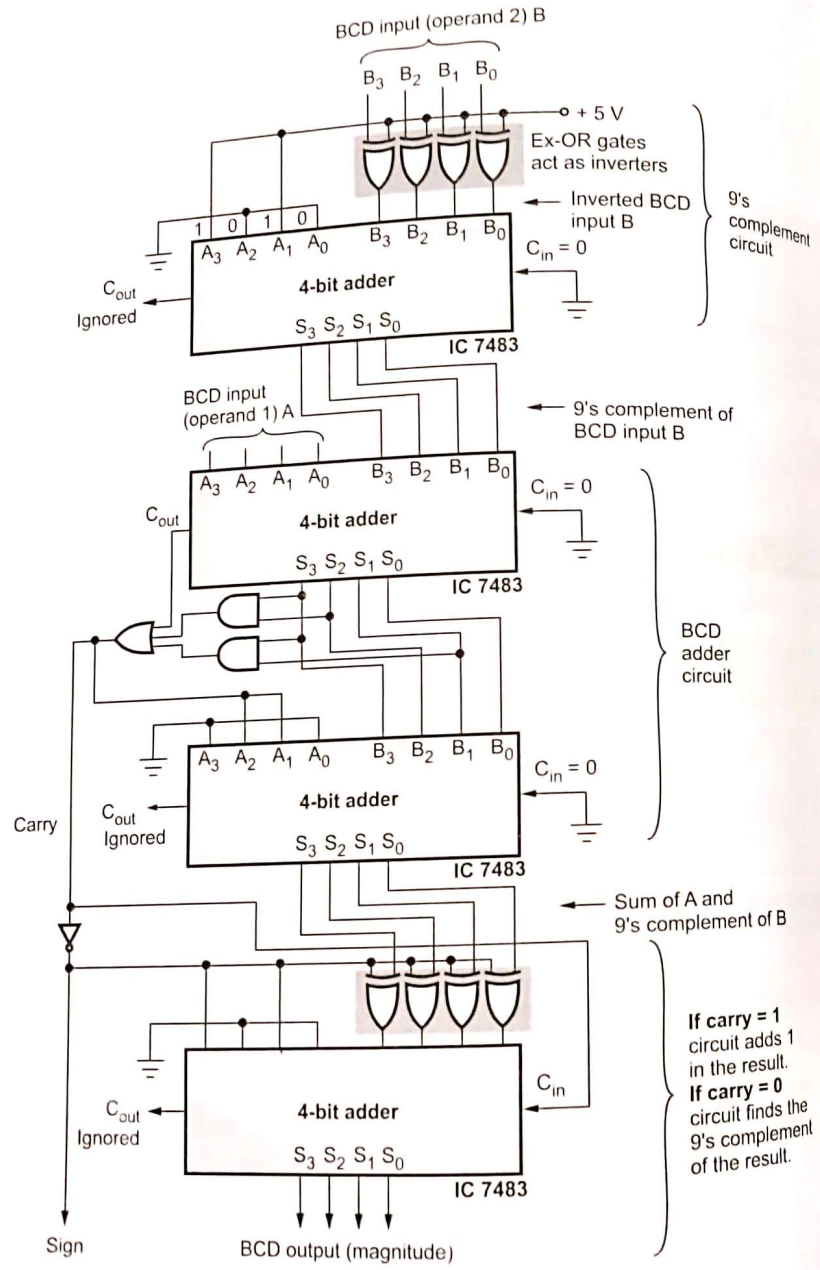


Fig. 8.8.5 4-bit BCD subtractor using 9's complement method

Fig. 8.8.5 shows the logic diagram of the circuit to implement above mentioned steps to perform BCD subtraction using 9's complement method. As shown in the Fig. 8.8.5, first binary adder finds the 9's complement of the negative number. It does this by inverting each bit of BCD number and adding 10 $(1010)_2$ to it. Let us find the 9's complement of 2.

	0	0	1	0	← BCD for 2
	1	1	0	1	← Inverting each bit
+	1	0	1	0	← Add 10 $(1010)_2$
Ignore carry → 1	0	1	1	1	← 9's complement for 2

Next two 4-bit binary adders perform the BCD addition. The last adder finds the 9's complement of the result if carry is not generated after BCD addition otherwise it adds carry in the result.

8.8.5 BCD Subtraction using 10's Complement Method

Subtraction of BCD numbers can be performed by representing negative numbers in the 10's complement form. The 10's complement of a decimal number is equal to the **9's complement plus 1**. The 10's complement can be used to perform subtraction by adding the minuend to the 10's complement of the subtrahend and dropping the carry.

The 9's complement of a decimal number is found by subtracting each digit in the number from 9. The 9's complement of each of the decimal digits is as follows :

9	9	9	9	9	9	9	9	9	9	
0	1	2	3	4	5	6	7	8	9	**Decimal digit**
9	8	7	6	5	4	3	2	1	0	**9's complement**

Table 8.8.4 9's complement

Example 8.8.6 *Find the 10's complement of* $(3497)_{BCD}$.

Solution : Steps for subtraction of BCD numbers using 10's complement method are as follows :

	9	9	9	9	
−	3	4	9	7	$(3497)_{BCD}$
	6	5	0	2	9's complement of $(3497)_{BCD}$
+				1	Add 1
	6	5	0	3	10's complement of $(3497)_{BCD}$

Step 1 : Find the 10's complement of a negative number.

Step 2 : Add two numbers using BCD addition (Minuend + 10's complement of subtrahend).

Step 3 : If carry is not generated result is negative and find the 10's complement of the result, otherwise result is positive and discard carry.

Example 8.8.7 *Perform $(46)_{10} - (22)_{10}$ in BCD using 10's complement.*

Solution : Step 1 : Find 10's complement of 22.

10's complement of 22 = 9's complement of 22 + 1 = (99 − 22) + 1 = 78

Step 2 : Add 46 and 10's complement of 22.

			1						Carry
	+ 46	+	0	1	0	0	0	1	1	0	$(46)_{BCD}$
	− 22		0	1	1	1	1	0	0	0	(78)10's complement of $(22)_{BCD}$
	24		1	0	1	1	1	1	1	0	
			1	1	1	1	1	1			Carry
			1	0	1	1	1	1	1	0	Invalid BCD numbers
		+	0	1	1	0	0	1	1	0	Add 6 in each digit
Discard carry			X	0	0	1	0	0	1	0	0	
				2				4			Result

Since there is a carry the result is positive and true.

∴ $(46)_{BCD} - (22)_{BCD} = \mathbf{(24)_{BCD}}$

Example 8.8.8 *Perform $(24)_{10} - (56)_{10}$ in BCD using 10's complement.*

Solution : Step 1 : Find 10's complement of 56.

10's complement of 56 = 9's complement of 56 + 1

= (99 − 56) + 1

= 44

Step 2 : Add $(24)_{10}$ and 10's complement of 56.

					1				Carry
24		0	0	1	0	0	1	0	0	$(24)_{BCD}$
− 56	+	0	1	0	0	0	1	0	0	10's complement of $(56)_{BCD}$
− 32	Carry	0	0	1	1	0	1	0	0	0	$(68)_{BCD}$

Since carry is 0 the answer is negative.

Step 3 : Take 10's complement of answer.

10's complement of 68 = 9's complement of 68 + 1

= (99 − 68) + 1 = 32

∴ $(24)_{10} - (56)_{10} = \mathbf{-(32)_{10}}$

Fig. 8.8.6 shows the logic diagram of the circuit to implement above mentioned steps to perform BCD subtraction using 10's complement method. As shown in the Fig. 8.8.6, first binary adder finds the 10's complement of the negative number (9's complement + 1). Next two 4-bit binary adders perform the BCD addition. Finally, last 4-bit binary adder finds the 10's complement of the number if carry is not generated after BCD addition.

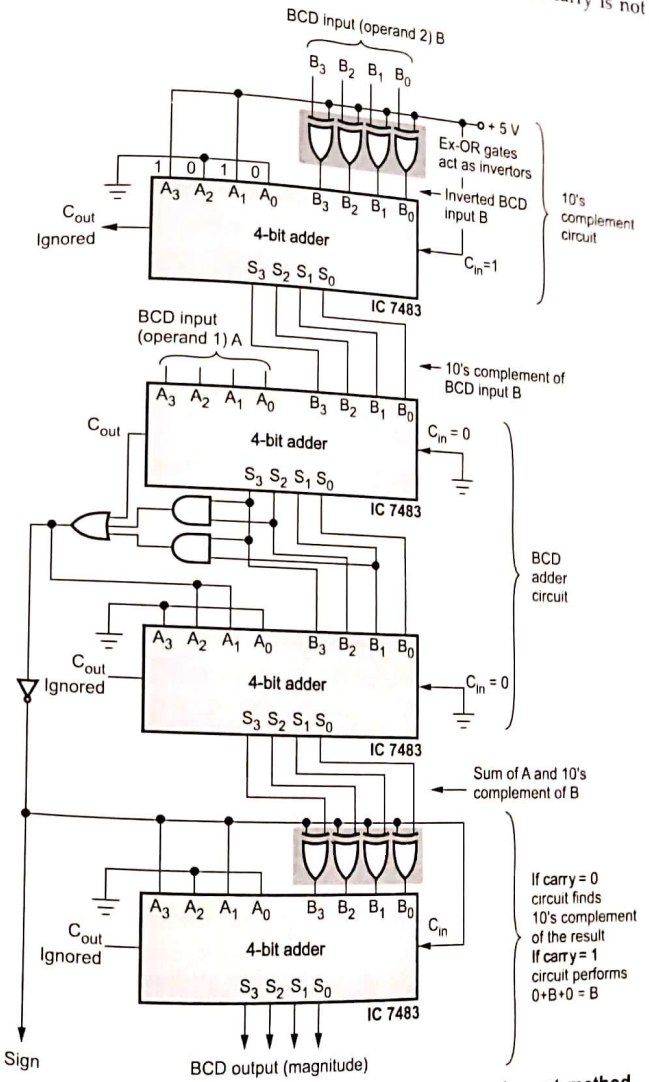


Fig. 8.8.6 4-bit BCD subtractor using 10's complement method

Review Questions

1. *Explain the BCD adder and BCD subtractor.*
2. *Explain the BCD adder and its working with block diagram.* **GTU : May-12, Marks 7**

8.9 Decimal Arithmetic Operations

In section 8.8, we have seen the addition and subtraction of BCD numbers and the implementation of it with combinational circuits. In this section, we see the BCD multiplication, BCD division and floating point operations.

8.9.1 BCD Multiplication

BCD Multiplication is similar to binary multiplication except for following points :

- A decimal multiplier has digits that range in value from 0 to 9, whereas a binary multiplier has only 0 and 1 digits.
- In binary multiplication, the multiplicand is added to the partial product if the multiplier bit is 1. On the other hand, in BCD multiplication, the multiplicand must be multiplied by the digit multiplier. The multiplication operation can be accomplished by adding the multiplicand to the partial product a number of times equal to the value of the multiplier digit.

Let us see the implementation of BCD multiplication. Assume that multiplier and multiplicands are four digit numbers. The Fig. 8.9.1 shows the typical hardware for the decimal multiplication. It consists of three registers, A, B and Q each having a corresponding sign flip-flop S_A, S_B and S_Q. The registers A and B are 20-bit so they can accommodate 5 BCD digits whereas register Q is 16-bit which can accommodate 4 BCD digits. Out of five digits of register B, four least significant digits (LSDs) are used to store magnitude of multiplicand. The 4-digit magnitude of multiplier is loaded in register Q. Sign flip-flops of register B and register Q are used to store sign of multiplicand and sign of the multiplier, respectively. The 5-digit BCD adder adds the five digits in parallel and places the sum in the 5 digit register A. If carry occurs after addition it is loaded in CY flip-flop. The least significant digit of Q register can be incremented or decremented by activating clock signal.

Multiplication algorithm

1. Initially, load multiplicand in B and multiplier in Q.
2. A = 0, E_B = 0 and iteration counter (IC) = number of digits in the multiplier. Sign of result = sign of multiplicand EX-OR sign of multiplier i.e.

   $S_A \leftarrow S_B \oplus S_Q$

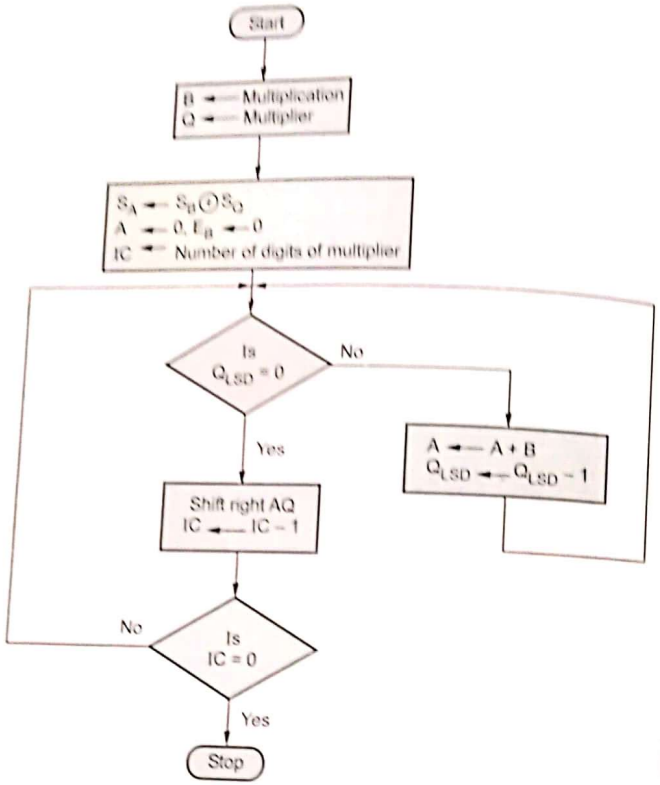


Fig. 8.9.2 Flowchart for BCD multiplication

Division algorithm

1. Initially, load divisor in B and dividend in Q.
2. Sign of Quotient = Sign of dividend EX-OR sign of divisor i.e., $S_Q \leftarrow S_A \oplus S_B$
   Iteration counter (IC) = Number of digits in the dividend
   $E_B \leftarrow 0$, $E_A \leftarrow 0$, $A \leftarrow 0$
3. Shift register AQ to the left by one digit and load empty Q_{LSD} equal to zero.
4. Subtract B from A($A \leftarrow A + \overline{B} + 1$) using 10's complement of B.

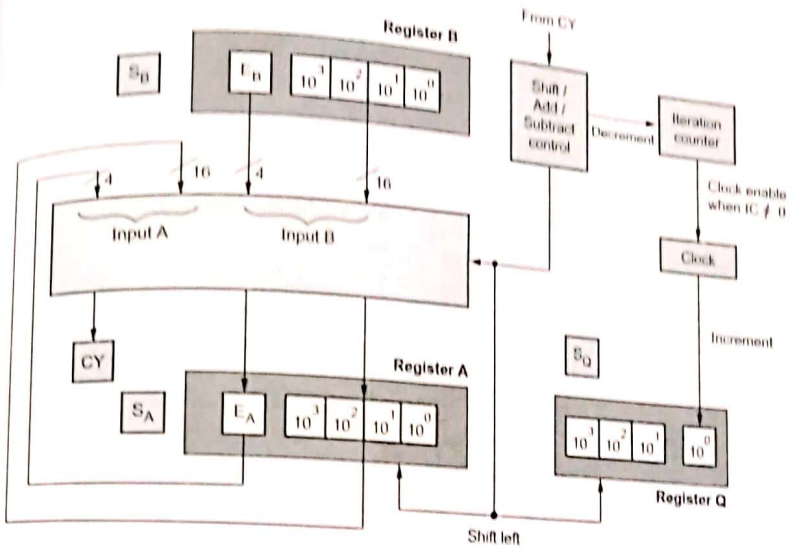


Fig. 8.9.3 Hardware for decimal division

5. If CY = 1 : It signifies that A ≥ B. In this case, increment quotient digit in Q_{LSD} once and subtract divisor again, i.e., repeat step 4.

   If CY = 0 : It signifies that A < B. In this case, add divisor in register A to restore the partial remainder.

6. Decrement and check iteration count (IC), if IC ≠ 0, repeat steps 3, 4 and 5; otherwise stop.

The Fig. 8.9.4 shows the flowchart for decimal division. (See Fig. 8.9.4 on next page).

Flowchart

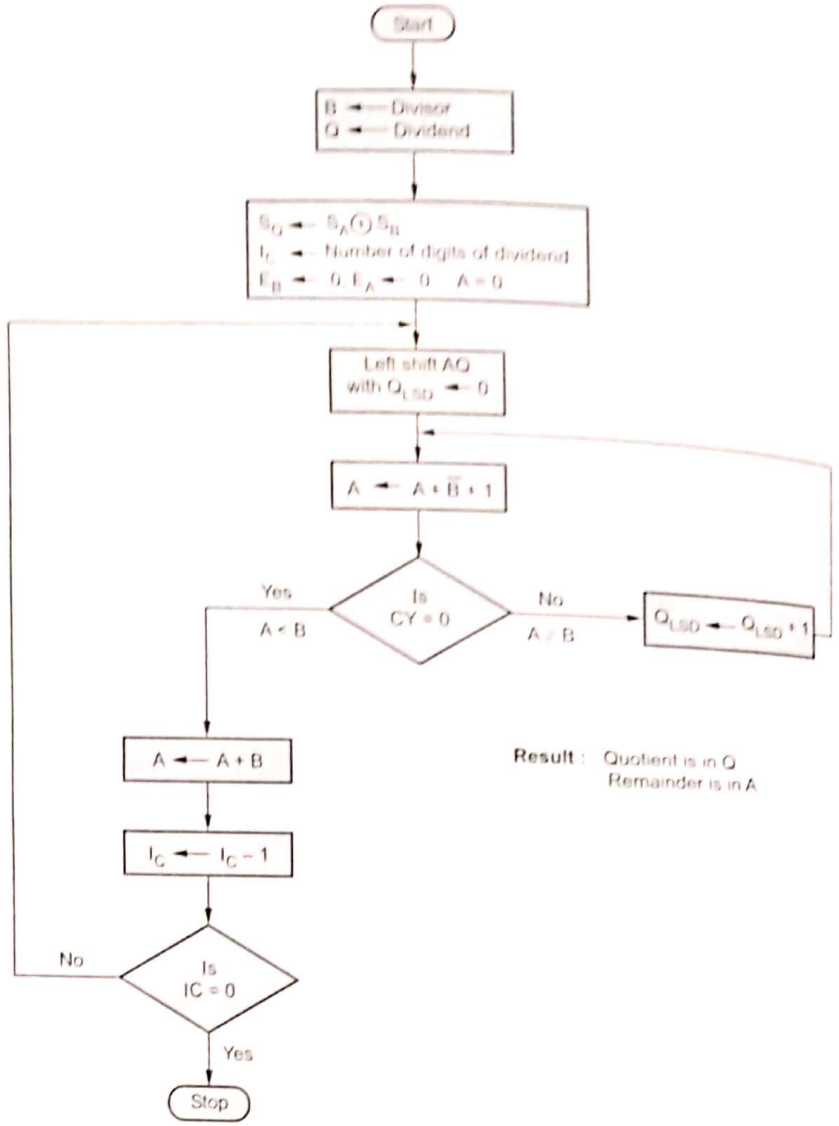


Fig. 8.9.4

Example 8.9.1 *Show the steps in 5288 ÷ 40.*

Solution : Divisor ← 40 Dividend ← 5288

10's complement of divisor = 99960

Review Questions

1. *Explain the algorithm for BCD multiplication.*
2. *Draw the flowchart for BCD multiplication.*
3. *Explain the algorithm for BCD division.*
4. *Draw the flowchart for BCD division.*

□□□

9 Input - Output Organization

Syllabus

Input-Output interface, Asynchronous data transfer, Modes of transfer, Priority interrupt, DMA, Input-Output Processor (IOP), CPU/IOP communication, Serial communication.

Contents

9.1 Input/Output Interface
9.2 I/O Bus and Interface Modules
9.3 Asynchronous Data Transfer
9.4 Asynchronous Serial Transfer
9.5 Asynchronous Communication Interface
9.6 Modes of Transfer
9.7 Serial Communication

9.1 Input/Output Interface

- **Input-output interface** enables transfer of data between internal storage and external I/O devices.
- In order to interface peripherals with the CPU, special communication links are required. Due to this communication link, the differences between the CPU and peripherals such as data transfer speed, mode of operation are overcome.

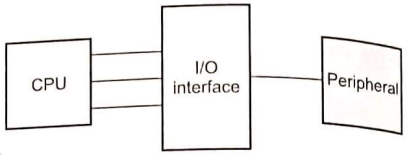


Fig. 9.1.1 I/O interface

- The differences between the CPU and peripherals are :

Sr. No.	Peripherals	CPU
1.	These are electro-mechanical and electromagnetic devices.	It is an electronic device.
2.	Data transfer rate is slower than that of the CPU.	Data transfer rate is faster than that of peripherals.
3.	Data is in form of codes.	Data is in word format.

- As peripherals are electromagnetic and electromechanical devices and CPU is an electronic device, signal conversion becomes necessary whenever required.
- Due to difference in data transfer rate, synchronization of peripheral with CPU may be required.
- The operating modes of all the peripherals connected to the CPU are different. Thus, each of the peripherals connected to the CPU should be controlled so that the operation of other peripherals continues without any disturbance.
- **Interface units** resolve all the above problems that occur while data transfer between peripherals and CPU.
- Interface unit is between the processor bus and the peripheral device and it supervises and synchronizes all input and output transfers.
- Each peripheral may have its own controller that supervises and controls operations of its particular mechanism.

Review Questions

1. *What is the need of I/O interface ?*
2. *Give difference between peripherals and CPU.*

9.2 I/O Bus and Interface Modules

The I/O interface is nothing but the hardware required to connect an I/O device to the bus. It is also called **I/O system**. The major requirements of an I/O interface are :

1. Control and timing
2. Processor communication
3. Device communication
4. Data buffering
5. Error detection

All these requirements are explained here. The I/O interface includes a control and timing requirements to coordinate the flow of traffic between internal resources (memory, system bus) and external devices. Processor communication involves different types of signal transfers such as

- Processor sends commands to the I/O system which are generally the control signals on the control bus.
- Exchange of data between the processor and the I/O interface over the data bus.
- The data transfer rate of peripherals is often much slower than that of the processor. So it is necessary to check whether peripheral is ready or not for data transfer. If not, processor must wait. So it is important to know the status of I/O interface. The status signals such as BUSY, READY can be used for this purpose.
- A number of peripheral devices may be connected to the I/O interface. The I/O interface controls the communication of each peripheral with processor. So it must recognize one unique address for each peripheral connected to it.

The I/O interface must able to perform device communication which involves commands, status information and data.

Data buffering is also an essential task of an I/O interface. Data transfer rates of peripheral devices are quite high than that of processor and memory. The data coming from memory or processor are sent to an I/O interface, buffered and then sent to the peripheral device at its data rate. Also, data are buffered in I/O interface so as not to tie up

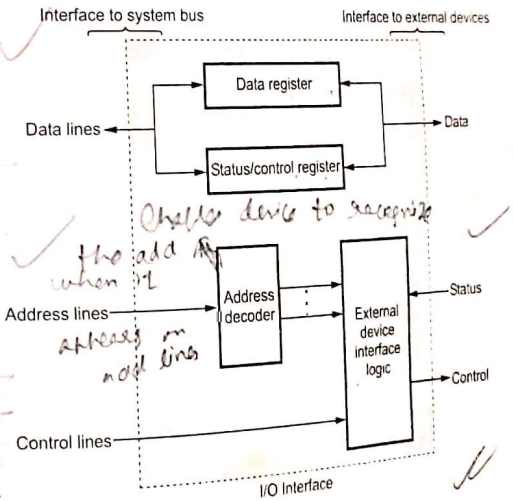
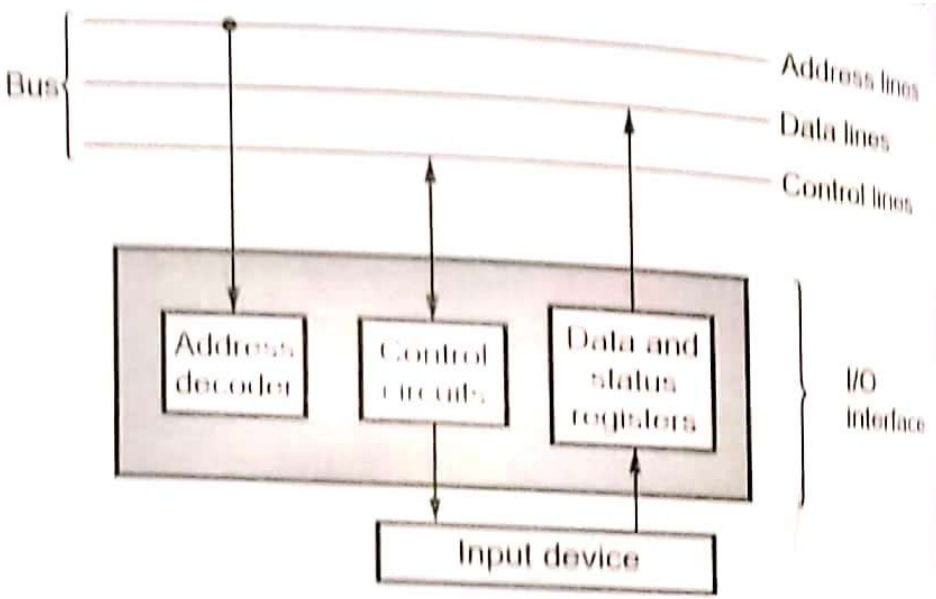


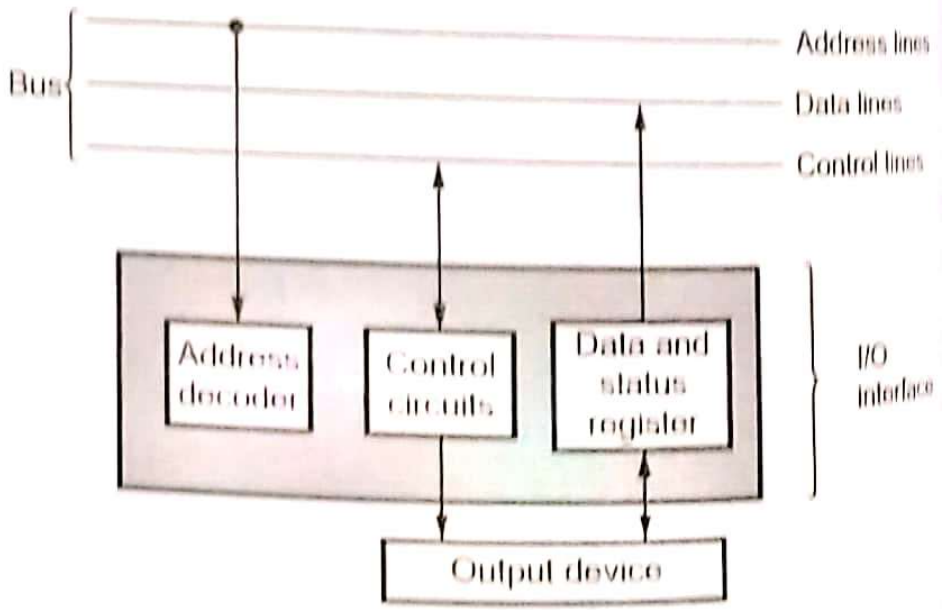
Fig. 9.2.1 Block diagram of I/O interface

and status signal from the I/O device and generates control signals to control the direction and speed of data transfer between processor and I/O devices.

The Fig. 9.2.2 shows the I/O interface for input device and output device. Here, for simplicity block schematic of I/O interface is shown instead of detail connections. The address decoder enables the device when its address appears on the address lines. The data register holds the data being transferred to or from the processor. The status register contains information relevant to the operation of the I/O device. Both the data and status registers are assigned with unique addresses and they are connected to the data bus.



(a) I/O interface for input device



(b) I/O interface for output device

Fig. 9.2.2

becomes an I/

the address of an I/O port. The usual memory related instructions are used for I/O related operations. The special I/O instructions are not required. Fig. 9.2.5 gives the I/O with memory mapped I/O.

Fig. 9.2.4 Address space

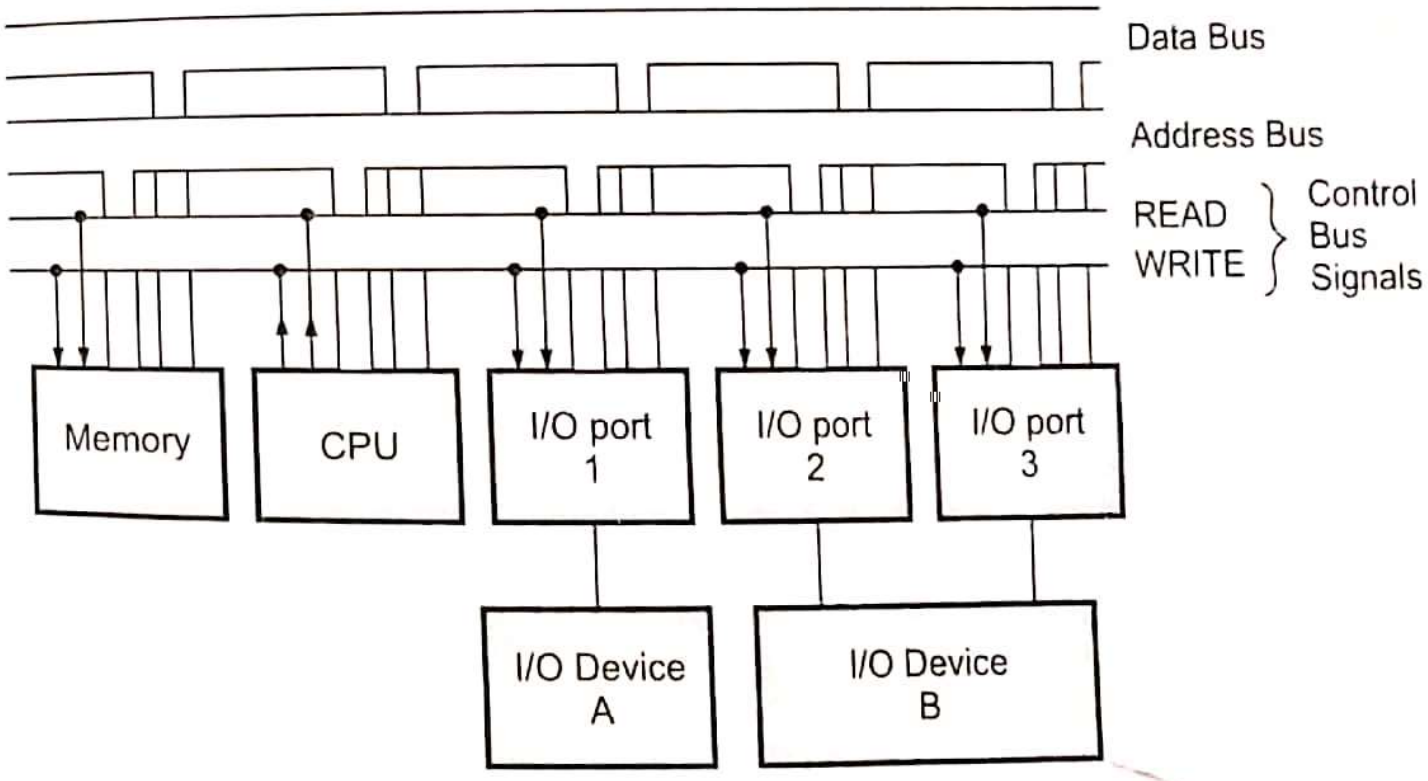


Fig. 9.2.5 Programmed I/O with memory mapped I/O

9.2.2 I/O Mapped I/O

If we do not want to reduce the memory address space, we allot a different I/O address space, apart from total memory space which is called I/O mapped I/O technique as shown in Fig. 9.2.6.

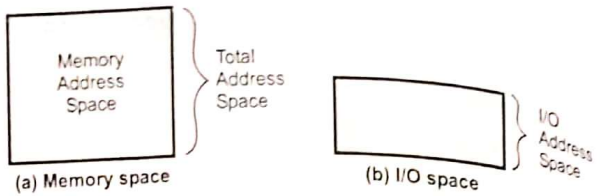


Fig. 9.2.6 Address space

Here, the advantage is that the full memory address space is available. But now, the memory related instructions do not work. Therefore, processor can only use this mode if it has special instructions for I/O related operations such as I/O read, I/O write.

Fig. 9.2.7 gives I/O with I/O mapped I/O technique.

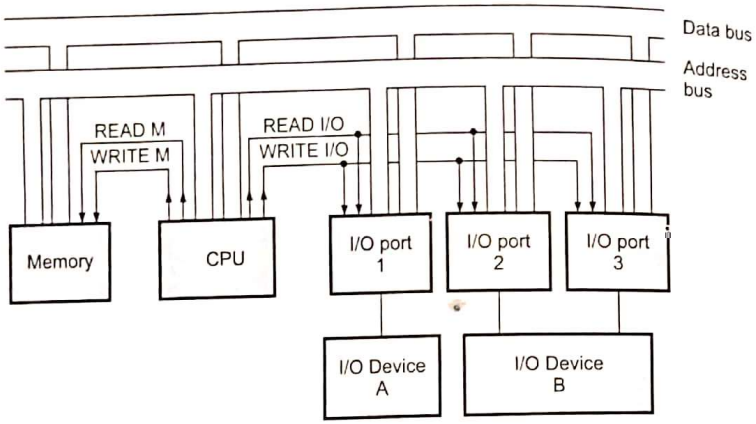


Fig. 9.2.7 Programmed I/O with I/O mapped I/O

9.2.3 I/O Versus Memory Bus

- As mentioned earlier processor communicates with I/O as well as memory.
- Memory bus has data lines, address lines and read/write control lines.

Three ways to communicate processor buses with memory and I/O

1. Two different buses are used, one for memory and the other for I/O.
2. One common bus is used for memory and I/O, but both (memory and I/O) have separate control lines.
3. Memory and I/O having one common bus and common control lines.

9.2.2.5 Example of I/O Interface

- Fig. 9.2.9 shows an example of I/O interface unit, that consist of bus buffers, timing and control circuit, status register, control register, ports (data registers).

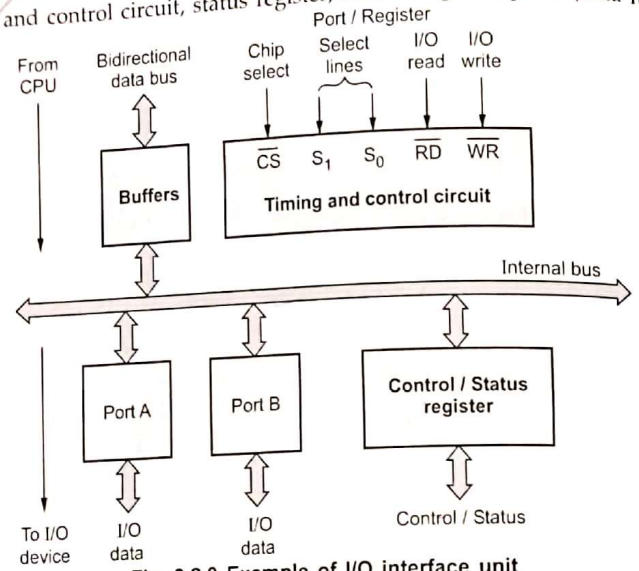


Fig. 9.2.9 Example of I/O interface unit

$\overline{CS}$	$\overline{RD}$	$\overline{WR}$	S_1	S_0	Port/Register selected
1	X	X	X	X	None
0	0	1	0	0	Port A read
0	1	0	0	0	Port A write
0	0	1	0	1	Port B read
0	1	0	0	1	Port B write
0	0	1	1	X	Control/status register read
0	1	0	1	X	Control/status register write

Table 9.2.1

Note
1. If $\overline{CS}$ = 0, S_1 S_0 = 10, $\overline{RD}$ = 0, $\overline{WR}$ = 1, then data is read from control register
2. If $\overline{CS}$ = 0, S_1 S_0 = 1 1, $\overline{RD}$ = 1, $\overline{WR}$ = 0, then data is written into status register.

- The communication link between the interface and the CPU is the data bus. The address assigned to the interface is determined by the chip select and port/register select signal. The I/O read and I/O write are the two control lines that specify the input or output of the interface. The four registers, viz port A, port B, control register, status register and the I/O devices connected to the interface can communicate directly.
- The transfer of data to and from the device is transferred to either port A or port B. The interface may operate with an output device or input device or both input and output device, it depends on the device to which the interface is connected. In case of magnetic disk where the device acts as input as well as output device, bidirectional lines are used. A word is sent to the appropriate interface device so as to pass the command to the I/O device.
- The function code is not required. There is a common bus for transfer of data, control and status information. The interface register that communicates with the CPU, distinguishes between the data, control or status information
- The control register receives the control information sent by CPU.
- The interface and I/O device that are attached to the CPU can vary their operating mode by loading the appropriate bits into the control register. For example port, A can be configured as an input port while port B can be configured as an output port.
- **Status register** bits determine the status and errors that occur during the data transfer. For example, a bit in status register may indicate error in the parity of data and a bit may indicate that port B is ready to receive new data from the I/O device connected.
- A **bidirectional data bus** is used for communication between the CPU and the interface register. The chip select $(\overline{CS})$ and two register select (S_0, S_1) signals are used to select the interface using address bus. Generally an address decoder must be provided externally so as to determine the address assigned to the interface register.
- The **chip select** $(\overline{CS})$ signal is enabled i.e. $\overline{CS}$ = 0 after selection of the interface by the address bus. S_1 and S_0 are connected to the least significant lines of the address bus.
- One of the four register (Viz. port A, port B, control and status registers) is selected by varying the values of S_0 and S_1 bit as shown in Fig. 9.2.9.
- When **I/O read signal** is enabled, the data from the selected register is transformed into CPU via the data bus.

- When **I/O write signal** is enable, the data from CPU is transferred into the selected register via data bus.
- $\overline{CS}$, $\overline{RD}$, $\overline{WR}$ signals are active low signals.

Review Questions

1. *Give the major requirements for an I/O module.*
2. *What are the signal transfers required during CPU communication ?*
3. *Draw and explain the block diagram of I/O interface.*
4. *Explain the I/O interface for input device.*
5. *Explain the I/O interface for output device.*
6. *Write a note on I/O bus and interface modules.*
7. *Explain in brief I/O bus connection with I/O devices.*
8. *What are the three types of commands used in I/O bus and I/O devices interfacing ?*
9. *Describe I/O versus memory bus in detail.*
10. *Compare isolated and memory mapped I/O.*
11. *Explain example of I/O interface with a neat labeled diagram.*

9.3 Asynchronous Data Transfer

The internal operations of digital systems are synchronized by means of clock signal. The CPU and an I/O interface are two independent digital systems. If they both share common clock between them, the data transfer between them is said to be **synchronous**. In most cases, the internal timing in one digital system is independent from the other, in that each uses its own private clock. In such cases, the data transfer between them is **asynchronous** in nature.

In **asynchronous** data transfer instead of common clock, control signals are used to assist the data transfer. Two methods are used to transfer data asynchronously between two independent units :

- Strobe
- Handshake

9.3.1 Strobe Control

In the strobe control asynchronous data transfer, a single control line (strobe) is used to indicate to the other unit when the transfer has to occur. The strobe signal may be activated by either the source or the destination unit. The Fig. 9.3.1 shows a source-initiated data transfer. In this the strobe signal informs the destination unit that the valid data is available on the data bus from the source unit.

unit then disables its data valid signal, which invalidates the data on the bus. The destination unit then disables its data accepted signal and the system goes into its initial stage.

The Fig. 9.3.4 shows the data transfer using handshaking signals when initiated by destination unit. Here, the destination unit activates **ready for data** signal when it is ready to accept data from source unit. In response to that, the source unit places the data on the bus and initiates the **data valid** signal. The destination unit then accepts the data from the data bus and disables the **ready for data** signal. Then, the source unit disables **data valid** signal and invalidates data on the bus. As a result, the system goes into its initial stage.

(See Fig. 9.3.4 on next page)

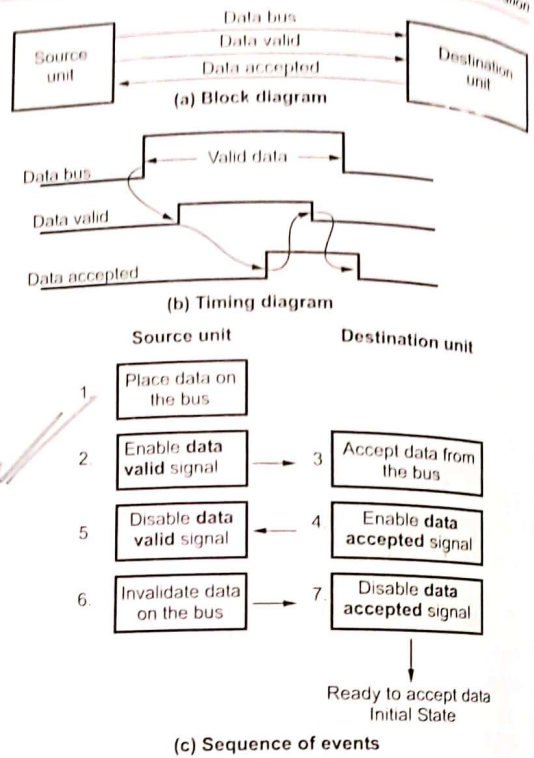


Fig. 9.3.3 Source initiated data transfer using handshaking

Advantages of handshaking scheme

1. The handshaking scheme provides high degree of flexibility and reliability because the successful completion of a data transfer relies on active participation by both units.
2. The scheme can use **timeout** signal to detect any problem in the completion of data transfer. For example, if one unit enables its handshaking control signal and if the return handshaking signal does not respond within a given time period, the unit detects an error and activates **timeout** signal to interrupt processor. In response, processor executes a service routine that takes appropriate error recovery action.

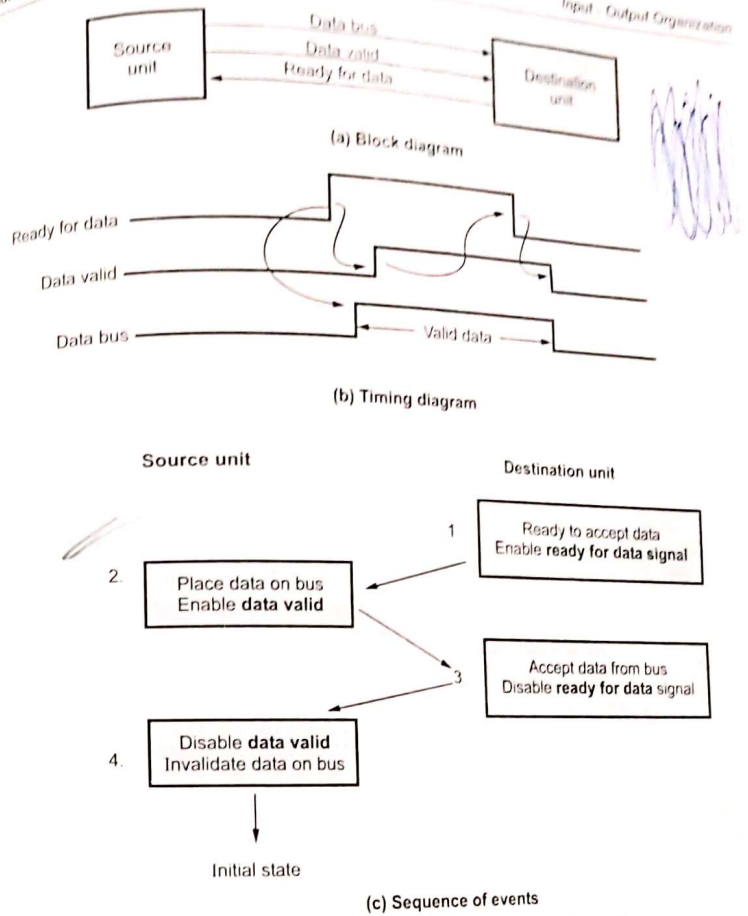


Fig. 9.3.4 Destination initiated data transfer using handshaking

Review Questions

1. *Explain the asynchronous transmission with the help of timing diagrams.*
2. *Explain the asynchronous serial transmission.*
3. *Draw and explain the block diagram of asynchronous serial interface.*

9.4 Asynchronous Serial Transfer

Most of the microprocessors are designed for parallel communication. In parallel communication number of lines required to transfer data depend on the number of bits

Parallel data transfer
It can transmit more than one data bit at a time.
Faster data transfer rate.
Needs more number of wires to connect devices in the system.
The interconnection penalty increases as distances increase. Thus not suitable for long distance communication.

rial and parallel data transfer

on

for asynchronous serial transmission. In this, the bits of a character or data an come at any rate (asynchronously) as ers are being sent, a line stays high at beginning of a character is indicated by ynchronize the transmitter and receiver ast significant bit first, followed by one indicate the end of character. Different ition of start bit, character and stop bits arry no information, but are required 9.4.2 illustrates how the data byte CAH s serial format.

5.	Start and stop bits are required to establish communication of each character.	Start and stop bits are not required to establish communication of each character, however, synchronization bits are required to transfer the data block.
6.	Used in low-speed transmissions at about speed less than 20 kbits/sec.	Used in high-speed transmissions.

Table 9.4.3 Comparison between asynchronous and synchronous serial data transfer

Example 9.4.1 *How many characters per second can be transmitted over a 4800 baud line in each of the following modes ? (Assume character code of 8 bits)*

i) Synchronous serial transmission.

ii) Asynchronous serial transmission with two stop bits.

iii) Asynchronous serial transmission with one stop bit.

Solution :

i) For synchronous transmission

$$\text{Number of characters transmitted per sec} = \frac{\text{Baud rate}}{\text{Character length}} = \frac{4800}{8} = \mathbf{600\ characters/sec}$$

ii) For asynchronous with two stop bits

$$\text{Number of characters transmitted per sec} = \frac{\text{Baud rate}}{\text{Frame length}}$$

$$= \frac{\text{Baud rate}}{1\text{ start bit} + 8\text{-bit character length} + 2\text{ stop bits}}$$

$$= \frac{4800}{11} = 436.36 \approx \mathbf{436\ characters/sec}$$

iii) For asynchronous with one stop bit

$$\text{Number of characters transmitted per sec} = \frac{\text{Baud rate}}{\text{Frame length}}$$

$$= \frac{\text{Baud rate}}{1\text{ start bit} + 8\text{-bit character length} + 1\text{ stop bits}}$$

$$= \frac{4800}{10} = \mathbf{480\ characters/second}$$

Review Questions

1. *Compare parallel and serial type of data transfer.*
2. *Explain asynchronous serial communication in detail.*

9.5 Asynchronous Communication Interface

To implement serial communication in microprocessor system we need basically two devices :

1) Parallel to serial converter 2) Serial to parallel converter.

To transmit byte data it is necessary to convert byte into eight serial bits. This can be done by using the parallel to serial converter. Similarly at the reception these serial bits must be converted into parallel 8-bit data. The serial to parallel converter is used to convert serial data bits into the parallel data.

The devices designed for this purpose are called Universal Asynchronous Receiver-Transmitter (UART). The devices which provides synchronous as well as asynchronous transmission and reception are called as universal synchronous asynchronous receiver-transmitter. A good example of UART is 8250 and USART is 8251.

A Fig. 9.5.1 shows the block diagram of a typical asynchronous communication interface. It consists of data bus buffer, Read/write control logic, modem control, transmit register, transmit control, receiver register, receiver control and shift registers.

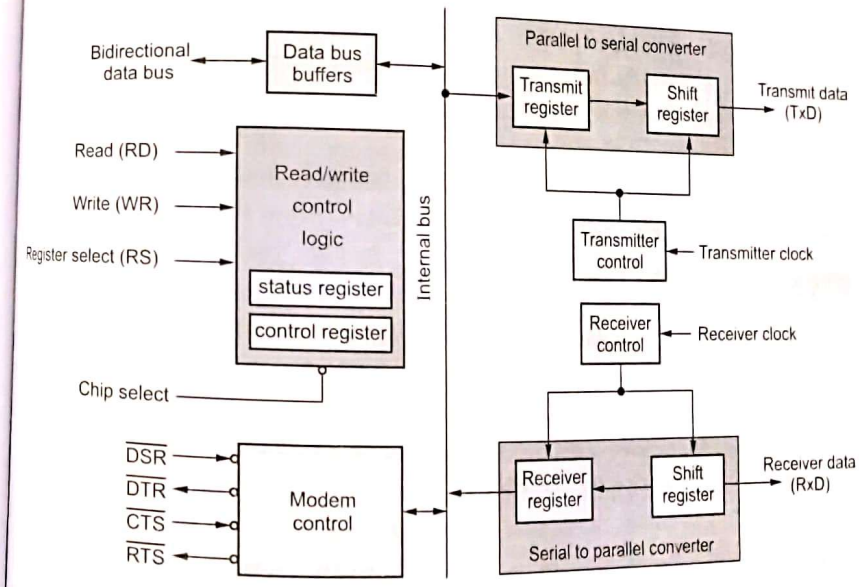


Fig. 9.5.1 Typical asynchronous serial communication interface

Data bus buffer : This tri-state, bi-directional buffer is used to connect the serial interface to the system data bus. Along with the data, control word, command words and the status information is transferred through this buffer.

Read/write control logic : This functional block accepts inputs from the system control bus and generates control signals for overall serial interface operation. It contains control word register that stores the control format for the device functional definition. It also contains the status register which gives the status of serial interface.

Transmit register and shift register : They accept parallel data from the CPU, adds the appropriate framing information, serialize it and transmit it on the TxD pin on the falling edge of transmitter clock.

Receiver register and shift register : They accept serial data on the RxD line, convert this serial data to parallel format, check for bits or characters and sends assembled characters to the CPU.

Transmit control : It manages all activities associated with the transmission of serial data. It accepts and issues signals both externally and internally to accomplish this function.

Receiver control : It manages all receiver-related activities. Along with data reception, it does false start bit detection, parity error detection, framing error detection, sync detection and break detection.

Modem control : Usually, the serial interface is supported with modem control signals. The modem control provides control circuitry for the generation of $\overline{RTS}$ and $\overline{DTR}$ and the reception of $\overline{CTS}$ and $\overline{DSR}$.

9.5.1 Transmitter Operation

- Initially, CPU checks the flag and status register so as to confirm whether the transmitter register is empty or not.
- If the transmitter register is found to be empty, then the CPU transfers data (character) to transmitter register and the interface clears the flag so as to indicate that the transmitter register is full.
- The first bit of transmitter register is the **start bit** which is set to 0.
- The transmission between the transmitter register and the shift register is in parallel form with enough number of stop bits appended into the shift register.
- After transmitting the data from transmitter register to the shift register, the transmitter register is declared empty.

- The data (character) in the shift register is transmitted by serial shifting i.e. one bit at a time, according to the baud rate specified.
- The CPU again transfers next data (character) to the transmitter register as it finds the transmitter register empty by checking the flag in the status register.
- The interface can load new data (character) as soon as the previous data (character) is transmitted and hence is known as **double buffered.**

9.5.2 Receiver Operation

- The receiver operation is exact opposite to that of the transmitter operation.
- The receiver control checks for 0 signal from the received data which means that the start bit is detected.
- As the start bit is detected, the further received data is shifted into shift register at the specified baud rate.
- The interface checks the parity and stop bits of the received data and then transfers the data (character) in parallel without start and stop bit from the shift register to the receiver register.
- The flag in the status register signifies whether the receiver register is full or empty. After the data transfer into the receiver register, the flag in the status register is set.
- The CPU checks the flag register and if it is set then the data from the receiver register is set.
- The CPU checks if any error has occurred and sets the corresponding bits in the status register in order to indicate the error. The CPU can check the status of status register at any instance.
- The interface checks three errors that may occur during the transmission :
  - **Parity error :** If the number of 1's in the received data do not have correct parity, parity error is said to be occurred.
  - **Framing error :** If correct number of stop bits are not received at the end of received data (character), framing error is said to be occurred.
  - **An overrun error :** If CPU does not read the received data (character) from the receiver register before accepting new data (character) in the shift register, an overrun error is said to be occurred.

9.5.3 First - In First - Out (FIFO) Buffer

- Fig. 9.5.2 shows FIFO buffer block diagram. The significance of FIFO buffer is that it is a memory unit in which the first input data (character) is the first output data

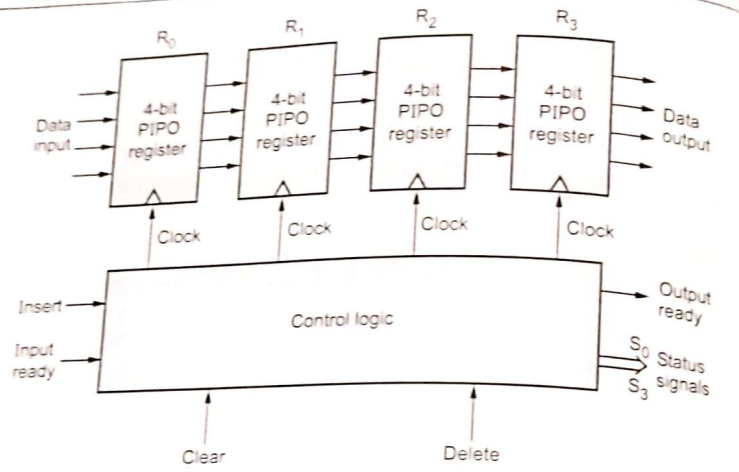


Fig. 9.5.2 Block diagram of FIFO buffer

(character). FIFO buffer is placed between two units, A source unit that gives input to FIFO buffer and a destination unit that receives output from the FIFO buffer.

- In Fig. 9.5.2, we have four data input and four data output and a control logic circuit.

Advantages of FIFO Buffer

- It can receive the input at certain data transfer rate and can output it at a different transfer rate.
- The output obtained from FIFO buffer is always in the same sequence as that of the input fed to it.
- It can match the slower input data rate with a faster output data rate and vise versa. Thus, preferred in the applications where asynchronous data transfer is used.

- Each 4-bit register can store a word with 4-bit word length. Thus the circuit in Fig. 9.5.2 can store 4 words each of 4 bit at a time. The number of bits of a word can be increased by increasing the number of data inputs given to each register whereas increasing the number of registers results into increase in number of words.
- The control logic has four inputs : insert, input ready, clear and delete. The outputs are output ready, status signals and clock signals.

- When the status signals S_0 to S_3 are set (logic level 1), the corresponding registers R_0 to R_3 have valid data. If any of the status signals (S_0 to S_3) go low (logic level 0) then that corresponding register has invalid data. For example if $S_1 = 1$ then R_1 will have 4-bit valid data and if $S_1 = 0$ then R_1 will have invalid data.
- The control logic decides the direction of flow of data in the registers connected to it. Clear signal of the control logic is used to initialise new set of data to be transferred.
- On enabling the input ready signal, data can be fed into the register R_0. Thus is done when S_0 is reset indicating that R_0 is empty. The insert control line enables the clock of R_0 register and thus makes R_0 possible to accept data from data input lines. Since the clock being enabled, the input ready signal is disabled indicating that the FIFO is busy and cannot accept new data. In the same way R_1 is indicated as empty register, leading R_0 to transfer its data into R_1 and enables the input ready signal indicating that the FIFO is free to accept next data word. As the FIFO becomes full, input ready signal is cleared i.e. set to 0. Insert and input ready signals are known as **destination initiated handshaking signals.**
- At the output end, the data received by the registers is stacked up in the sequence of their reception.
- When $S_3 = 1$, the output ready is enabled and thus indicates that a valid data is available at R_3 register. The destination unit receives the output from the FIFO buffer and enables the delete signal of the control logic. Further the output ready signal is disabled and hence the FIFO declares that the next data (character) received is invalid output data (character). The data can move from register R_3 to register R_4 only when delete signal is disabled. There will be no data in R_3 register if FIFO is empty. Thus, output ready and delete signals are known as **source initiated handshaking signals.**

Review Questions

1. *Explain in detail asynchronous communication interface.*
2. *What are the three types of errors that may occur during transmission ?*
3. *Write a short note on FIFO buffer.*

9.6 Modes of Transfer

9.6.1 Programmed I/O

- I/O operations will mean a data transfer between an I/O device and memory or between an I/O device and the processor.

- The processor's software checks the I/O devices every so often. During this check, the processor tests to see if any device needs servicing. Fig. 9.6.1 shows the flowchart for this.

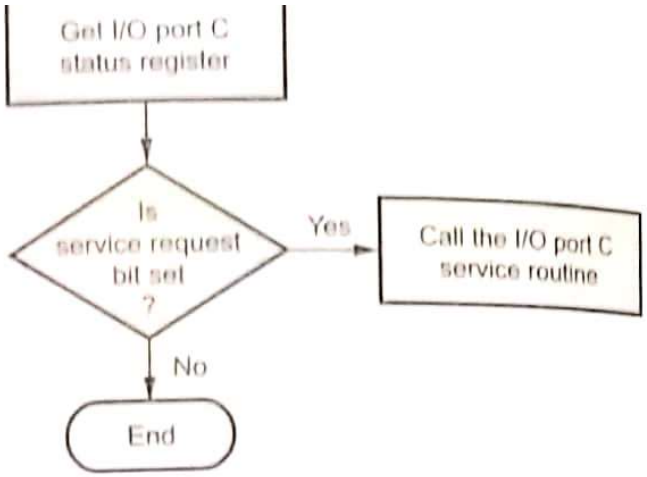


Fig. 9.6.1 Flowchart for I/O service routine

- This is a simple program which services I/O ports A, B and C. The routine checks the status of I/O ports in proper sequence.
- If the request bit is set, the corresponding service routine is called to complete the I/O request.
- This test and service procedure continues until all the I/O port status registers are tested and all the I/O ports requesting service are serviced. Once this is done, the processor continues to execute the normal programs.
- The routine assigns priorities to the different I/O devices. Once the routine is started, the service request bit at port A is always checked first. Once port A is checked, port B is checked, and then port C. However, the order can be changed by simply changing the routine and thus the priorities.

9.6.2 Interrupt-Initiated I/O

- Sometimes it is necessary to have the computer automatically execute one of a collection of special routines whenever certain conditions exists within a program

9.6.2.2 Handling Multiple Devices

- To handle interrupts from multiple devices processor has to perform following tasks.
  - It has to recognize the device requesting an interrupt.
  - It has to obtain the starting address of interrupt service routine corresponds to interrupt request. Because each interrupt request has its own interrupt service routine.
  - It has to allow the device to interrupt while another interrupt is being serviced.
  - It has to take decision that which interrupt should be serviced first when there are simultaneous interrupt requests from two different devices.
- In modern processors such capabilities are implemented by using vectoring of interrupts, by nesting of interrupts and by assigning priorities to the interrupts.

Vectored Interrupts

- When the external device interrupts the processor (interrupt request), processor has to execute interrupt service routine for servicing that interrupt.
- If the internal control circuit of the processor produces a CALL to a predetermined memory location which is the starting address of interrupt service routine, then that address is called **vector address** and such interrupts are called **vector interrupts**.
- For vector interrupts fastest and most flexible response is obtained since such an interrupt causes a direct hardware-implemented transition to the correct interrupt-handling program. This technique is called **vectoring**.
- When processor is interrupted, it reads the vector address and loads it into the PC.
- There are two ways to support vector interrupts :
  - Fixed vector address,
  - Programmable vector address.
- The processors which support fixed vector address approach have default vector address for each interrupt. The programmer cannot change this address.
- The processor which supports programmable vector address approach maintain the table in memory called the **interrupt vector table.**
- The Interrupt Vector Table (IVT) is an array of memory locations which holds the vector addresses of interrupts supported by the processor.
- When interrupt occurs the processor reads corresponding vector address given in the interrupt vector table and proceed for execution of interrupt service routine.

5.	It does not need initialization of stack.	It needs initialization of stack.
6.	System throughput decreases as number of I/O devices connected in the system increases.	System throughput does not depend on number of I/O devices connected in the system.

Table 9.6.1 Comparison between programmed I/O and interrupt driven I/O

9.6.3 Interrupt Priority

- When interrupt requests arrive from two or more devices simultaneously, the processor has to decide which request should be serviced first and which one should be delayed. The processor takes the decision with the help of interrupt priorities. It accepts the request having the highest priority.
- In case polling to identify the interrupting device, priority is automatically assigned by the order in which devices are polled.
- No further arrangement is required to accommodate simultaneous interrupt requests.

9.6.3.1 Daisy-Chaining Priority

- In case of vectored interrupts, the priority of any device is usually determined by the way the device is connected to the processor. Most common way to connect the devices is to form a daisy chain, as shown in the Fig. 9.6.3.

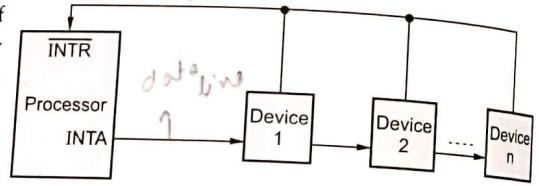


Fig. 9.6.3 Interrupt priority system using daisy-chain

9.6.3.2 Arrangement of Priority Groups

- As shown in the Fig. 9.6.3, the interrupt request line ($\overline{INTR}$) is common to all devices and the interrupt acknowledge line (INTA) is connected in a daisy-chain fashion.
- In daisy-chain fashion the signal is allowed to propagate serially through the devices.
- When more than one devices issue an interrupt request, the $\overline{INTR}$ line is activated and processor responds by setting the INTA line. This signal is received by device 1. Device 1 passes the signal to the device 2 only if it does not require any service. If device 1 requires service, it blocks the INTA line and puts its identification code on the data lines. Therefore, in daisy-chain arrangement, the device that is electrically closest to the processor has the highest priority.
- The Fig. 9.6.4 shows another arrangement for handling priority interrupts. Here, device are organised in groups and each group is connected at a different priority level. Within a group, devices are connected in a daisy-chain.

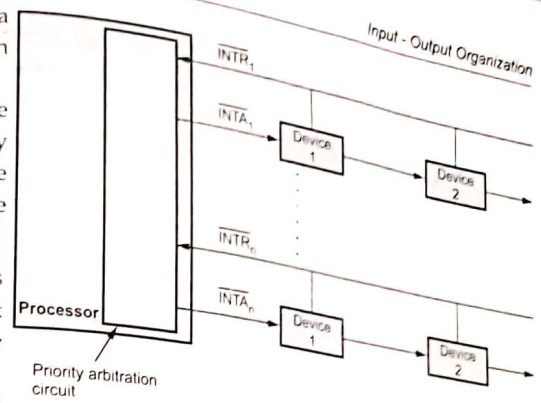


Fig. 9.6.4 Arrangement of priority groups

9.6.3.3 Vectored Interrupts Scheme / Priority Interrupt Hardware

The Fig. 9.6.5 shows a widely used vectored interrupt scheme. As shown in Fig. 9.6.5, interrupt request signals are stored in the interrupt register. The programmable interrupt mask register is used to disable any or all of the interrupt request lines. For any AND gate, one input is the interrupt request signal from interrupt register and second input is the inverted mask condition for the same interrupt from interrupt mask

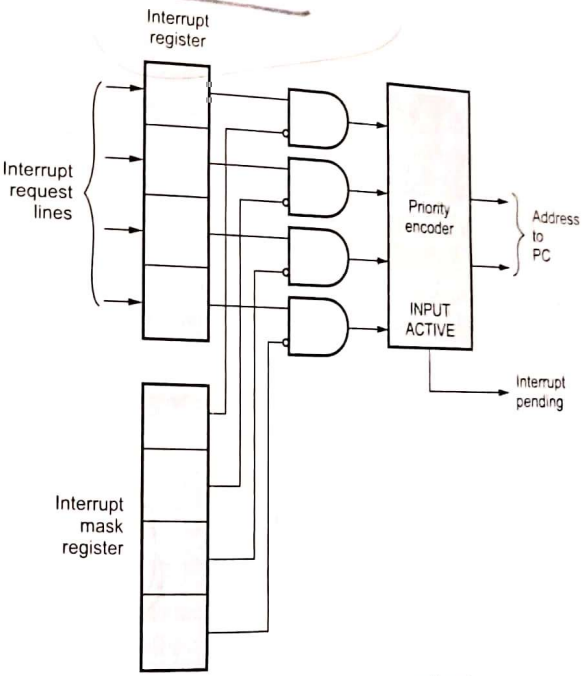


Fig. 9.6.5 A vectored interrupt scheme

register. When any interrupt is masked (1), the input to AND gate is zero, hence output goes zero. This means when interrupt is masked, the corresponding interrupt request line is disabled. When interrupt request is present for unmasked interrupt, that is fed into a priority encoder which produces the address which is then inserted into the program counter.

9.6.3.4 Priority Encoder

- A priority encoder is an encoder circuit that includes the priority function. In priority encoder, if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence.
- Table 9.6.2 shows truth table of 4-bit priority encoder.
- Table 9.6.2 shows D_3 input with highest priority and D_0 input with lowest priority. When D_3 input is high, regardless of other inputs output is (Y_1 Y_0 = 11) 11.
- The D_2 has the next priority. Thus, when $D_3 = 0$ and $D_2 = 1$, regardless of other two lower priority input, output is 10.
- The output for D_1 is generated only if higher priority inputs are 0 and so on.
- The output V (a valid output indicator) indicates, one or more of the inputs are equal to 1. If all inputs are 0, V is equal to 0 and the other two outputs (Y_1 and Y_0) of the circuit are not used.

Inputs				Outputs		
D_0	D_1	D_2	D_3	Y_1	Y_0	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

Table 9.6.2 Truth table of 4-bit priority encoder

K-map simplification

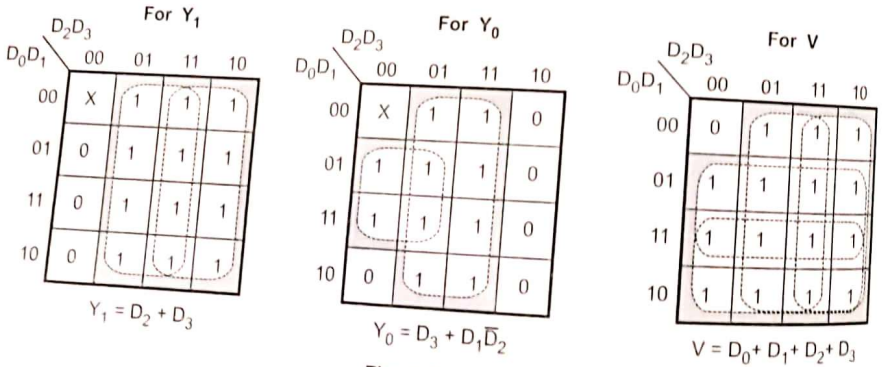


Fig. 9.6.6

Logic diagram

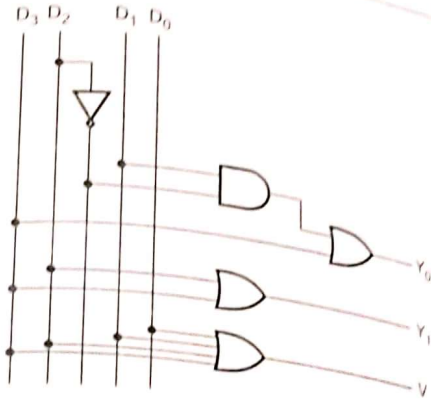


Fig. 9.6.7

9.6.4 Direct Memory Access (DMA)

- In software control data transfer, processor executes a series of instructions to carry out data transfer. For each instruction execution fetch, decode and execute phases are required. Fig. 9.6.8 gives the flowchart to transfer data from memory to I/O device.
- Thus to carry out these tasks processor requires considerable time. So this method of data transfer is not suitable for large data transfers such as data transfer from magnetic disk or optical disk to memory.

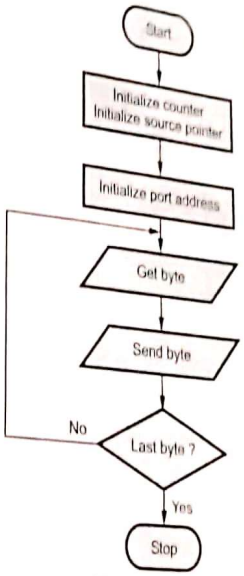


Fig. 9.6.8

Drawbacks in programmed I/O and interrupt driven I/O

1. The I/O transfer rate is limited by the speed with which the CPU can test and service a device.
2. The time that the CPU spends testing I/O device status and executing a number of instructions for I/O data transfers can often be better spent on other processing tasks.

- To overcome above drawbacks an alternative technique, hardware controlled data transfer can be used.

9.6.4.1 DMA Operation

- DMA controlled data transfer is used for large data transfer. For example to read bulk amount data from disk to main memory.
- To read a block of data from the disk processor sends a series of commands to the disk controller device telling it to search and read the desired block of data from the disk.
- When disk controller is ready to transfer first byte of data from disk, it sends DMA request DRQ signal to the DMA controller.
- Then DMA controller sends a hold request HRQ, signal to the processor HOLD input. The processor responds this HOLD signal by floating its buses and sending out a hold acknowledge signal HLDA, to the DMA controller.
- When the DMA controller receives the HLDA signal, it takes the control of system bus.
- When DMA controller gets control of the buses, it sends the memory address where the first byte of data from the disk is to be written. It also sends a DMA acknowledge, DACK signal to the

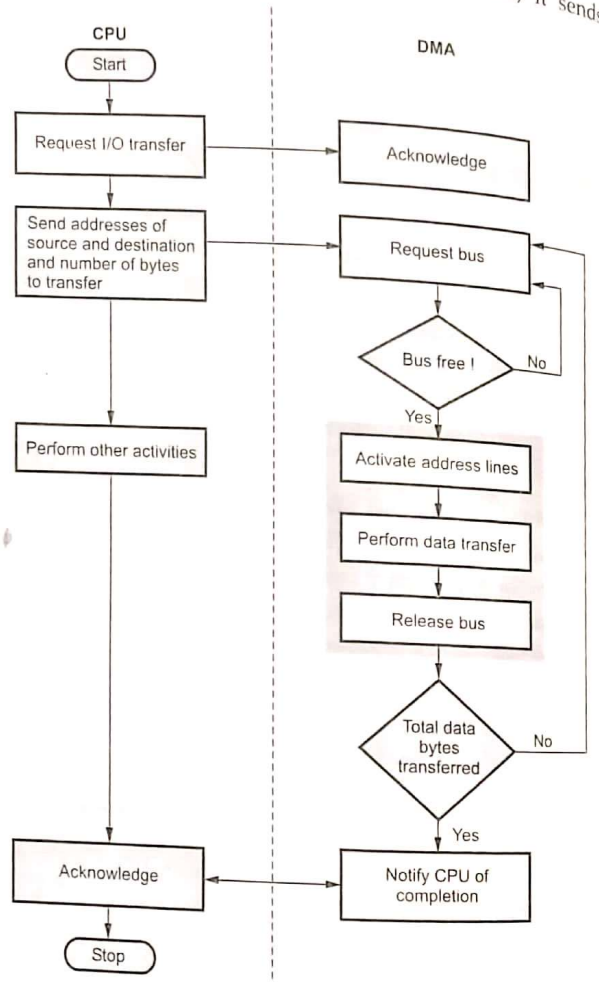


Fig. 9.6.9 Interaction of CPU and DMA

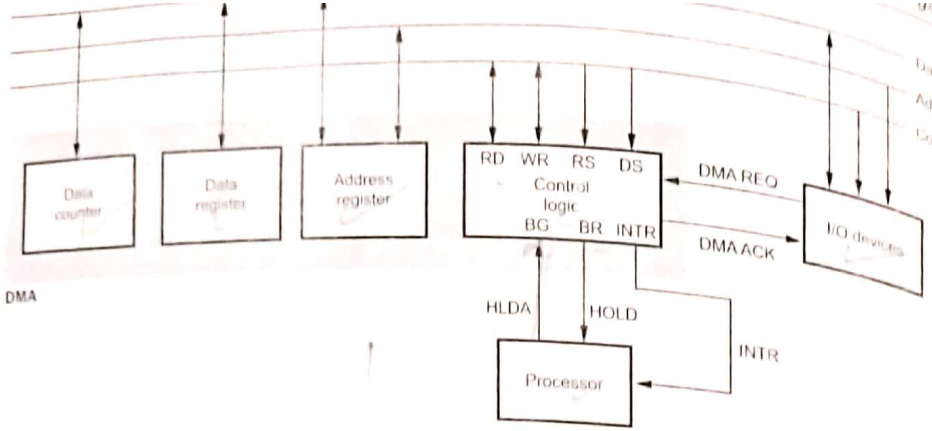


Fig. 9.6.10 Typical DMA block diagram

- 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 signals (RD and WR are now output signals for DMA). DMA consists of data count, data register, address register and control logic.
- Data counter register stores the number which gives the number data transfers to be done in one DMA cycle. It is automatically decremented after each word transfer.
- Data register acts as buffer whereas address register initially holds the starting address of the device. Actually, it stores the address of the next word to be transferred. It is automatically incremented or decremented after each word transfer.
- After each transfer, data counter is tested for zero. When the data count reaches zero, the DMA transfer halts.
- The DMA controller is normally provided with an interrupts capability, in which case it sends an interrupt to processor to signal the end of the I/O data transfer.

9.6.4 Data Transfer Modes

- DMA controller transfers data in one of the following three modes :
  - Single transfer mode (cycle stealing)
  - Block transfer mode
  - Demand or burst transfer mode

Single transfer mode (Cycle stealing mode)

In this mode device can make only one transfer (byte or word). After each transfer DMAC gives the control of all buses to the processor. Due to this processor can have access to the buses on a regular basis.

It allows the DMAC to time share the buses with the processor, hence this mode is most commonly used.

The operation of the DMA in a single transfer mode is as given below :

1. I/O device asserts DRQ line when it is ready to transfer data.
2. The DMAC asserts HLDA line to request use of the buses from the processor.
3. The processor asserts HLDA, granting the control of buses to the DMAC.
4. The DMAC asserts $\overline{DACK}$ to the requesting I/O device and executes DMA bus cycle, resulting data transfer.
5. I/O device deasserts its DRQ after data transfer of one byte or word.
6. DMA deasserts $\overline{DACK}$ line.
7. The word/byte transfer count is decremented and the memory address is incremented.
8. The HOLD line is deasserted to give control of all buses back to the processor.
9. HOLD signal is reasserted to request the use of buses when I/O device is ready to transfer another byte or word. The same process is then repeated until the last transfer.
10. When the transfer count is exhausted, terminal count is generated to indicate the end of the transfer.

Block transfer mode

In this mode device can make number of transfers as programmed in the word count register. After each transfer word count is decremented by 1 and the address is decremented or incremented by 1. The DMA transfer is continued until the word count "rolls over" from zero to FFFFH, a Terminal Count (TC) or an external END of Process ($\overline{EOP}$) is encountered. Block transfer mode is used when the DMAC needs to transfer a block of data.

The operation of DMA in block transfer mode is as given below :

1. I/O device asserts DRQ line when it is ready to transfer data.
2. The DMAC asserts HLDA line to request use of the buses from the microprocessor.
3. The microprocessor asserts HLDA, granting the control of buses to the DMAC.
4. The DMAC asserts $\overline{DACK}$ to the requesting I/O device and executes DMA bus cycle, resulting data transfer.
5. I/O device deasserts its DRQ after data transfer of one byte or word.
6. DMA deasserts $\overline{DACK}$ line.

7. The word/byte transfer count is decremented and the memory address is incremented.
8. When the transfer count is exhausted, the data transfer is not complete and the DMAC waits for another DMA request from the I/O device, indicating that it has another byte or word to transfer. When DMAC receives DMA request these through are repeated.
9. If the transfer count is not exhausted, the data transfer is complete then DMAC deasserts the HOLD to tell the microprocessor that it no longer needs the buses.
10. Microprocessor then deasserts the HLDA signal to tell the DMAC that it has resumed control of the buses.

Demand transfer mode

In this mode the device is programmed to continue making transfers until a TC or external $\overline{EOP}$ is encountered or until DREQ goes inactive.

The operation of DMA in demand transfer mode is as given below :

1. I/O device asserts DRQ line when it is ready to transfer data.
2. The DMAC asserts HLDA line to request use of the buses from the microprocessor.
3. The microprocessor asserts HLDA, granting the control of buses to the DMAC.
4. The DMAC asserts $\overline{DACK}$ to the requesting I/O device and executes DMA bus cycle, resulting data transfer.
5. I/O device deasserts its DRQ after data transfer of one byte or word.
6. DMA deasserts $\overline{DACK}$ line.
7. The word/byte transfer count is decremented and the memory address is incremented.
8. The DMAC continues to execute transfer cycles until the I/O device deasserts DRQ indicating its inability to continue delivering data. The DMAC deasserts HOLD signal, giving the buses back to microprocessor. It also deasserts $\overline{DACK}$.
9. I/O device can re-initiate demand transfer by reasserting DRQ signal.
10. Transfer continues in this way until the transfer count has been exhausted.

The flowcharts in the Fig. 9.6.11 summarized the three data transfer modes of DMA.

9.6.4.5 Use of DMA in a Computer System

- The Fig. 9.6.12 (a) shows the use of DMA in a computer system.
- The DMA is used to connect a high-speed network to the computer bus. The DMA control handles the data transfer between high-speed network and the computer system.
- It is also used to transfer data between processor and floppy disk with the help of floppy disk controller.
- Let us see how DMA controller does the data transfer between floppy disk and the processor. The Fig. 9.6.12 (b) shows the interface required for such transfer.
- The sequence of events that takes place during the data transfer are as follows :
- When processor needs some data from the disk, it sends a series of command words to registers inside the floppy disk controller.
- The floppy disk controller then proceeds to find the specified track and sector on the disk. (See Fig. 9.6.12 (a) and (b) on next page).
- In the mean while processor loads the DMA data counter and address register. The data counter is loaded with count equal to the number of bytes to be transferred to or from the memory. The address register is loaded with the starting address of the memory.
- When the controller reads the first byte of data from a sector, it sends DMA request, DRQ signal to the DMA controller. DMA request sends a hold request signal to the HOLD input of the processor.
- The processor then floats its buses and sends a hold-acknowledge signal to the DMA controller.
- The DMA controller then sends out the first transfer address on the bus and asserts the $\overline{DACK}$ input of the FDC to tell it that the DMA transfer is underway.
- When the number of bytes specified in the DMA controller initialization has been transferred, the DMA controller asserts the $\overline{EOP}$ (end of process) signal, which is connected to the TC (Terminal Count) input of the FDC.
- This causes FDC to generate interrupt signal to tell the processor that the requested block of data has been read from disk to a buffer in memory.
- Similar process is required to write data into the disk.

9.6.5 I/O Processor (IOP)

The I/O processor (IOP) has an ability to execute I/O instructions and it can have complete control over I/O operation. The I/O instructions are stored in main memory. When I/O transfer is required, the CPU initiates an I/O transfer by instructing the I/O

9.6.5.3 CPU and IOP Communication

The communication between CPU and IOP may be different for different processor and IOP configurations. However, in most of cases the memory based control blocks are used to store the information about the task to be performed. The processor uses these blocks to leave information in it for the other processor. The memory control block are linked, i.e. the address of the next memory based control blocks is available in the previous memory based control block.

The Fig. 9.6.13 shows the flowchart of sequence of operations that are carried out during the CPU and IOP communication.

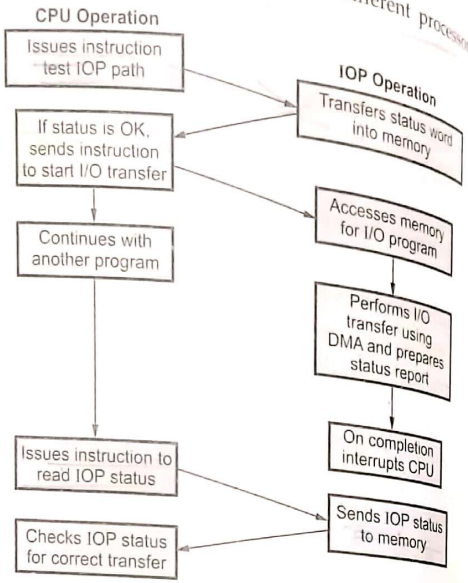


Fig. 9.6.13 CPU and IOP communication

The sequence of operations carried out during CPU and IOP communication are :

1. CPU checks the existance of I/O path by sending an instruction.
2. In response to this IOP puts the status word in the memory stating the condition of IOP and I/O device (Busy, ready, etc.)
3. CPU checks the status word and if all conditions are OK, it sends the instruction to start I/O transfer along with the memory address where the IOP program is stored.
4. After this CPU continues with another program.
5. IOP now conducts the I/O transfer using DMA and prepares status report.
6. On completion of I/O transfer, IOP sends an interrupt request to the CPU.
7. The CPU responds to the interrupt by issuing an instruction to read the status from the IOP. The status indicates whether the transfer has been completed or if any errors occurred during the transfer.

ITB	0F	End of intermediate transmission block
ETB	17	End of transmission block
ETX	03	End of text

Table 9.7.2

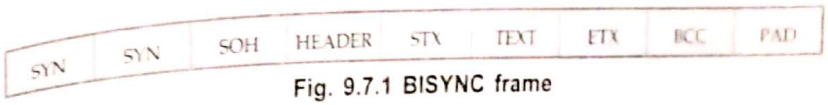


Fig. 9.7.1 BISYNC frame

BISYNC is usually set up for half-duplex transmission. The transmitter sends a block of data and then waits while the receiver checks for errors.

Errors : If the receiver detects no errors, it sends an acknowledge to the transmitter. The transmitter spends about 50 % of this time waiting for acknowledge signal, which reduces the transmission efficiency. Serial transmission protocol is Serial Data Link Control (SDLC).

9.7.2.3 HDLC and SDLC Protocols : Bit Oriented Protocols

The BISYNC protocol just discussed has following disadvantages :

- It works only in half-duplex mode.
- Except for XMODEM, it has difficulty in transmitting pure 8-bit binary data such as object code for programs.
- It is not easily adapted to serve multiple units sharing a common data link.

To solve these difficulties in BISYNC protocol the International Standards Organization (ISO) proposed the High-Level Data Link Control protocol (HDLC) and IBM developed the Synchronous Data Link Control (SDLC) protocol. These two protocols are nearly identical. Let us study them and understand any significant differences between them.

We have seen that BISYNC is a byte controlled protocol. Unlike BISYNC, SDLC and HDLC are **Bit Oriented Protocols (BOP)**. In these protocols, the messages are treated simply as a string of bits rather than a string of characters. The group of bits which make up a message is referred to as a **frame**. Three such frames are used. These are :

- Information frame (I frame)
- Supervisory Control Sequence frame (S frame)
- Command/response frame (U frame)

They have same basic format, as shown in Fig. 9.7.2

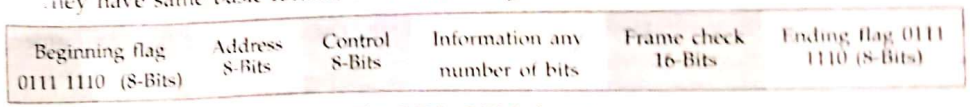


Fig. 9.7.2 SDLC frame

SDLC/HDLC frame format	Bits in control field							
	7	6	5	4	3	2	1	0
I-frame (Commands/Responses)	Nr	Nr	Nr	P/F	Ns	Ns	Ns	0
S-frame (Commands/Responses)	Nr	Nr	Nr	P/F	S	S	0	1
U-frame (Commands/Responses)	M	M	M	P/F	M	M	1	1

Fig. 9.7.3 Details of 8-bit control field of frame

Note

- Sending order - Bit 0 first, Bit 7 last
- Nr - The transmitting station receive sequence number, Bit 6 is the low-order bit.
- Ns - The transmitting station send sequence number, Bit 2 is the low-order bit.
- P/F - The POLL bit for primary station transmissions, and the FINAL bit for secondary station transmissions.
- S - The supervisory function bits
- M - The modifier function bits

Each part of the frame is referred to as **field**. The start of transmission is indicated by beginning flag (0111 1110) of the flag field. Immediately after the flag field is an 8-bit address field. It allows each frame to be addressed to a particular station among a network of stations. The next field is a control field. Each bit in the control character has a specific meaning as shown in the Fig. 9.7.3

The information field, which is present only in information frames, can have any number of bits in HDLC protocol, but in SDLC the number of bits has to be a multiple of 8. A special hardware circuit is provided to modify the flag bit pattern (0111 1110) in the information field. The hardware modifies the bit stream between flags so that there are never more than five 1's in sequence. It inserts 0-bit after any series of five 1's. A complementary circuit is provided at the receiving end to remove these extra zeros.

The next field in a frame is the 16-bit frame check sequence (FCS) field. This field contains the cyclic redundancy word derived from all the bits between the beginning and end flags. This CRC value is recomputed by the receiving system to check for errors.

Finally, a frame is terminated by another flag byte. The ending flag for one frame may be the starting flag for another frame.

In HDLC/SDLC protocols the communication takes place between two or more stations in the network. In which one of the station is always set up as a controller for the link. This station is called the **primary station**. Other stations on the link are referred

10.1 Memory Hierarchy

- Ideally, computer memory should be fast, large and inexpensive. Unfortun
  is impossible to meet all the three of these requirements using the
  memory.
- Increased speed and size are achieved at increased cost.
- Very fast memory system can be achieved if SRAM chips are used. Th
  are expensive and for the cost reason it is impracticable to build a lar
  memory using SRAM chips. The only alternative is to use DRAM chips
  main memories.
- Processor fetches the code and data from the main memory to exe
  program. The DRAMs which form the main memory are slower devices
  necessary to insert wait states in memory read/write cycles. This red
  speed of execution.
- The solution for this problem is come out with the fact that most of the
  programs work with only small sections of code and data at a particular
  the memory system, small section of SRAM is added along with main
  referred to as **cache memory**.
- The program which is to be executed is loaded in the main memory, bu
  of program (code) and data that work at a particular time is usually acce
  the cache memory.
- This is accomplished by loading the active part of code and data f

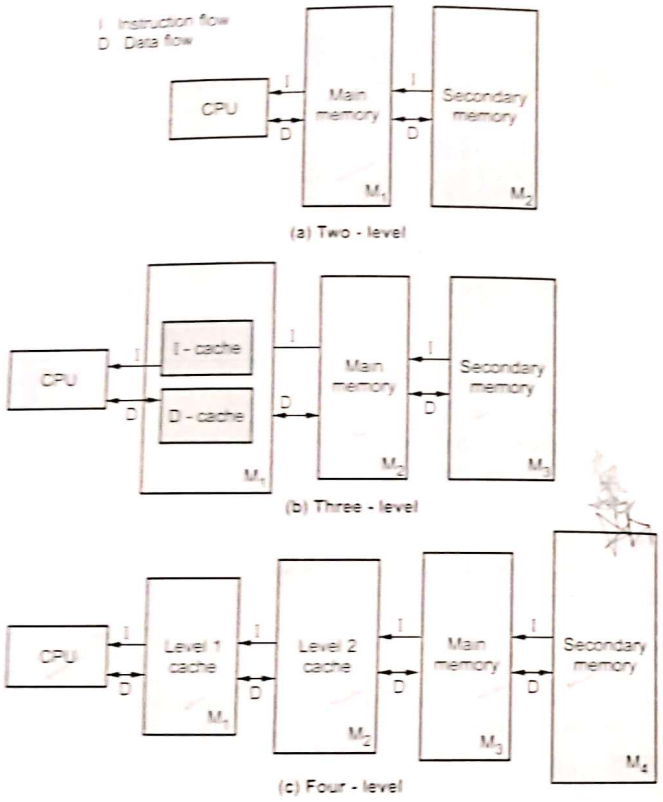


Fig. 10.1.3 Memory hierarchy

10.1.1 Multiprogramming

- Now-a-days many operating systems can execute number of programs concurrently. This concept is known as **multi programming**, where two or more programs exist simultaneously at different locations of memory hierarchy. Thus the computer remains busy in execution of several programs in a particular sequence.
- **Example :** While a program execution is going on by the CPU if an I/O transfer requirement is raised, the CPU initializes the I/O processor to start the I/O data transfer enabling the CPU to be available for other program execution.

Requirements of multiprogramming are :

- Run partial programs
- Vary main memory utilization for a particular program.
- Movement of program across memory hierarchy.

- Sometimes the program length is too large that main memory space is not enough for its storage. And also a computer system consist of number of programs which cannot be stored in the main memory all the time. Here arises the requirement of auxiliary memory.
- **Auxiliary memory** stores all programs. The required program or segment of program is transferred to the main memory and the CPU executes the program.
- An operating system maintains the recently active portion of information in the main memory.
- The **memory management system** controls the flow of data (information) between the auxiliary memory and main memory.

Review Questions

1. *Write a short note on memory hierarchy*
2. *What is multiprogramming ?*

10.2 Main Memory

- The main memory of a computer is used to store data and programs during computer operation. It uses a semiconductor technology and hence commonly known as **semiconductor memory.**
- The Fig. 10.2.1 shows the classification of semiconductor memory.

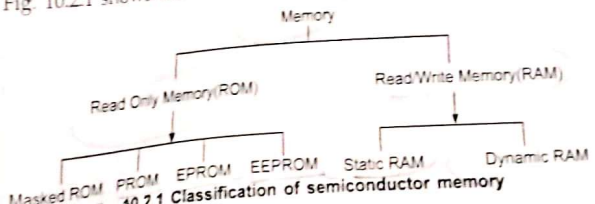


Fig. 10.2.1 Classification of semiconductor memory

10.2.1 Semiconductor Read/Write Memories

- The two basic forms of semiconductor read/write memories are :
  - Static RAM (SRAM)
  - Dynamic RAM (DRAM)

10.2.2 Static RAM

- Memories that consists of circuits capable of retaining their state as long as power is applied are known as **static memories**.
- These are Random Access Memory (RAM) and hence combinedly called **static RAM memories**.

10.2.2.1 Static RAM Cell

- The Fig. 10.2.2 shows the implementation of static RAM cell. It consists of two cross-coupled inverters as a latch and two transistors T_1 and T_2 which act as a switches.

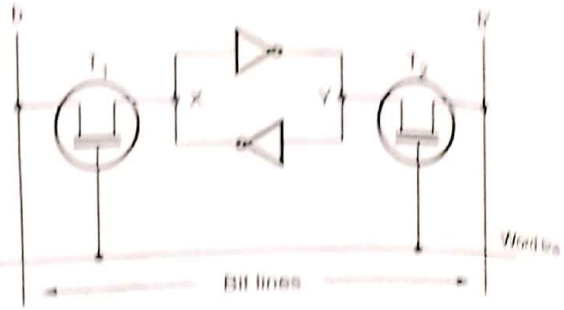


Fig. 10.2.2 Static RAM cell

- The latch is connected to two bit lines by transistors T_1 and T_2. The word line controls the opening and closing of transistors T_1 and T_2. When word line is at logic 0 level (Ground level), the transistors are off and the latch retains its state.

Read operation

- For read operation, word line is made logic 1 (high) so that both transistors are ON. Now if the cell is in state 1, the signal on bit line b is high and the signal on bit line b' is low. The opposite is true if the cell is in state 0. The b and b' are complements of each other. The sense/write circuits connected to the bit lines monitor the states of b and b' and set the output accordingly.

Write operation

- For write operation, the state to be set is placed on the line b and its complement is placed on line b' and then the word line is activated. This action forces the cell into the corresponding state and write operation is completed.

10.2.2.2 CMOS Cell

- The Fig. 10.2.3 shows the CMOS cell. Here, transistor pairs (T_3, T_5) and (T_4, T_6) form the cross-coupled inverters and transistors T_1 and T_2 act as a switches. The latch is connected to two bit lines by transistors T_1 and T_2.
- The word line controls the opening and closing of transistors T_1 and T_2. When word line is at logic 0 level (ground level), the transistors T_1 and T_2 are off and the latch retains its state.

Sr. No.	Static RAM	Dynamic RAM
1.	Static RAM contains less memory cells per unit area.	Dynamic RAM contains more memory cells as compared to static RAM per unit area.
2.	It has less access time hence faster memories.	Its access time is greater than static RAMs.
3.	Static RAM consists of number of flip-flops. Each flip-flop stores one bit.	Dynamic RAM stores the data as a charge on the capacitor. It consists of MOSFET and the capacitor for each cell.
4.	Refreshing circuitry is not required.	Refreshing circuitry is required to maintain the charge on the capacitors after every few milliseconds. Extra hardware is required to control refreshing. Thus makes system design complicated.
5.	Cost is more.	Cost is less.

10.2.5 Semiconductor Read Only Memories (ROM)

- We can't write data in Read Only Memories (ROM). It is non-volatile memory i.e. it can hold data even if power is turned off.
- Generally, ROM is used to store the binary codes for the sequence of instructions and data such as look up tables. This is because this type of information does not change.
- ROMs are also accessed randomly with unique addresses.
- The Fig. 10.2.5 shows the typical configuration of a ROM cell. It consists of a transistor T and switch P.
- The transistor T is driven by the word line.
- The contents of cell can be read from the cell when word line is logic 1.
- A logic value 0 is read if the transistor is connected to ground through switch P. If switch P is open, a logic value 1 is read.
- The bit line is connected through a resistor to the power supply.
- A sense circuit at the end of the bit line generates the proper output value.

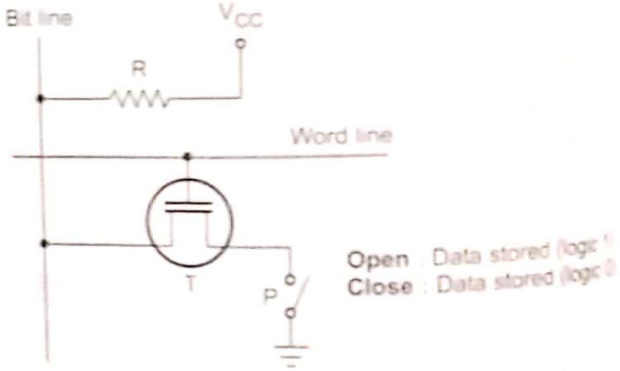


Fig. 10.2.5 ROM cell

- The important point is that we can erase the stored data in the EPROMs by exposing the chip to ultraviolet light through its quartz window for 15 to 20 minutes, as shown in the Fig. 10.2.7.

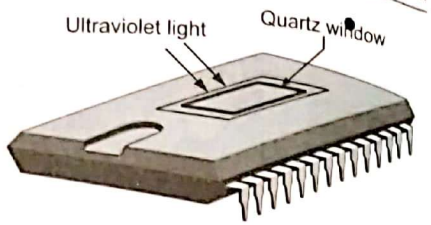


Fig. 10.2.7 EPROM

- It is not possible to erase selective information, when erased the entire information is lost.
- The chip can be reprogrammed.
- This memory is ideally suitable for product development, experimental projects and college laboratories, since this chip can be reused many times.

EPROM Programming :

- When erased each cell in the EPROM contains 1. Data is introduced by selectively programming 0's into the desired bit locations. Although only 0's will be programmed, both 1's and 0's can be presented in the data.
- During programming address and data are applied to address and data pins of the EPROM. When the address and data are stable, program pulse is applied to the program input of the EPROM. The program pulse duration is around 50 ms and its amplitude depends on EPROM IC. It is typically 5.5 V to 25 V.
- In EPROM, it is possible to program any location at any time - either individually, sequentially or at random.

10.2.5.3 EEPROM (Electrically Erasable Programmable Read Only Memory)

- Electrically erasable programmable ROMs also use MOS circuitry very similar to that of EPROM.
- Data is stored as charge or no charge on an insulated layer or an insulated floating gate in the device.
- The insulating layer is made very thin ($< 200\ \text{Å}$). Therefore, a voltage as low as 20 to 25 V can be used to move charges across the thin barrier in either direction for programming or erasing.
- EEPROM allows selective erasing at the register level rather than erasing all the information since the information can be changed by using electrical signals.
- The EEPROM memory also has a special chip erase mode by which entire chip can be erased in 10 ms. This time is quite small as compared to time required to erase EPROM. It can be erased and reprogrammed with device right in the circuit.

Disadvantage

- EEPROMs are most expensive and the least dense ROMs.

10.2.5.4 Bootstrap Loader

- **Bootstrap loader** program is used to start the computer software operation after turning ON the power.
- Bootstrap loader is stored in the ROM of main memory.
- As RAM is a volatile memory, data contents of RAM are destroyed after turning OFF the power.
- ROM is nonvolatile in nature i.e. the data contents in ROM remain unchanged irrespective of the power OFF or power ON condition.
- The computer start up turns ON the computer system and initiates the program execution of the initial program.
- As the power is turned ON, the program counter is set to the first address of the bootstrap loader by the computer hardware.
- The part of operating system is loaded from the disk to the main memory, by the bootstrap loader and thus the control is given to the operating system. Thus , the computer is enabled for general purpose.

10.2.6 RAM and ROM Chips

- Today various sizes of RAM and ROM chips are available in the market.
- Many times a single chip is not sufficient to fulfill the memory requirement of a computer system. To overcome this problem, two or more number of chip are combined and the required memory size is obtained.

10.2.6.1 RAM Chips

- **The reason why a RAM chip is preferred to communicate with the CPU.**
  - It may have one or more control inputs, enabling the user to select only the required chip.
  - It has bidirectional data bus which allows the data transfer to and from the CPU and memory required during the read and write operations.
- A bidirectional data bus consist of tristate buffer (buffer with three states).
- The three states of tristate buffer are :
  - Logic zero
  - Logic 1
  - High impedance

- Logic zero and logic 1 states are digital signals whereas the high impedance state is equivalent to open circuit i.e. there is no signal at the output and thus has no logical significance.
- Fig. 10.2.8 shows the block diagram of a RAM chip.
- A RAM chip consist of chip select signal $(\overline{CS})$, read/write signal $(R / \overline{W})$, address bus and bidirectional data bus.

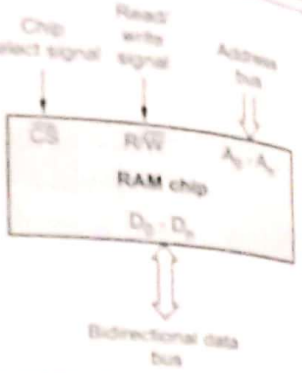


Fig. 10.2.8 Block diagram of RAM chip

Note In some RAM ICs there are more than one chip select $(\overline{CS})$ signals. For example, CS_1 is active high signal and $\overline{CS_2}$ is active low signal and when $CS_1 = 1$, $\overline{CS_2} = 0$ the corresponding RAM chip get selected.

- In general, **chip select (CS) signal** is active low signal i.e. the RAM chip with $\overline{CS} = 0$ is selected. While in case of **read/write signal (R/W)**, when $R/\overline{W} = 1$ read operation is performed whereas when $R/\overline{W} = 0$ write operation is performed.
- Table 10.2.1 shows the function table of the control signals related to a RAM chip.

$\overline{CS}$	$R/\overline{W}$	Memory operation	Data bus state
1	X	Undefined	High impedance
0	X	Undefined	High impedance
0	0	Write	Input data into RAM chip
0	1	Read	Output data from RAM chip

Table 10.2.1 Function table of a typical RAM chip

- The **address bus** specifies the location in RAM from where the data is to be read or the location into which data is to be written.
- The **bidirectional data bus** specifies that the data can move in both the directions i.e. the data can be read from and can also be written into the RAM chip.
- The $R/\overline{W}$ signal can control the memory operations and bus buffers corresponding to the data bus.

10.2.6 ROM Chips

- ROM chip is similar to that of the RAM chip except the fact that ROM is Read Only Memory i.e. ROM chip can only read the data and so data bus is only in the output mode.
- Fig. 10.2.9 shows the block diagram of a ROM chip.
- Considering same size of ROM and RAM chip, more number of ROM bits can occupy the given memory space as compared to that of RAM. This is possible due to the internal binary cells present in ROM require less space than that in case of RAM.
- To enable the ROM chip, chip select signal $(\overline{CS})$ must be zero. As ROM can only read data, R/W signal is not needed. The address bus specify the location in ROM from where the data is to be read. Some ROM chips may have RD signal.

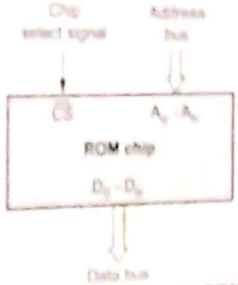


Fig. 10.2.9 Block diagram of a ROM chip

10.2.7 Memory Address Map

The maximum size of the memory that can be used in any computer is determined from the number of address lines provided by the processor used in the computer. For example, if processor has 20 address lines, it is capable of addressing up to $2^{20} = 1$ M(Mega) memory locations. Similarly, processors whose instructions generate 32-bit address can access a memory that contains up to $2^{32} = 4$ G (Giga) memory locations. The number of locations represents the size of the address space of the computer.

We have seen that, most computers are byte addressable and they can have big-endian or little-endian arrangement. The Intel processors use little-endian arrangement while big-endian arrangement is used in the Motorola 68000 processor. The ARM architecture can be configured to use either arrangement.

The maximum number of bits that can be transferred from memory or to the memory depend on the data lines supported by the processor. The number of data lines reported by processor gives the word length of the processor and hence the computer.

The control lines from the processor decides the memory operation. In case of read operation, $\overline{RD}$ signal is activated. It is used to enable the active low output enable signal of the memory. In case of write operation, $\overline{WR}$ signal is activated to indicate the write operation. The data transfer between the memory and processor takes place through the use of two processor registers, usually called MAR (Memory Address Register) or

As shown in the Fig. 10.2.11 (a) memory chip has 11 address [illegible] select ($\overline{CS}$) and two control lines, read ($\overline{RD}$) to enable output buffer and write ($\overline{WR}$) to enable the input buffer. The internal decoder is used to decode the address lines. Fig. 10.2.11 (b) shows the logic diagram of a typical EPROM (Erasable Programmable Read-Only Memory) with 4096 (4 K) registers. It has 12 address lines A_0-A_{11}, one chip select ($\overline{CS}$), one read control signal. Since EPROM is a read only memory, it does not require the ($\overline{WR}$) signal.

Example 10.2.1 *If memory is having 12 address lines and 8 data lines, then find out number of registers per memory locations and word length.*

Solution : Number of registers/memory locations = $2^N = 2^{12}$

$$= 4096$$

Word length = M-bit

= 8-bit

Example 10.2.2 *If memory has 8192 memory locations, then it has 13 address lines.*

Solution : The Table 10.2.2 summarizes the memory capacity and address lines required for memory interfacing.

Memory capacity	Address lines required
1 K = 1024 memory locations	10
2 K = 2048 memory locations	5
4 K = 4096 memory locations	12
8 K = 8192 memory locations	13
16 K = 16384 memory locations	14
32 K = 32768 memory locations	15
64 K = 65536 memory locations	16

Table 10.2.2

10.2.8 Expanding Word Size and Memory Capacity

It is possible to expand word size of memory by connecting two or more ICs together. The word size of memory IC can be increased by connecting two memory ICs in such a way that their data bus is in series and address bus in parallel. Both memory ICs are selected simultaneously by common chip select signal to access entire expanded word at a time. The example 10.2.1 illustrates method of word expansion.

Example 10.2.3 *Design 1 K × 8 RAM using two 2114 ICs.*

Solution :

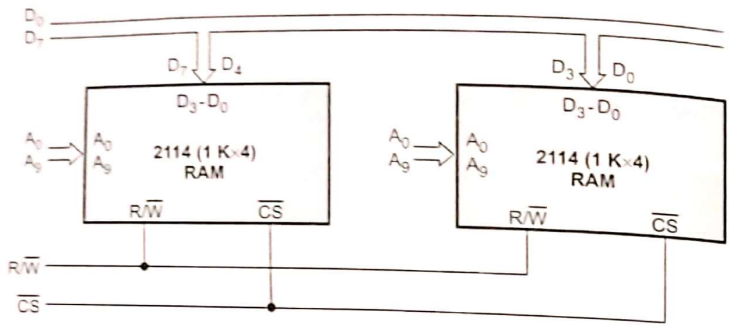


Fig. 10.2.12 Expanding word size

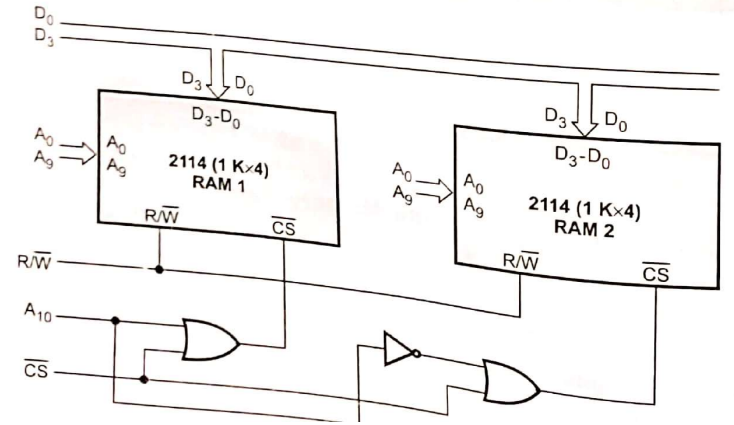
The memory capacity can be increased by connecting two or more memory ICs in parallel. This means that the address, data and control lines are connected in parallel to all memory ICs. Each IC is selected by the separate chip select signal generated by the address decoder. This is illustrated by the following example.

Memory Map

A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Memory address
0	0	0	0	0	0	0	0	0	0	Starting address
1	1	1	1	1	1	1	1	1	1	End address

Example 10.2.4 *Design 2 K × 4 RAM using two 2114 ICs.*

Solution :



Memory Map

A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Memory ICs
0	0	0	0	0	0	0	0	0	0	0	RAM1
0	1	1	1	1	1	1	1	1	1	1	
1	0	0	0	0	0	0	0	0	0	0	RAM2
1	1	1	1	1	1	1	1	1	1	1	

Example 10.2.5 *Implementation of 2M words of 32-bit each using 512 K × 8 static memory chips.*

Solution : The Fig. 10.2.14 shows the implementation of 2 M words of 32-bit each using 512 K×8 static memory chip. In this implementation, we have expand word size as well as capacity of the memory. Since each memory chip has byte organization, we have to use 4 chips in each row to get the word size of 32 bits (8 × 4). Similarly, to increase memory capacity up to 2 M we have to use 4 memory chips of capacity 512 K in each column. (See Fig. 10.2.14 on next page.)

Example 10.2.6 *For a 32 K memory, obtain memory map to interface memory to 16-bit processor.* **RGTU : May-06**

Solution :

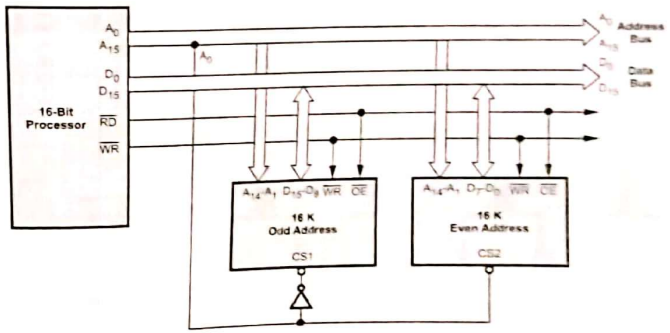
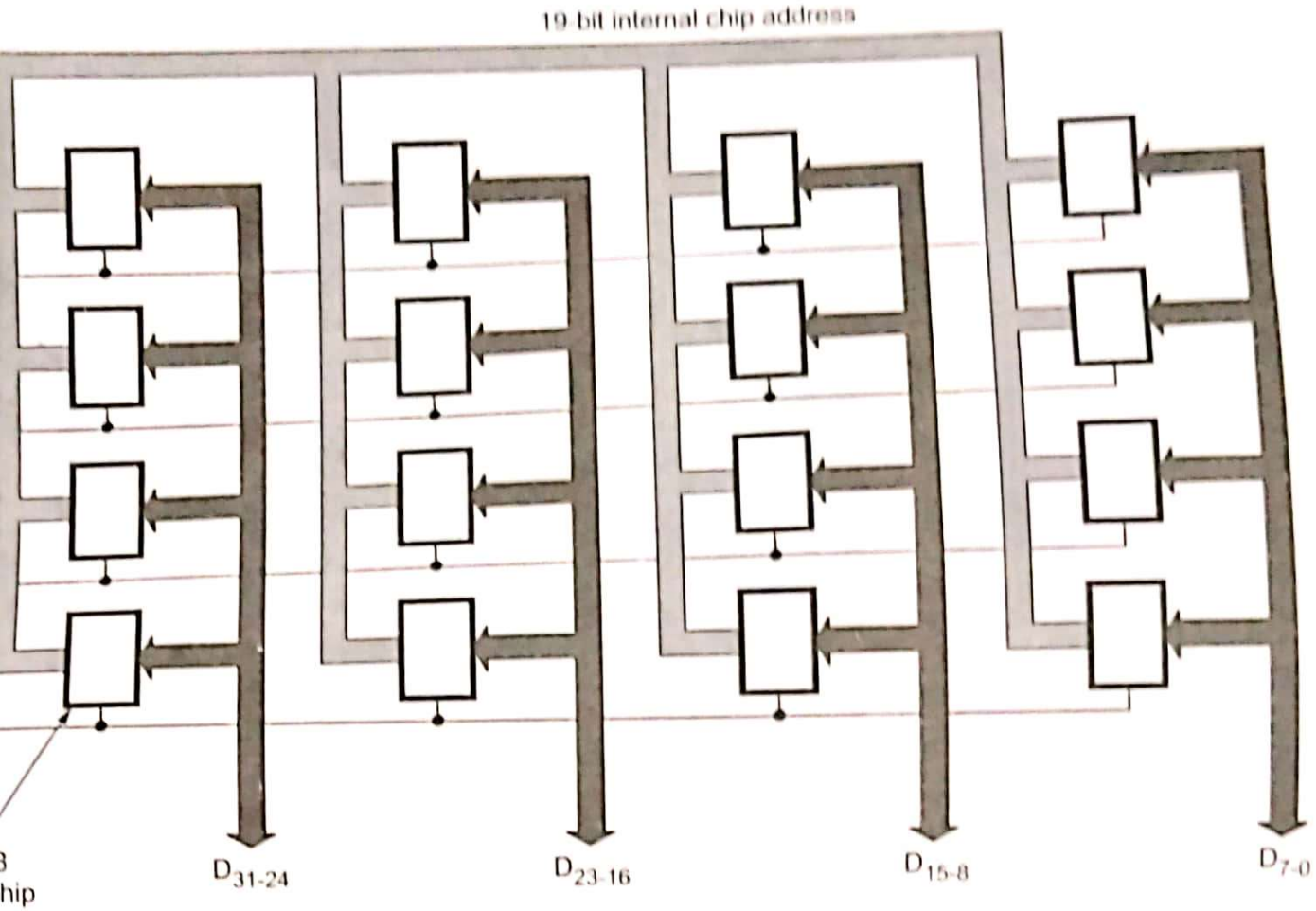


Fig. 10.2.15

Memory Map :

Memory	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	Address
Even Address	X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0000H
	X	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	7FFEH
Odd Address	X	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0001H
	X	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	7FFFH


19-bit internal chip address
D31-24
D23-16
D15-8
D7-0

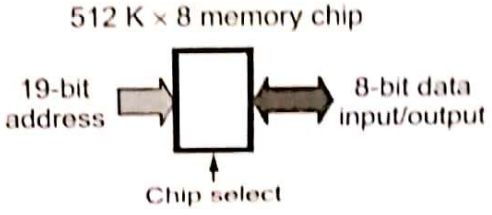

512 K × 8 memory chip
19-bit address
8-bit data input/output
Chip select

Fig 10.2.14

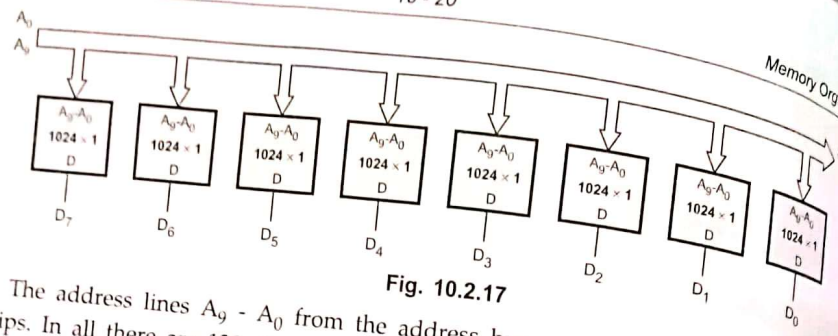


Fig. 10.2.17

The address lines A_9 - A_0 from the address bus are connected in parallel to all the chips. In all there are 128 chips. These 128 chips are divided into 16 groups consists of 8 chips in each group. The address lines A_{13} - A_{10} are decoded to generate 16 chip select signals one for each group so that at a time all chips from one group will be selected when corresponding chip select signal is activated.

Review Questions

1. *What are the types of RAMs ?*
2. *Write a short note on SRAM.*
3. *Explain DRAM with diagram. Also give advantages and disadvantages.*
4. *Differentiate between static RAM and dynamic RAM.*
5. *Draw and explain the working of ROM cell.*
6. *Write short note on : PROM*
7. *Write short note on : EPROM*
8. *Explain EEPROM*
9. *Write a short note on bootstrap loader.*
10. *Explain RAM and ROM chips with help of block diagram.*
11. *Explain in detail the memory connection with CPU.*
12. *What is meant by memory address map ?*

10.3 Auxiliary Memory

10.3.1 Magnetic Disk Memory

10.3.1.1 Magnetic Surface Recording

A magnetic disk is a thin, circular metal plate. It is coated with a thin magnetic film, usually on both sides. Digital information is stored on the magnetic disk by magnetizing the magnetic surface in a particular direction as shown in the Fig. 10.3.1



Fig. 10.3.1 Bit representation of magnetic disk

The disks are mounted on a rotary drive so that the magnetized surface moves in close proximity to magnetizing coil or head as shown in Fig. 10.3.2. The head consists of a magnetic yoke and the magnetizing coil, as shown in the Fig. 10.3.3. Digital information can be stored on the magnetic film by applying current pulses of suitable polarity to the magnetizing coil. This causes the magnetization of the film in the area immediately below the head. The same head is used for reading the stored information. In this case, when the surface of the disk passes under the head, it generates a current and induces voltage in the coil of the same polarity as the one already recorded. The polarity of this voltage is monitored by the control circuitry to determine the state of magnetization of the film.

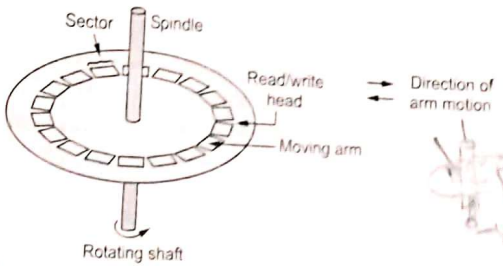


Fig. 10.3.2 (a) Rotating shaft

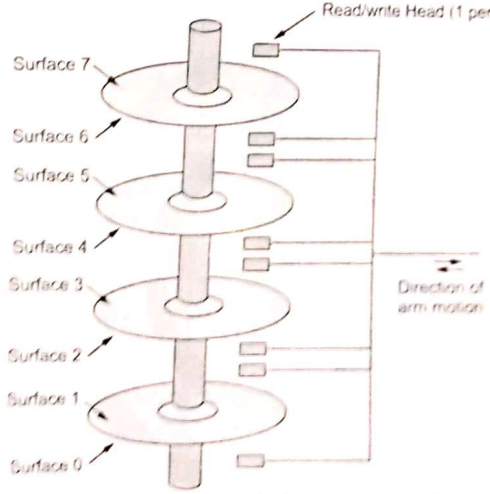


Fig. 10.3.2 (b) Mechanical structure

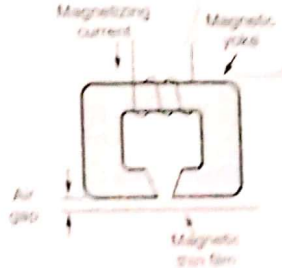


Fig. 10.3.3 Read/Write head detail

10.3.1.2 Data Organization and Formatting

The data on the disks is organized in a concentric set of rings as shown in the Fig. 10.3.4 These concentric set of rings are called as **tracks**. Each track has a same width as head and adjacent tracks are separated by **gaps**. The gap between two adjacent tracks prevents, or at least minimizes, errors due to misalignment of the head or simply interference of magnetic fields. Data bits are stored serially on each track. The same numbers of bits are stored on each track. Thus, the density, in bits per linear inch increases as we move from the outermost track to the innermost track. The tracks are further divided into sectors, as shown in the Fig. 10.3.4 Each sector stores a block of data which can be transferred to or from the disk. To avoid magnetic interference between two adjacent sectors the gap is introduced between two adjacent sectors.

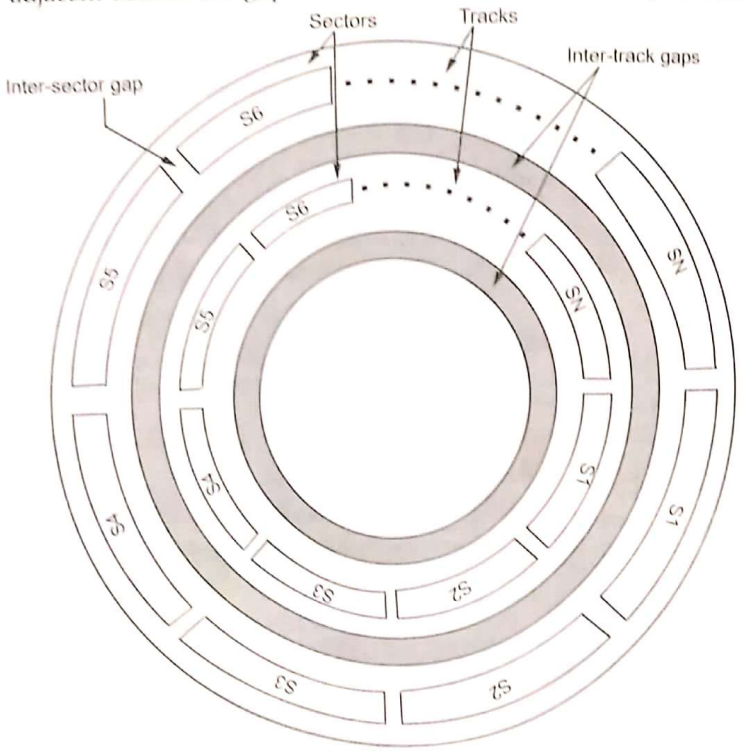


Fig. 10.3.4 Disk data layout

Data on disk is addressed by specifying the disk number or head number, track number and the sector number. The start and end of each sector is determined by the control data stored on each sector.

Fig. 10.3.5 shows disk format for disk. As shown in the Fig. 10.3.5, each track contains 30 fixed length sectors of 600 bytes each. Each sector holds 512 bytes of data plus control information useful to the disk controller. The control information is represented by ID field, which is used as an unique identifier to locate a particular sector.

In fixed head disk, there is one head per track. These heads are mounted on a rigid arm as shown in the Fig. 10.3.2 In a movable head disk, there is only one head and it is moved between tracks by control mechanism. The non removable disk is permanently mounted in the disk drive whereas removable disk can be removed and replaced with another disk.

In single sided disk, the magnetic coating is applied on only one side of the disk. On the other hand, in double sided disk it is on both the sides of disk, doubling the storage capacity.

Some disk drives accommodate multiple surfaces stacked vertically apart with separate pair of heads as shown in the Fig. 10.3.2.

Finally, disk drives are also classified according to head positions : contact, fixed gap or aerodynamic gap.

10.3.1.4 Typical Disks

The specifications of some common commercially available disk drives are shown in Table 10.3.2.

Capacity	Cylinders	Heads	Sectors	Bytes/Sector
4.302 GB	523	255	63	512
2.159 GB	523	128	63	512
1.08 GB	523	64	63	512

Table 10.3.2

10.3.1.5 Access Time

The disk access time is the time delay between receiving an address and the beginning of the actual data transfer. The disk access time has two time components seek time and rotational delay. The **seek time** is the time required to move the read/write head to the proper track. This time depends on the initial position of the head relative to the track specified in the address. The average seek time is in the range of 5-8 ms. The **rotational delay** also known as **latency time** is the amount of time that elapses after the head is positioned over the correct track until the starting position of the addressed section passes under the read/write head.

10.3.2 Floppy Disk Memory

The magnetic disks that we described above are called hard-disks. These disks, in the form of disk-packs are suitable for storage of large amount of data. **Floppy disks** are smaller, simpler and cheaper disk units that consist of a flexible, removable, plastic diskettes coated with magnetic material. The diskette is enclosed in a plastic jacket which has an opening where the head makes contact with the diskette. A hole in the

diskette allows a spindle mechanism in the disk drive to position and rotate the diskette as shown in the Fig. 10.3.6.

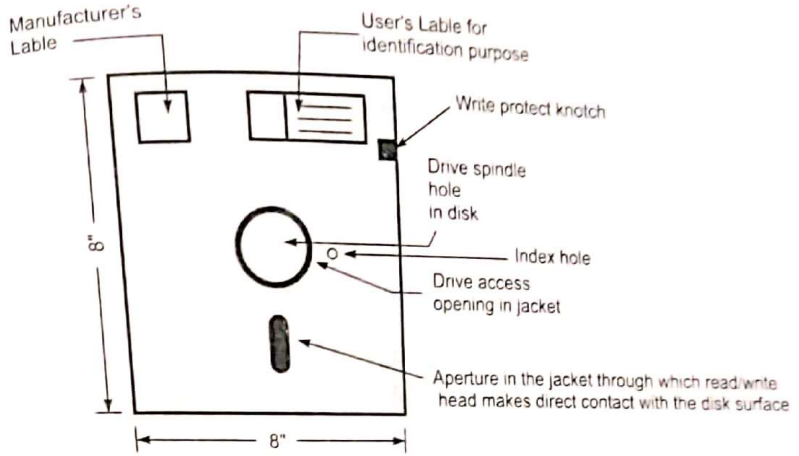


Fig. 10.3.6

10.3.2.1 Specifications of Floppy Disks

Physical specifications	Capacity
5.25 inch floppy disk, 1 sided	180 kB
5.25 inch floppy disk, 2 sided	360 kB
5.25 inch floppy disk, 2 sided	1.2 MB
3.5 inch floppy disk, 2 sided	1.44 MB

Table 10.3.3

10.3.2.2 Disk Format

The data on the disks is organized in a concentric set of rings. These concentric set of rings are called as tracks. Each track has same width as head and adjacent tracks are separated by gaps. The gap between two adjacent tracks prevents, or at least minimizes errors due to misalignment of the head or simply interference of magnetic fields. Data bits are stored serially on each track. The same number of bits are stored in each track. The same number of bits are stored on each track. Thus, the density in bits per linear inch increases as we move from the outermost track to the innermost track. The tracks are further divided into sectors. In a PC, the DOS organizes 9 sectors (8 sectors in old DOS version) on each track. Each sector stores 512 bytes of data which can b

transferred to or from the disk. To avoid magnetic interference between two adjacent sectors, a gap is introduced between two adjacent sectors.

Fig. 10.3.7 shows IBM 3740 floppy disk track format. As shown in the Fig. 10.3.7, each sector consists of ID field and a data field. The ID field contains the address of the sector. The data field contains the actual data stored in the sector. As we can see, besides the bytes used to store data, many bytes are used for track and sector identification, synchronization, error checking and buffering between sectors. Address marks shown at several places in this format are used to identify the start of the field.

(See Fig. 10.3.7 on next page).

10.3.2.3 Storage Density

The IBM 3740 format shown in Fig. 10.3.7 is set up for single density recording. An 8-in disk in this format has one index track and 76 data tracks. Since each track has 26 sectors with 128 data bytes in each sector, the total is about 250 kbytes. If double density recording is used, the capacity increases to about 500 kbytes. Table 10.3.4 shows the number of tracks, number of sectors and some other information for floppy disks commonly used with IBM PC.

Parameter	$3\frac{1}{2}$ Inch Disk		$5\frac{1}{4}$ Inch Disk	
	1.44 MB HD	720 K LD	1.2 MB HD	360 K DSDD
Sectors per track	18	9	15	9
Total number of sectors	2880	1440	2400	720
Number of tracks	80	80	80	40
Sectors per cluster	1	2	1	2
Allowable entries in root directory	224	112	224	112

Table 10.3.4 Floppy disks tracks, sectors and capacities

In IBM 3740 floppy disk format, in each sector 2 bytes are used to store CRC error checking code. These are used to check for errors when the ID and the data are read out. One way the 2 CRC bytes can be produced is to treat the 128 data bytes as a single large binary number and divide this number by a constant. The 16-bit remainder from this division is written in after the data bytes as the CRC bytes. When the data bytes and the CRC bytes are read out, the CRC bytes are subtracted from the data string. The result is divided by the original constant. Since the original remainder has already been subtracted, the result of the division should be zero if the data was read out correctly.

mark (first record) can be used as a *header or identifier* for this file. This allows user to identify a particular file from number of files recorded on the tape.

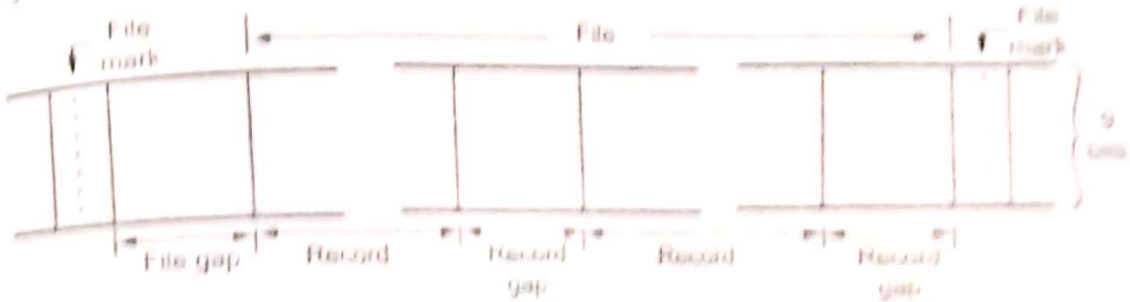


Fig. 10.3.10 Organization of data on magnetic tape

We know that, the data on a nine track tape is stored in parallel consists of data byte and a parity bit. Now-a-days tapes are available with several hundred tracks in which a read write head can simultaneously access all tracks. Data transfer takes place when the tape is moving at constant velocity relative to a read write head. Hence the maximum transfer data transfer rate depends largely on the storage density along the tape and the speed of the tape.

Advantages of magnetic tape

1. **Unlimited storage** : The storage capacity of magnetic tape is virtually unlimited because we can use as many tapes as required for recording.

2. **High data density** : A typical 10½ inch reel of magnetic tape is 2400 inches long and is able to hold upto 6250 characters per inch. Therefore, the maximum capacity of magnetic tape is about 180 million characters.

3. **Low cost** : The cost of magnetic tape is much less as compared to other storage devices.

4. **Rapid transfer rate** : Data transfer rate for a magnetic tape can be in excess of 1 million bytes per second.

5. **Ease of handling** : Tape is compact and light weighted it is easier to use it.

6. **Portability** : The tape is wound on a reel and it is packed in a plastic package, which is very convenient way of carrying information from one place to another.

Attribute	Magnetic tape	Magnetic disk
Access	Sequential	Direct or sequential
Access time	More	Less
Environmental restrictions	More	Less
Data transfer rate	Comparatively less	More
Reliability	Less	More
Portability	More	Less
Cost	Less	High
Application	Used for backups	Used as a secondary storage device in computer systems

10.4 Associative Memory

Many data-processing applications require the search of items in a table stored in emory. They use object names or number to identify the location of the named or umbered object within a memory space. For example, an account number may be arched in a file to determine the holder's name and account status. To search an ject, the number of accesses to memory depends on the location of the object and the ficiency of the search algorithm.

The time required to find an object stored in memory can be reduced considerably if jects are selected based on their contents, not on their locations. A memory unit essed by the content is called an **associative memory** or **Content Addressable emory (CAM)**. This type of memory is accessed simultaneously and in parallel on the sis of data content rather than by specific address or location.

10.4.1 Hardware Organization

The Fig. 10.4.1 shows the block diagram of an associative memory. It consists of mory array with match logic for m n-bit words and associated registers. The ument register (A) and key register (K) each have n-bits per word. Each word in mory is compared in parallel with the contents of the argument register. The words match with the word stored in the argument register set a corresponding bits in the h register. Therefore, reading can be accomplished by a sequential access to memory hose words whose corresponding bits in the match register have been set.

he key register provides a mask for choosing a particular field or bits in the ment word. Only those bits in the argument register having 1's in their

argument register A and the key register K have the bit configuration shown below. Only the three rightmost bits of A are compared with memory words because K has 1's in these positions.

A 1 1 0 1 1 0 1 0

K 0 0 0 0 0 1 1 1

Word 1 0 1 0 1 0 0 1 0 match

Word 2 1 1 0 1 1 1 0 0 no match

The Fig. 10.4.2 shows the associated memory with cells of each register. The cells in the memory array are

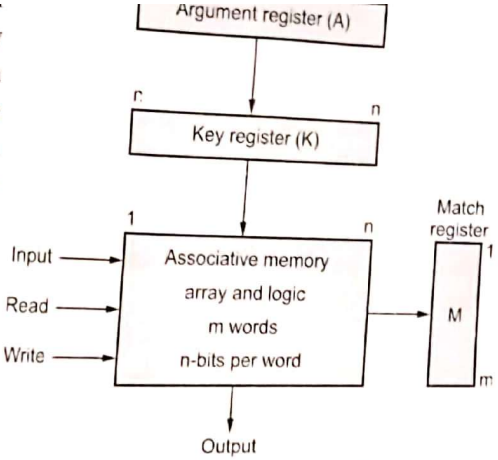
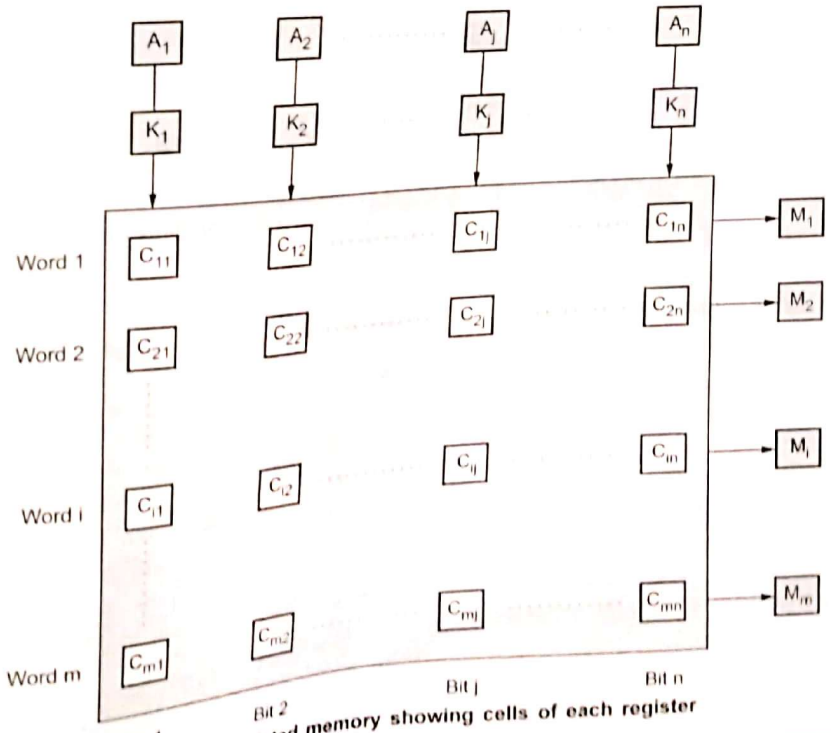


Fig. 10.4.1 Block diagram of associative memory



10.4.2 Associated memory showing cells of each register

Fig. 11.4.4 Match logic for one word of associative memory

11.4.2 Read Operation

In the read operation, all matched words are read in sequence by applying a read signal to each word line whose corresponding M_i bit is a logic 1. In applications where no two identical items are stored in the memory, only one word may match the unmasked argument field. In such case, we can use M_i output directly as a read signal for the corresponding word. The contents of the matched word will be presented automatically at the output lines and no special read signal is needed.

11.4.3 Write Operation

In write operation, we consider two separate cases : 1. It is necessary to load entire memory with new information 2. It is necessary to replace one word with new information. The entire memory can be loaded with new information by addressing each location in sequence. This will make the memory device a random access memory for writing and a content addressable memory for reading. Here, the address for the input can be decoded as in a random access memory. Thus instead of having m address lines, one for each word, the number of address lines can be reduced by the use of decoder to d lines, where $m = 2^d$

To implement the write operation in the second case the tag register is used. This register has as many bits as there are words in the memory for every active (valid) word stored in the memory, the corresponding bit in the tag register is set to 1. When word is deleted from the memory, the corresponding tag bit is cleared, i.e. it is set to logic 0. The word is then stored in the memory by scanning the tag register until the first 0 bit is encountered. This gives the first available inactive word and a position for writing a new word. After the new word is stored in the memory it is made active by setting the corresponding tag bit in the tag register.

Review Questions

1. *Write a short note on associative memory.*
2. *Describe by means of block diagram how multiple matched words can be read out from an associative memory.*
3. *What is CAM ? Explain it with block diagram.*
4. *Explain associative memory with its hardware organization. Explain how the data is read and write in the associative memory.*

10.5 Cache Memories

- In a computer system the program which is to be executed is loaded in the main memory (DRAM). Processor then fetches the code and data from the main memory to execute the program. The DRAMs which form the main memory are slower devices. So it is necessary to insert wait states in memory read/write cycles. This reduces the speed of execution.
- To speed up the process, high speed memories such as SRAMs must be used. But considering the cost and space required for SRAMs, it is not desirable to use SRAMs to form the main memory. The solution for this problem is come out with the fact that most of the microcomputer programs work with only small sections of code and data at a particular time.
- **Definition** : The part of program (code) and data that work at a particular time is usually accessed from the SRAM memory. This is accomplished by loading the active part of code and data from main memory to SRAM memory. This small section of SRAM memory added between processor and main memory to speed up execution process is known as **cache memory.**
- Fig. 10.5.1 shows a simplest form of cache memory system.
- A cache memory system includes a small amount of fast memory (SRAM) and a large amount of slow memory (DRAM). This system is configured to simulate a large amount of fast memory.
- Cache controller implements the cache logic. If processor finds that the addressed code or data is not available in cache - the condition referred to as **cache**

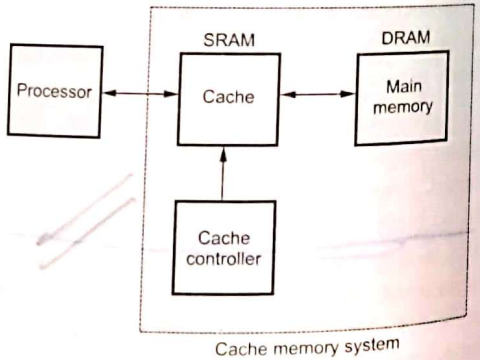


Fig. 10.5.1 Cache memory system

miss, the desired memory block is copied from main memory to cache using cache controller. The cache controller decides which memory block should be moved in or out of the cache and in or out of main memory, based on the requirements. (The cache block is also known as **cache slot or cache line**.)

- The percentage of accesses where the processor finds the code or data word it needs in the cache memory is called the **hit rate/hit ratio**. The hit rate is normally greater than 90 percent.

$$\text{Hit rate} = \frac{\text{Number of hits}}{\text{Total number of bus cycles}} \times 100\ \%$$

Example 10.5.1 *The application program in a computer system with cache uses 1400 instruction acquisition bus cycle from cache memory and 100 from main memory. What is the hit rate ? If the cache memory operates with zero wait state and the main memory bus cycles use three wait states, what is the average number of wait states experienced during the program execution ?*

Solution :

$$\text{Hit rate} = \frac{1400}{1400 + 100} \times 100 = 93.3333\ \%$$

$$\text{Total wait states} = 1400 \times 0 + 100 \times 3 = 300$$

$$\text{Average wait states} = \frac{\text{Total wait states}}{\text{Number of memory bus cycles}} = \frac{300}{1500} = 0.2$$

10.5.1 Mapping

- Usually, the cache memory can store a reasonable number of memory blocks at any given time, but this number is small compared to the total number of blocks in the main memory. The correspondence between the main memory blocks and those in the cache is specified by a **mapping function.**
- The mapping techniques are classified as :
  1. Direct-mapping technique
  2. Associative-mapping technique
     - **Fully-associative**
     - **Set-associative techniques.**
- To discuss these techniques of cache mapping we consider a cache consists of 128 blocks of 16 words each, for a total of 2048 (2 K) words and assume that the main memory has 64 K words. This 64 K words of main memory is addressable by a 16-bit address and it can be viewed as 4 K blocks of 16 words each. The group of 128 blocks of 16 words each in main memory form a page.

address are compared with the tag
match, then the desired word is in that block of the cache. If there is no match, then the block containing the required word must first be read from the main memory and loaded into the cache.

- This means that to determine whether requested word is in the cache, only tag field is necessary to be compared. This needs only one comparison.
- The main drawback of direct mapped cache is that if processor needs to access same memory locations from two different pages of the main memory frequently, the controller has to access main memory frequently. Since only one of these locations can be in the cache at a time. For example, if processor want to access memory location 100H from page 0 and then from page 2, the cache controller has to access page 2 of the main memory. Therefore, we can say that direct-mapped cache is easy to implement, however, it is not very flexible.

10.5.1.2 Associative-Mapping (Fully-Associative Mapping)

- The Fig. 10.5.3 shows the associative-mapping technique. In this technique, a main memory block can be placed into any cache block position. As there is no fix block, the memory address has only two fields : **word** and **tag**. This techniques is also referred to as **fully-associative cache.**
- The 12-tag bits are required to identify a memory block when it is resident in the cache. The high-order 12-bits of an

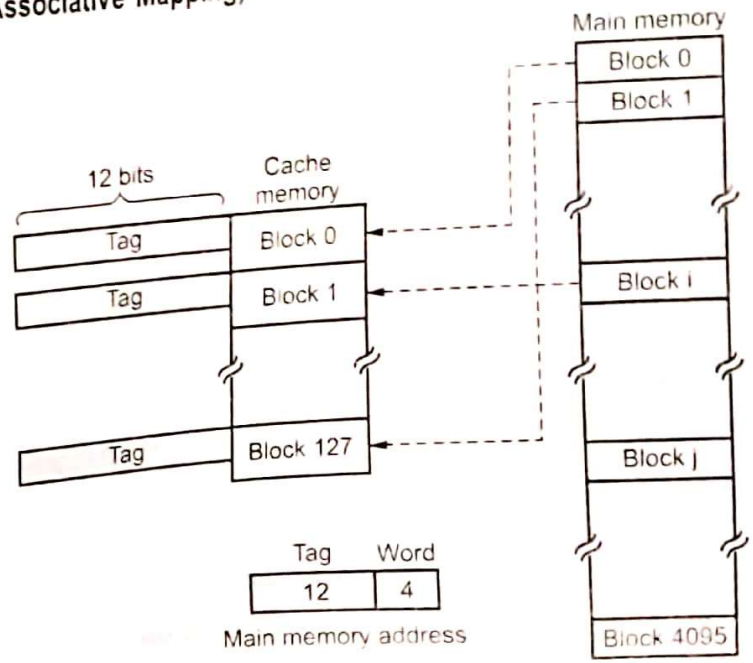


Fig. 10.5.3 Associative-mapped cache

address received from the CPU are compared to the tag bits of each block of the cache to see if the desired block is present.

- Once the desired block is present, the 4-bit word is used to identify the necessary word from the cache.
- This technique gives complete freedom in choosing the cache location in which to place the memory block. Thus, the memory space in the cache can be used more efficiently.
- A new block that has to be loaded into the cache has to replace (remove) an existing block only if the cache is full.
- In such situations, it is necessary to use one of the possible replacement algorithm to select the block to be replaced.
- **Disadvantage :** In associative-mapped cache, it is necessary to compare the higher-order bits of address of the main memory with all 128 tag corresponding to each block to determine whether a given block is in the cache. This is the main disadvantage of associative-mapped cache.

10.5.3 Set-Associative Mapping

- The set-associative mapping is a combination of both direct and associative mapping.
- It contains several groups of direct-mapped blocks that operate as several direct-mapped caches in parallel.
- A block of data from any page in the main memory can go into a particular block location of any direct-mapped cache. Hence the contention problem of the direct-mapped technique is eased by having a few choices for block placement.
- The required address comparisons depend on the number of direct-mapped caches in the cache system. These comparisons are always less than the comparisons required in the fully-associative mapping.
- Fig. 10.5.4 shows two way set-associative cache. Each page in the main memory is organized in such a way that the size of each page is same as the size of one directly mapped cache. It is called two-way set-associative cache because each block from main memory has two choices for block placement.
- In this technique, block 0, 64, 128,.....4032 of main memory can map into any of the two (block 0) blocks of set 0, block 1, 65, 129,...., 4033 of main memory can map into any of the two (block 1) blocks of set 1 and so on.
- As there are two choices, it is necessary to compare address of memory with the tag bits of corresponding two block locations of particular set. Thus for two-way

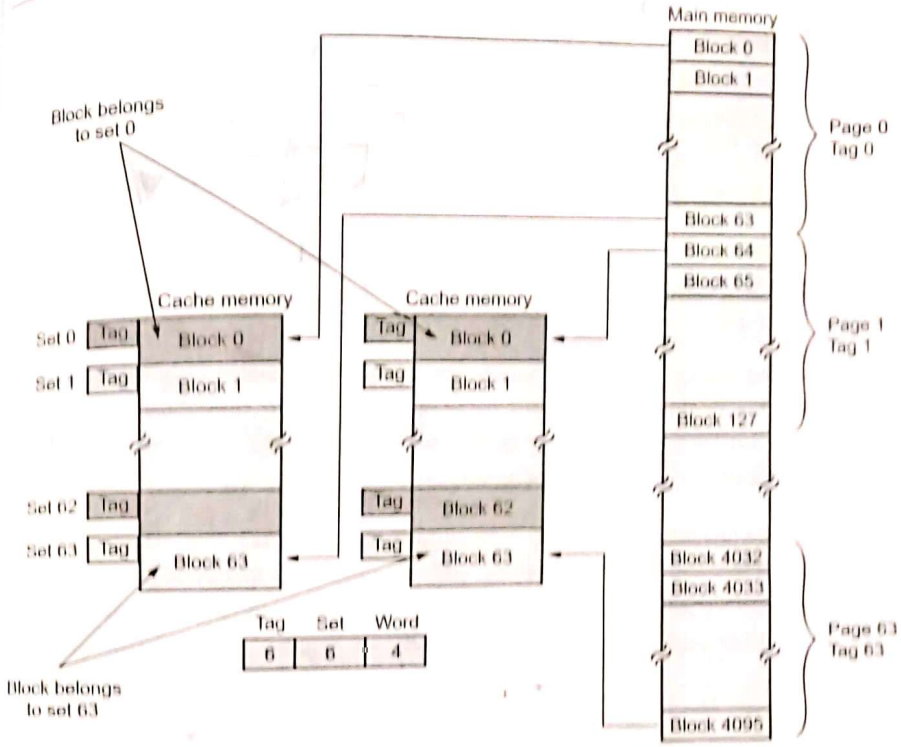


Fig. 10.5.4 Two-way set-associative cache

set-associative cache, we require two comparisons to determine whether a given block is in the cache.

- Since there are two direct-mapped caches, any two bytes having same offset from different pages can be in the cache at a time. This **improves the hit rate** of the cache system.
- To implement set-associative cache system, the address is divided into three fields, as shown in Fig. 10.5.4.
- The 4-bit word field selects one of the 16 words in a block.
- The set field needs 6-bits to determine the desired block from 64 sets. However, there are now 64 pages. To identify a block belongs to a particular page from 64 pages, six tag bits are required.

Example 10.5.2 *Consider a cache consisting of 256 blocks of 16 words each, for a total of 4096 (4 K) words and assume that the main memory is addressable by a 16-bit address and it consists of 4 K blocks. How many bits are there in each of the TAG, BLOCK/SET and word fields for different mapping techniques ?* **PU : May-07, Dec.-12**

Solution : We know that memory address is divided into three fields. We will now find the exact bits required for each field in different mapping techniques.

a) Direct-mapping

Word bits : We know that each block consists of 16 words. Therefore, to identify each word we must have (2^4 = 16) four bit reserved for it.

Block bits : The cache memory consists of 256 blocks and using direct-mapped technique block k of the main memory maps onto block k modulo 256 of the cache. It has one to one correspondence and requires unique address for each block. To address 128 block we require (2^8 = 256) eight bits.

Tag bits : The remaining 4 (16 – 4 – 8) address bits are tag bits which stores the higher address of the main memory.

The main memory address for direct-mapping technique is divided as shown below :

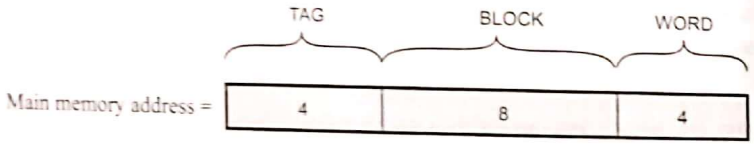


Fig. 10.5.5 (a)

b) Associative-mapping

Word bits : The word length will remain same i.e. 4 bits.

In the associative-mapping technique, each block in the main memory is identified by the tag bits and an address received from the CPU is compared with the tag bits of each block of the cache to see if the desired block is present. Therefore, this type of technique does not have block bits, but all remaining bits (except word bits) are reserved as tag bits.

Block bits : 0

Tag bits : To address each block in the main memory (2^{12} = 4096) 12 bits are required and therefore, there are 12 tag bits.

The main memory address for direct mapping technique is divided as shown below :

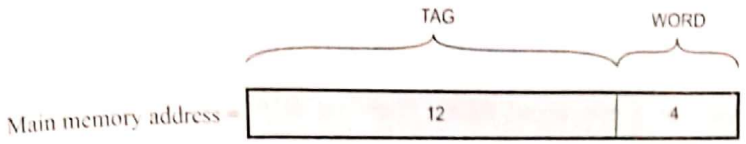


Fig. 10.5.5 (b)

c) Set-associative mapping

Let us assume that there is a 2-way set-associative mapping. Here, cache memory is mapped with two blocks per set. The set field of the address determines which set of the cache might contain the desired block.

Word bits : The word length will remain same i.e. 4 bits

Set bits : There are 128 sets (256/2). To identify each set (2^7 = 128) seven bits are required.

Tag bits : The remaining 5 (16 – 4 – 7) address bits are the tag bits which stores higher address of the main memory.

The main memory address for 2-way set associative mapping technique is divided as shown below :

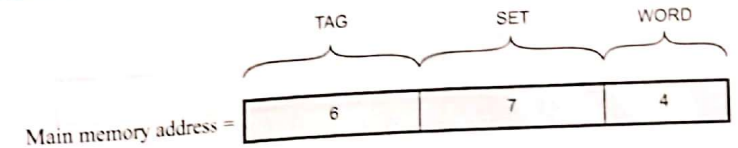


Fig. 10.5.6

Example 10.5.3 *A block set-associative cache consists of 64 blocks divided into 4 block sets. The main memory contains 4096 blocks, each consists of 128 words of 16 bits length :*

i) How many bits are there in main memory ?

ii) How many bits are there in each of the TAG, SET and WORD fields ? **PU : May-05, Dec.-10**

Solution : i) Number of bits in main memory :

= Number of blocks × Number of words per block × Number of bits per word

= 4096 × 128 × 16

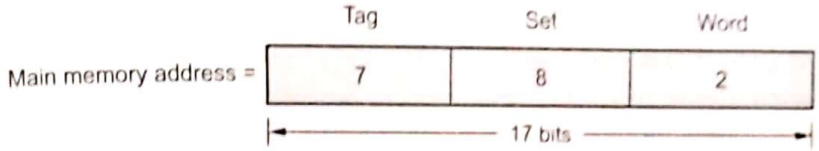
= 8388608 bits

Number of sets in the cache memory = $\frac{512}{2}$ = **256 sets**

Number of bits in set field = $\log_2 256 = \log_2 2^8$ = **8 bits**

Number of bits in word field = $\log_2 4 = \log_2 2^2$ = **2 bits**

Number of bits in tag field = 17 – 8 – 2 = **7 bits**



Example 10.5.6 *A direct mapped cache has the following parameters : cache size = 1 K words, Block size = 128 words and main memory size is 64 K words. Specify the number of bits in TAG, BLOCK and WORD in main memory address.* **PU : May-10**

Solution : Word bits = 109_2 128 = **7-bits**

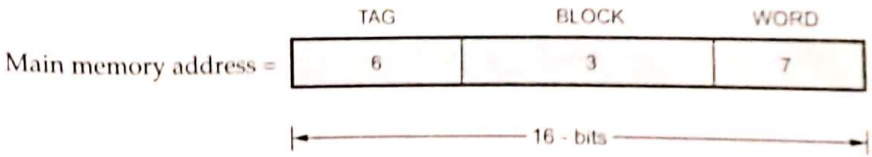
$$\text{Number of blocks} = \frac{\text{Cache size}}{\text{Words in each block}} = \frac{1\,K}{128} = 8$$

∴ Number of block bits = $\log_2 8$ = **3-bits**

Number of address bits to address main memory

= $\log_2 64\,K = \log_2 2^{16}$ = 16 - bits

∴ Tag bits = 16 – 3 – 7 = **6-bits**



10.5.1.4 Comparison between Mapping Techniques

Sr.No.	Direct-mapping	Associative-mapping	Set-associative-mapping
1.	Each block from the main memory has only one possible location in the cache	A block of data from main memory can be placed into any cache block position.	A block of data from main memory can go into a particular block location of any direct-mapped cache.
2.	Needs only one comparison	Needs comparison with all tag bits.	Needs number of comparisons equal to number of blocks per set

- **Bus watching** : In bus watching, cache controller invalidates the cache entry, if another master writes to a location in shared memory which also resides in the cache memory. Fig. 10.5.7 shows bus watching.
- **Hardware transparency** : In hardware transparency, accesses of all devices to the main memory are routed through the same cache or by copying all cache writes both to the main memory and to all other caches that share the same memory. Fig. 10.5.8 shows hardware transparent system.
- **Non-cacheable memory** : The 80386DX can partition its main memory into a cacheable and non-cacheable memory. By designing shared memory as non-cacheable memory cache coherency can be maintained, since shared memory is never copied into cache.

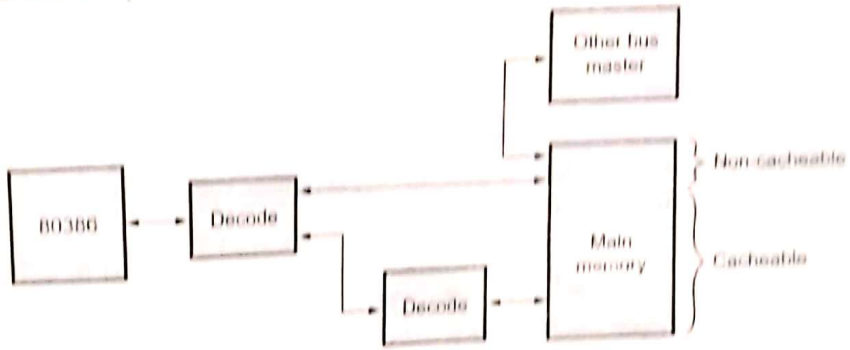


Fig. 10.5.9 Memory

- **Cache flushing** : To avoid data inconsistency, a cache flush writes any altered data to the main memory and caches in the system are flushed before a device writes to shared memory.

10.5.3 Writing into Cache

- In a cache system, two copies of same data can exist at a time, one in cache and one in main memory. If one copy is altered and other is not, two different sets of data become associated with the same address. To prevent this, the cache system has updating systems such as : **write through system, buffered write through system** and **write-back system.**

10.5.3.1 Write through System

- The cache controller copies data to the main memory immediately after it is written to the cache. Due to this, main memory always contains a valid data and thus any block in the cache can be overwritten immediately without data loss.
- The write through is a **simple approach.**

- This approach **requires time to write data** in main memory with increase in bus traffic.
- This in effect **reduces the system performance**.

10.5.3.2 Buffered Write through System

- In buffered write through system, the processor can start a new cycle before the write cycle to the main memory is completed. This means that the write accesses to the main memory are buffered.
- In such systems, read access which is a "cache hit" can be performed simultaneously when main memory is updated.
- However, two consecutive write operations to the main memory or read operation with cache "miss" require the processor to wait.

10.5.3.3 Write-Back System

- In a write-back system, the **alter** (update) bit in the tag field is used to keep information of the new data. If it is set, the controller copies the block to main memory before loading new data into the cache.
- Due to one time write operation, number of write cycles are reduced in write-back system. But this system has following disadvantages.
  - Write-back cache controller logic is more complex.
  - It is necessary that, all altered blocks must be written to the main memory before another device can access these blocks in main memory.
  - In case of power failure, the data in the cache memory is lost, so there is no way to tell which locations of the main memory contain old data. Therefore, the main memory as well as cache must be considered volatile and provisions must be made to save the data in the cache.

10.5.3.4 Replacement Algorithms

- When a new block is brought into the cache, one of the existing blocks must be replaced, by a new block.
- In case of direct-mapping cache, we know that each block from the main memory has only one possible location in the cache, hence there is no choice. The previous data is replaced by the data from the same memory location from new page of the main memory.
- For associative and set-associative techniques, there is a choice of replacing existing block. The choice of replacement of the existing block should be such that the probability of accessing same block must be very less. The replacement algorithms do the task of selecting the existing block which must be replaced.

10. Consider a cache consisting of 256 blocks of 16 words each for a total of 4096 (4 K) words and assume that the main memory is addressable by a 16 bit address and it consists of 4 K blocks. How many bits are there in each of the TAG, BLOCK/SET and WORD fields for different mapping techniques ?

11. A direct mapped cache has the following parameters : Cache size = 1 K words, Block size = 128 words and main memory size is 64 K words. Specify the number of bits in TAG, BLOCK and WORD in main memory address.

12. Explain the following terms : i) Cache updation policies ii) Cache hit and cache miss.

13. Discuss in detail hardware cache coherent schemes.

14. What are the different replacement algorithms ? Explain LRU algorithm in detail.

15. List and explain write policies used with cache memory.

16. What is cache coherence ?

17. Explain in detail various cache memory organizations.

18. Explain set associative mapping of cache memory.

10.6 Virtual Memory

In most modern computers, the physical main memory is not as large as the address space spanned by an address issued by the processor. Here, the virtual memory technique is used to extend the apparent size of the physical memory. It uses secondary storage such as disks, to extend the apparent size of the physical memory. Let us see how this technique works.

When a program does not completely fit into the main memory, it is divided into segments. The segments which are currently being executed are kept in the main memory and remaining segments are stored in the secondary storage devices, such as a magnetic disk. If an executing program needs a segment which is not currently in the main memory, the required segment is copied from the secondary storage device. When a new segment of a program is to be copied into a main memory, it must replace another segment already in the memory. In modern computers, the operating system moves program and data automatically between the main memory and secondary storage. Techniques that automatically swaps program and data blocks between main memory and secondary storage device are called **virtual memory**. The addresses that processor issues to access either instruction or data are called **virtual** or **logical address**. The set of such addresses is called **address space.** Thus the address space is the set of addresses generated by programs as they reference instruction and data. These addresses are translated into physical addresses by a combination of hardware and software components. The physical address is an address of main memory. The set of such addresses is called the **memory space.** The memory space consists of the actual main memory locations directly addressable for processing. The address space is allowed to be larger than the memory space in computers with virtual memory. If a virtual address

refers to a part of the program or data space that is currently in the main memory, then the contents of the appropriate location in the main memory are accessed immediately. On the other hand, if the referenced address is not in the main memory, its contents must be brought into a suitable location in the main memory before they can be used.

We have seen that, how virtual memory removes the programming burdens of a small, limited amount of main memory. Along with this it also allows efficient and safe sharing of memory among multiple programs. Consider a number of programs running at once on a computer. The total memory required by all the programs may be much larger than the amount of main memory available on the computer, but only a fraction of this memory is actively being used at any point in time. The main memory need to contain only the active portions of the many programs. This allows use to efficiently share the processor as well as the main memory.

10.6.1 Address Mapping using Pages

Let us assume that address space is 8 K and memory space is 4 K. For mapping purpose, address space and memory space are divided into group of equal sizes. Let us assume size is 1 K words. Thus address space is divided into 8 equal groups called **pages** and memory space is divided into 4 equal groups called **blocks.** This is illustrated in Fig. 10.6.1.

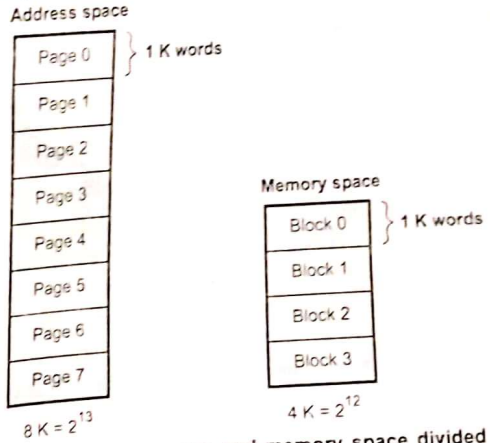


Fig. 10.6.1 Address space and memory space divided into groups of 1 K words

Example 10.6.1 *An address space is specified by 32 bits and corresponding memory space by 24 bits.*

i) How many words are there in the address space ?

ii) How many words are there in the memory space ?

iii) If a page consists of 4 K words, how many pages and blocks are there in the systems.

RGTU : May-05

Solution :

i) Words in the address space = 2^{32} = 4 G words

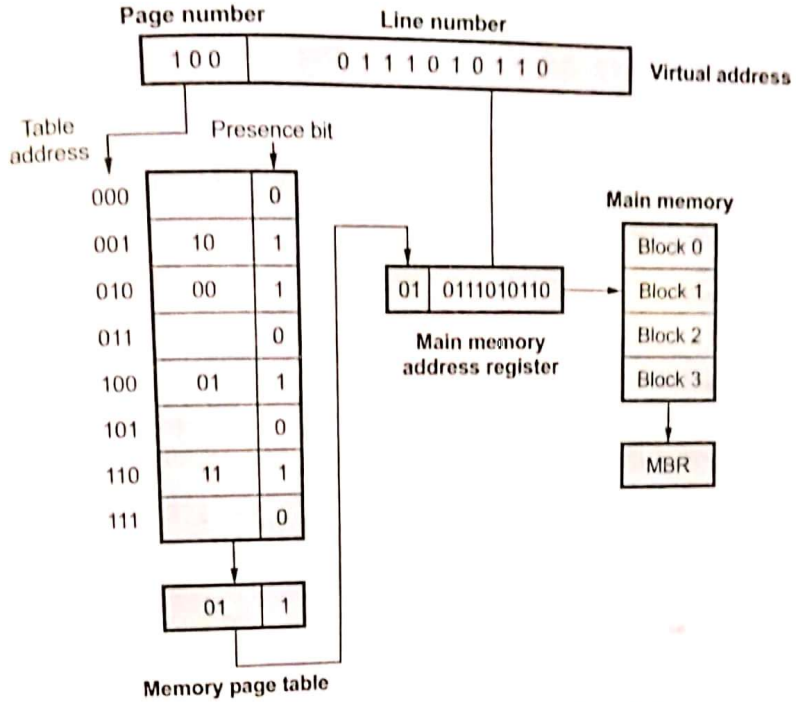


Fig. 10.6.2 Memory table in the paged system

FIFO Algorithm

It is the simplest page-replacement algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For our example reference string, our three frames are initially empty. The first three references (6, 0, 1) cause page faults, and the required pages are brought into these empty frames.

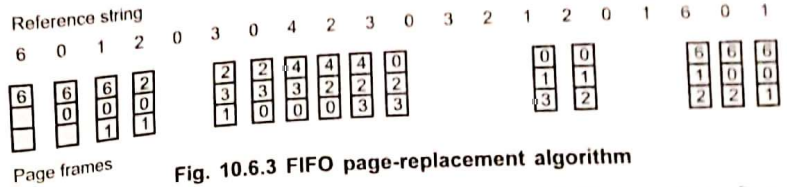


Fig. 10.6.3 FIFO page-replacement algorithm

The next reference (2) replaces page 6, because page 6 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in page 0 being replaced, since it was the first of the three pages in memory (0, 1 and 2) to be brought in. This replacement means that the next

reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Fig. 10.6.3. Every time a fault occurs, we show which pages are in out three frames. There are 15 faults algorithm.

The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. It may replace the most needed page as it is oldest page.

Optimal Algorithm

An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms. It has been called OPT or MIN. It simply states that "*Replace the page that will not be used for the longest period of time*". Use of this page-replacement algorithm guarantees the lowest possible page-fault rate for a fixed number of frames.

For example, on our sample reference string, the optimal page replacement algorithm would yield nine page faults, as shown in Fig. 10.6.4. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 6, because it will not be used until reference 18, whereas page 0 will be used at 5 and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which had 15 faults. (If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with less than nine faults.

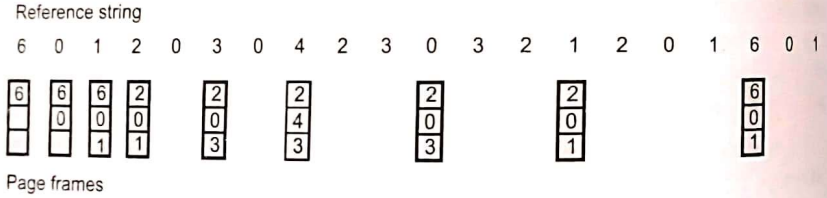


Fig. 10.6.4 Optimal page replacement algorithm

Unfortunately, the optimal page replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

LRU Algorithm

The algorithm which replaces the page that has not been used for the longest period of time is referred to as the *Least Recently Used* (LRU) algorithm.

The result of applying LRU replacement to our example reference string is shown in Fig. 10.6.5. (See Fig. 10.6.5 on next page). The LRU algorithm produces 12 faults. It can be noticed that the first five faults are the same as the optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in

memory, page 2 was used least recently. The most recently used page is page 0, and just before that page 3 was used. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3 since, of the three pages in memory {0, 3, 4}, page 3 is the least recently used. Despite these problems, LRU replacement with 12 faults is still much better than FIFO replacement with 15.

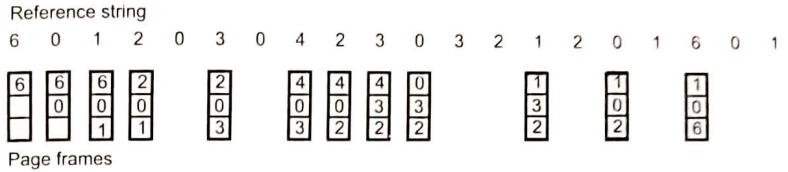


Fig. 10.6.5 LRU page replacement algorithm

Example 10.6.2 *A virtual memory system has an address space of 8 K words, a memory space of 4 K words and pages of 4 K words and page and block size of 1 K words. The following page references changes occur during a given time interval.*

4, 2, 0, 1, 2, 6, 1, 4, 0, 1, 0, 2, 3, 5, 7

Determine the four pages that are resident in main memory after each page reference change if the replacement algorithm used is

i) LRU ii) LFU iii) FIFO **RGTU : May-09, June-10**

Solution : The Fig. 10.6.6 shows the address space and memory space split into groups of 1K words. At any given time, up to four pages of address space may reside in main memory in any one of the four blocks.

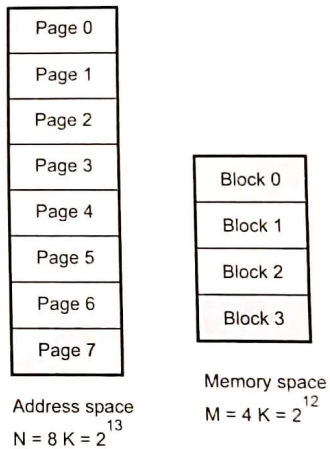
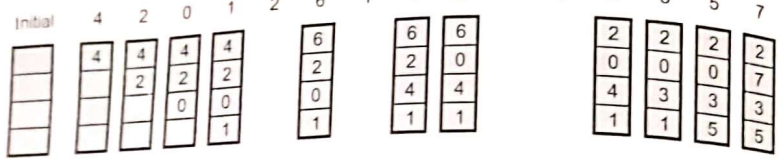
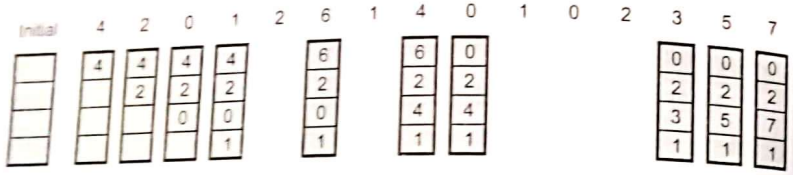


Fig. 10.6.6 Address space and memory space

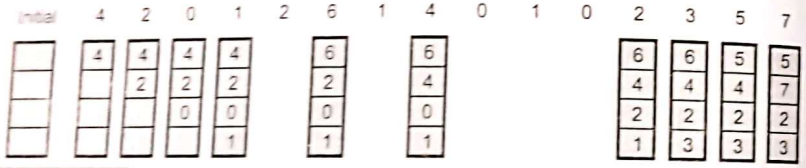
i) LRU : Least Recently Used



ii) LFU : Least Frequently Used



iii) FIFO : First in First Out



Review Questions

1. *What is virtual memory ? Why is it necessary to implement virtual memory ?*
2. *Draw and explain the virtual memory organization.*
3. *What is the use of virtual memory in computer system ? Explain in detail, how the program is run through virtual memory.*
4. *What is paging ? Explain how paging can be implemented in CPU to access virtual memory.*
5. *Write a note on virtual memory.*
6. *What is the need of virtual memory in the computer system ? Explain how the page map table is organized in virtual memory system.*
7. *Explain the role of virtual memory.*
8. *What is page fault ? What techniques are used to handle page fault ?*
9. *Explain the concept of address space and memory space in virtual memory.*

❑❑❑

11 Multiprocessors

Syllabus

Characteristics of multiprocessors, Interconnection structures, Inter-Processor arbitration, Inter-Processor communication and synchronization, Cache coherence, Shared memory multiprocessors.

Contents

11.1 Characteristics of Multiprocessors
11.2 Interconnection Structures
11.3 Interprocessor Arbitration
11.4 Interprocessor Communication and Synchronization
11.5 Cache Coherence
11.6 Shared Memory Multiprocessors

If a microprocessor system contains two or more components that can execute instructions independently then the system is called **multiprocessor** system. Multiprocessor system uses a distributed approach. Here, more than one processors are used to do the subtasks instead of doing entire task by a single processor. This system has following advantages :

1. Improves cost/performance ratio of the system.
2. Several processors may be combined to fit the needs of an application while avoiding the expense of the unneeded capabilities of a centralized system. Yet this system provides room for expansion.
3. Tasks are divided among the modules. If failure occurs, it is easier and cheaper to find and replace the malfunctioning processor than replacing the failing part of complex processor.
4. In Multiprocessor systems, a failure or error in one part has a limited effect on the rest of the system. If fault occurs in one processor, a second processor can take the responsibility of performing the task of the failure processor. This improves the reliability of the system at the cost of some loss in the efficiency.

Multiprocessors can be grossly characterized by two main attributes :

- Multiprocessor is a single computer that includes multiple processors and
- Processors in the multiprocessor system may communicate and cooperate at different levels in solving a given problem. The communication between them may occur by sending messages from one processor to the other or by sharing a common memory.

Multiprocessors and multicomputer systems have same basic goal of performing the concurrent operations in the system. However, there exists an important distinction between multiple computer systems and multiprocessors. The distinction is based on the extent of resource sharing and cooperation in the solution of a problem. A multiple computer system consists of several autonomous computers which may or may not communicate with each other. On the other hand, a multiprocessor system is controlled by one operating system which provides interaction between processors and their programs at the process, data set, and data element levels.

The Fig. 11.1.1 shows typical multiprocessor organization. It consists of two or more processors of approximately comparable capabilities. All processor share access to common sets of memory modules, input-output channels, and peripheral devices. Along with shared memory and input-output devices, each processor has its own local memory and input-output devices. The interprocessor communication is done through the shared memories or through an interrupt network. Most importantly, the entire system is

communicate from one processor to the other is on the order of the bandwidth of the memory. One of the limiting factors to the expansion of a CCS is the performance degradation due to memory contentions which occur when two or more processors attempt to access the same memory unit concurrently. However, when high-speed or real-time processing is desired, Closely Coupled Systems (CCS) may be used. There are two models of a CCS :

1. CCS without private cache
2. CCS with private cache.

11.1.1.1 CCS without Private Cache

The Fig. 11.1.2 shows the closely coupled multiprocessor system without private cache. It consists of P processors, M memory modules and C input-output channels. These units are connected through a set of three interconnection networks, viz.

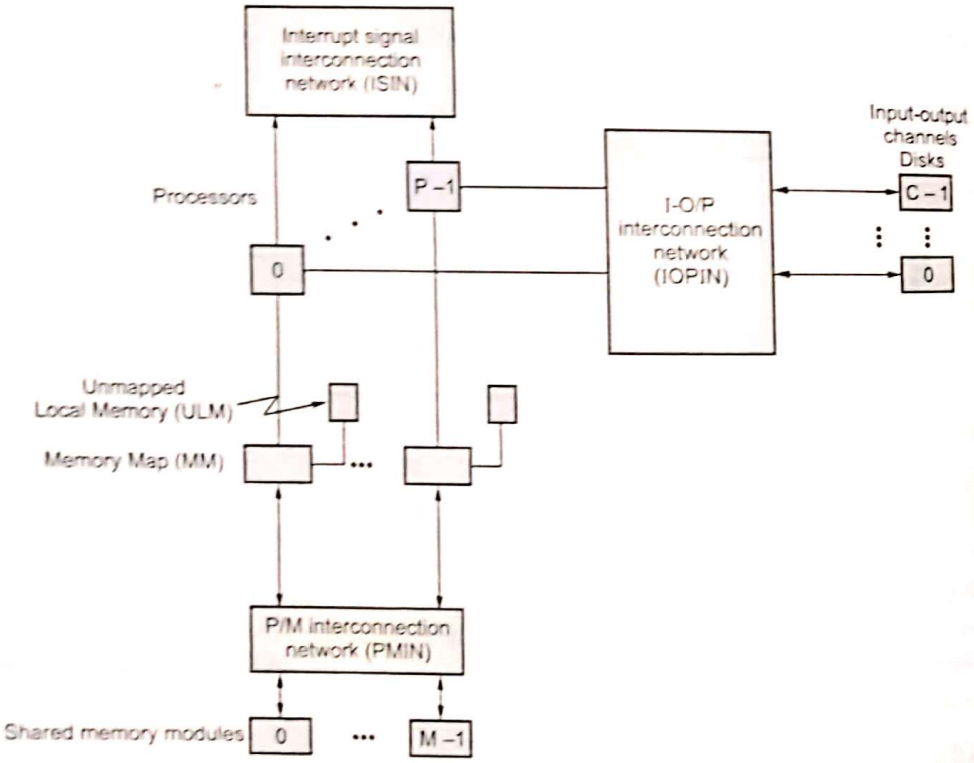


Fig. 11.1.2 CCS without private cache

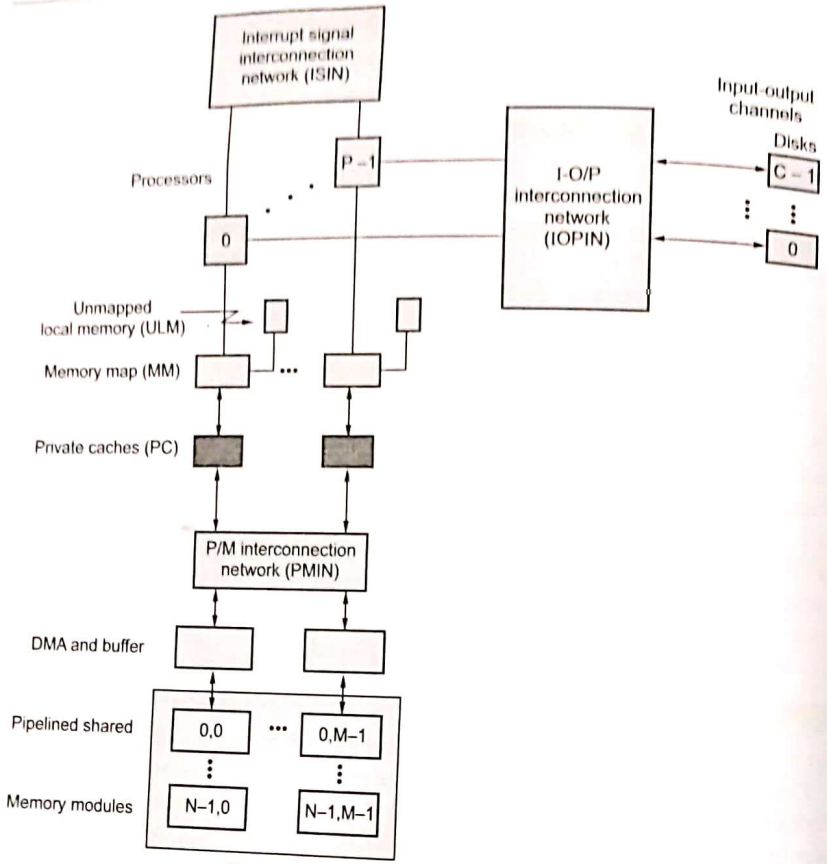
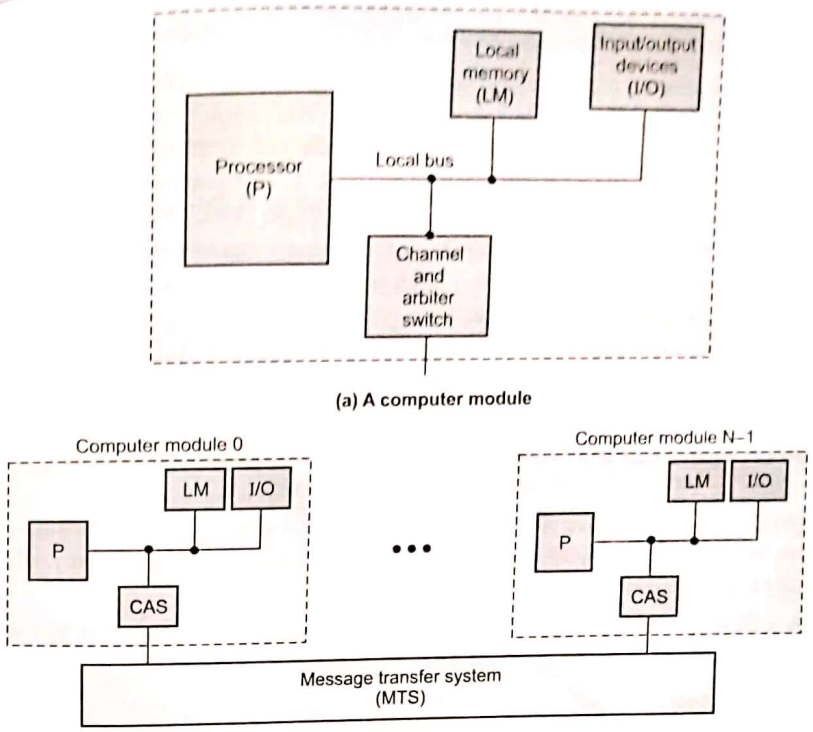


Fig. 11.1.3 CCS with private cache

loose. Hence, such systems are also referred to as a **distributed systems**. These systems are usually efficient when the interactions between tasks are minimal.

As shown in Fig. 11.1.4, a computer module of a nonhierarchical loosely coupled multiprocessor system. It consists of various computer modules connected to the message transfer system. Each computer module consists of a processor, a local memory, local input-output devices and an interface to other computer modules. The interface to the other module usually consists of Channel and Arbiter Switch (CAS).

The computer modules initiate communication with other modules by requesting MTS for the service. In case of simultaneous requests, the arbiter is responsible for choosing one of the simultaneous requests according to a given service discipline. The arbiter is also responsible for delaying other requests until the servicing of the selected



(a) A computer module

(b) Loose coupling of computer modules

Fig. 11.1.4 Nonhierarchical loosely coupled multiprocessor system

request is completed. The channel within the CAS usually consists of a high speed communication memory which is used for buffering block transfers of messages. It is accessible to all the processors.

The message-transfer systems for a nonhierarchical LCS are divided into two categories : 1. Simple time shared bus or 2. Shared memory system.

The time shared bus is a common communication path connecting all functional units. On the other hand shared memory system consists of a set of memory modules and a processor-memory interconnection network or a multiported main memory. In a multiported memory system, the arbitration and selection logic for the access control are distributed into the memory modules. The configuration of MTS is one of the most important factors that determine the performance of the multiprocessor system. In LCS configurations with a single time shared bus, the performance is limited by message arrival rate on the bus, the message length and bus capacity (in bits/sec). In LCS that

- Cross bar switch
- Multiport memory
- Multistage switching network
- Hypercube system

11.2.1 Time Shared Bus or Common Bus

In a time shared bus interconnection system for multiple processors provides a common communication path connecting all of the functional units. The Fig. 11.2.1 shows single bus multiprocessor organization with a common communication path. The organization shown in Fig. 11.2.1 is simplest and easiest to reconfigure. Such interconnection networks do not have any active components; they are totally passive. In these networks, transfer operations are controlled completely by the bus interfaces of the sending and receiving units.

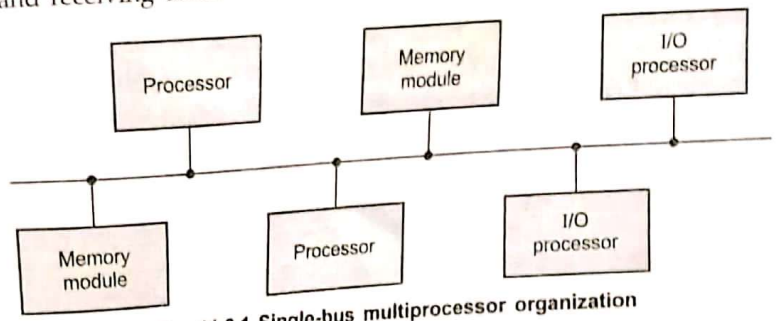


Fig. 11.2.1 Single-bus multiprocessor organization

The conflict-resolution methods such as static or fixed priorities, first-in-first-out (FIFO) queues and daisy chains are used to decide the access to the common bus. The example of PDP-11 Unibus. It has 56 signal lines to provide the control, address lines and data paths necessary to transfer 16-bit word.

The advantage of single-bus organization is that it is quite reliable and relatively inexpensive. However, it can cause complete system failure as a result of malfunctioning in any of the bus interface circuits.

Such organization is that as number of processors and memory increases, the chances of the bus contention problem also increases. This decreases system throughput and increases arbitration logic. To solved some of the problems mentioned above without an appreciable increase in system complexity or decrease in reliability usually two unidirectional buses are used, as shown in Fig. 11.2.2. However, a single transfer operation in such a system usually requires the use of both buses.

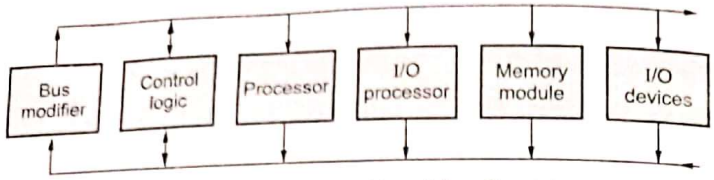


Fig. 11.2.2 Multiprocessor with unidirectional buses

The increase in system complexity and decrease in reliability due to two buses suggested to use multibus bidirectional buses as shown in Fig. 11.2.3.

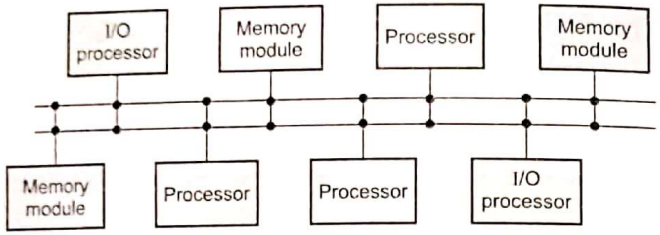


Fig. 11.2.3 Multi-bus multiprocessor organization

Multibus permits multiple simultaneous bus transfers but increases in system complexity. Along with the organization there are several factors that affect the characteristics and performance of a bus. These factors are :

- Number of active devices on the bus
- The bus-arbitration algorithm
- Centralization or distribution of control
- Data width
- Synchronization of data transmission and
- Error detection

11.2.2 Cross Bar Switch

If the number of buses in a common bus system is increased, a point is reached at which there is a separate path available for each memory module. Fig. 11.2.4 shows the crossbar switch system organization for multiprocessors which provides separate path for each memory

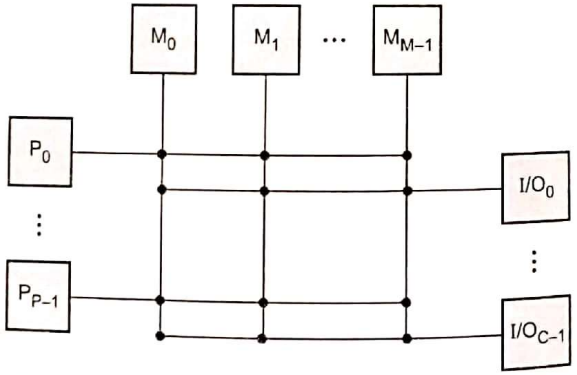


Fig. 11.2.4 Crossbar switch system organization for multiprocessors

module. The interconnection network shown in Fig. 11.2.4 is called **nonblocking** crossbar.

The crossbar switch has a separate bus associated with each memory module. Therefore, the maximum number of transfers that can take place simultaneously is limited by the number of memory modules and the bandwidth-speed product of the buses rather than the number of paths available.

Important characteristics of a system utilizing a crossbar matrix are :

1. Extreme simplicity of the switch-to-functional unit interfaces.
2. Ability to support simultaneous transfers for all memory units.

To provide these features, it requires hardware capabilities which are capable of switching parallel transmissions and capable of resolving multiple requests for access to the same memory module occurring during a single memory cycle.

The Fig. 11.2.5 shows the functional structure of a crosspoint in a crossbar network. It consists of multiplexer, memory and arbitration modules. Each processor activates a memory module request signal (REQ) to arbitration unit. The arbitration module selects the processor with highest priority using a priority encoder. After selecting a processor, arbitration module return an acknowledge signal (ACK) to it. On receiving the ACK signal, processor initiates the memory operation. Multiplexer module is used to connect data, control and address signals from the selected processor to the memory module.

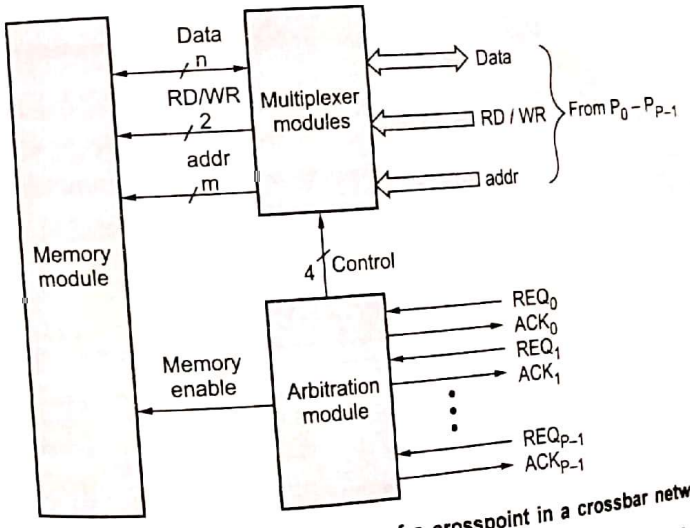


Fig. 11.2.5 Functional structure of a crosspoint in a crossbar network

The required flexibility in access to the input/output devices can be provided by using a switch on the device side of the input/output processor or channel.

important points

Sr. No.	Time shared bus	Crossbar switch	Multiport memory
1.	Lowest overall system cost for hardware and least complex	Most complex. Highest total transfer rate.	Most expensive since most of control and switching circuitry is in the memory unit.
2.	Easy to modify hardware system configuration.	Functional units are simplest and cheapest.	Functional units permit low cost uniprocessor.
3.	Overall system capacity limited by bus transfer rate. Bus fail then whole system fail.	Usually cost-effective for multiprocessors only as basic switching matrix is required to assemble any functional units into a working configuration.	Potential for a very high total transfer rate in overall system.
4.	Expanding may degrade system performance.	System expansion improves overall performance.	Difficult to modify as the design decision is made quite early.
5.	Lowest efficiency.	Expansion of system is limited only by size of the switch matrix.	Large number of cables and connector are required.
6.	Organization is suitable for smaller systems only.	Reliability of system can be improved by segmentation or redundancy within switch.	

Table 11.2.1

11.2.4 Multistage Switching (Binary Switching) Networks

In multistage switching networks a 2 × 2 crossbar switch is used as a basic component. It has two inputs, labeled A and B, and two outputs, labeled 0 and 1. The control inputs C_A and C_B associated with the switch establish the connection between the input and output terminals as shown in the Fig. 11.2.8. If control input is zero it connects the input to the 0 output and if it is one it connects the input to the 1 output. The 2 × 2 switch also has the capability to arbitrate between conflicting requests. If both inputs A and B require the same output terminal, then only one of them will be connected and the other will be blocked or rejected. (Refer Fig. 11.2.8 on next page).

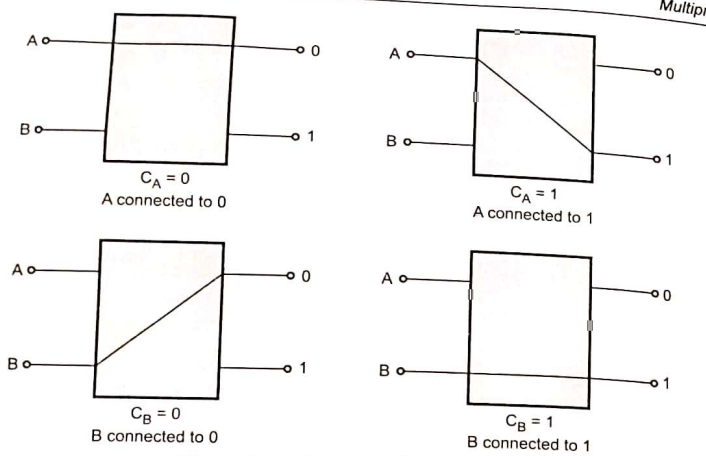


Fig. 11.2.8 2 × 2 crossbar switch

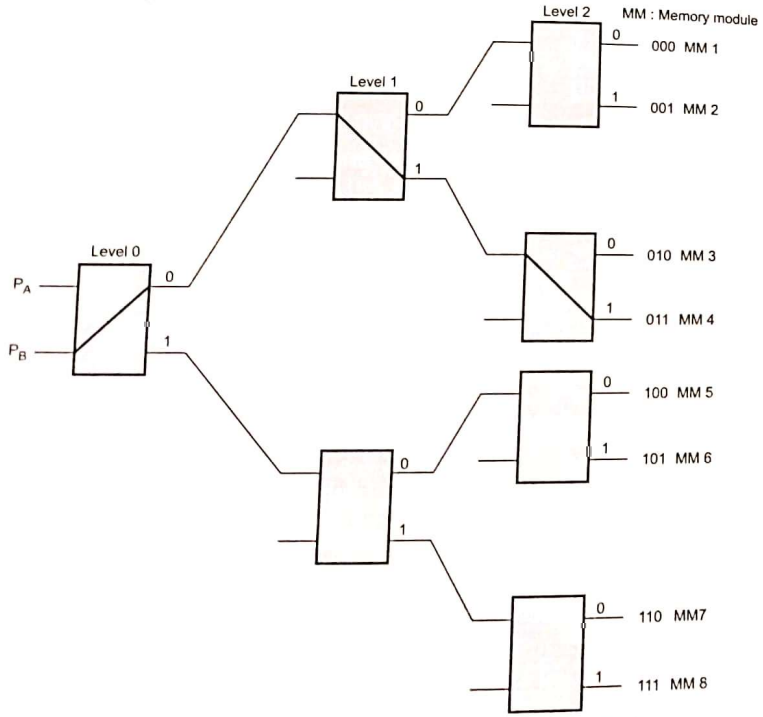


Fig. 11.2.9 1 to 8 way switch using basic 2 × 2 switch

Using the 2 × 2 crossbar switch as a building block, it is straightforward to construct a multistage network to control the communication between a number of sources and destinations as shown in the Fig. 11.2.9. The connections are accomplished by constructing a **binary tree** of cross bar switches. With this connection it is possible to connect the input to one of eight possible destinations. Consider that two inputs are the two processors P_A and P_B and they are connected through switches to eight memory modules marked in binary from 000 through 111. The path from source to destination goes through three levels. One binary bit is assigned to each level to choose output in that level. The first bit of the 3-bit destination number determines the switch output in the first level, the second bit specifies the output of the switch in the second level, and the third bit specifies the output of switch in the third level. For example, to connect P_B to memory module 011, it is necessary to form a path from P_B to output 0 in first level, output 1 in the second level and output 1 in the third level.

Many different patterns of 2 × 2 switches can be constructed to form different multistage networks such as omega network, butterfly network and so on. Let us see the construction of omega network, as shown in Fig. 11.2.10. As shown in the Fig. 11.2.10, four possible connections of 2 × 2 switches used in constructing the 8 × 8 omega network. Three levels of 2 × 2 switches are needed to construct 8 × 8 network. In general, an n-input omega network has $\log_2^n$ levels. The levels are labeled from 0 to $\log_2^{n-1}$ from the input end to the output end. Data routing is controlled by inspecting the destination code in binary. When the ith high-order bit of the destination code is a 0, a 2 × 2 switch at level i connects the input to the upper output. Otherwise, the input is directed to the lower output. (See Fig. 11.2.10 on next page).

In this network, it can be observed that, there is exactly one path from each source to any particular destination. There are some request patterns which cannot be connected simultaneously. For example, any two sources cannot be connected simultaneously to destinations 000 and 111.

11.2.5 Hypercube Interconnection

The hyper cube interconnection is a binary n-cube architecture. With this interconnection it is possible to connect 2^n processors. Each processor form a node of the cube. It is not necessary that at each node there is a processor; however the node can have local memory module or I/O interface. The processor at one node has a direct communication path to n other nodes. Since there are 2^n nodes, there are 2^n distinct n-bit binary addresses.

The Fig. 11.2.11 shows the hyper cube structure for n = 2, 3 and 4. A two cube structure has 4 nodes, three cube structure has 8 nodes and 4 cube structure has 16 nodes. As shown in the Fig. 11.2.11, a 4 cube structure is formed by interconnecting two 3 cubes. The Fig. 11.2.11 also shows the binary addresses assigned to different

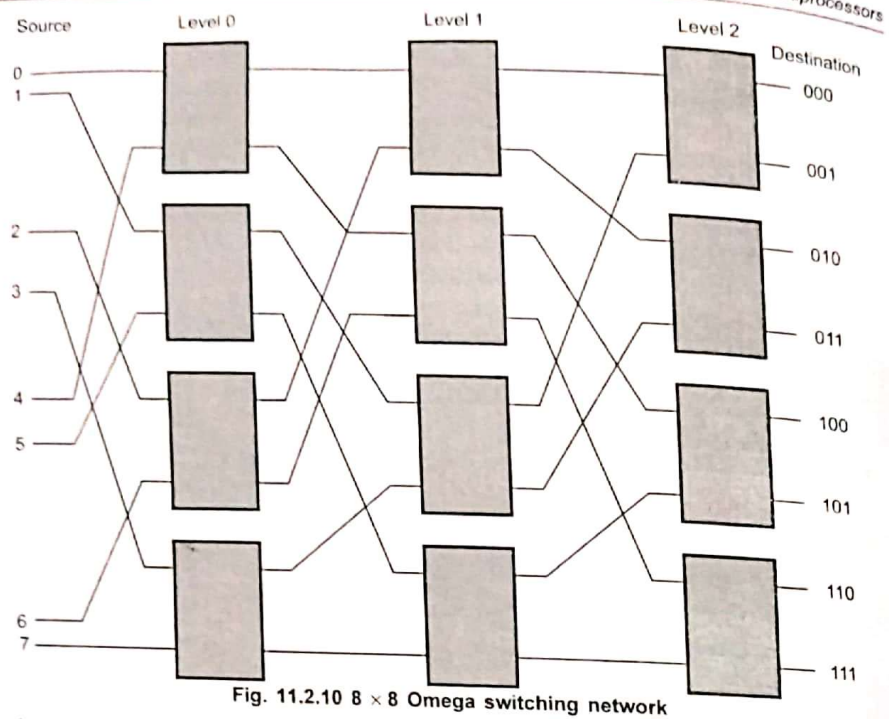
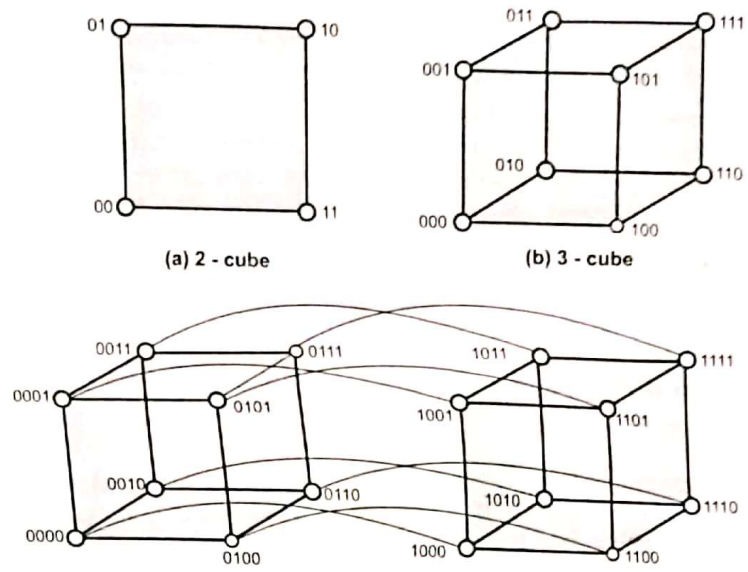


Fig. 11.2.10 8 × 8 Omega switching network

nodes. They are assigned such that the addresses of two neighbours differ in exactly one bit position. For example, the three neighbours of the node with address 000 in the 3 - cube structure are 001, 010 and 100. Each of these binary numbers differ from address 000 by one bit value.

In n-cube structure, we require 1 to n links to route the message from one node to another. For example, in 3 - cube structure we require one link to communicate from node 000 to 010, two links to communicate from node 000 to 101 and three links to communicate from node 000 to 111. We can calculate the number links require to communicate from one node to another. This is done by Exclusive-ORing the binary value of source node with the binary value to destination node. The number of 1s in the result are the number of links required for the communication. For example, in a 3-cube structure, to transfer message from node 001 to node 100 we require two links because Exclusive-ORing of 001 and 100 is 101 and result has two ones.



(a) 2 - cube (b) 3 - cube

(c) 4 - cube

Fig. 11.2.11 Hypercube structures

Review Questions

1. *Draw and explain the time shared bus.*
2. *Draw and explain the crossbar switch system organization for multiprocessor.*
3. *Draw and explain the functional structure of crosspoint in a crossbar network.*
4. *Write a short note on multiport memory system.*
5. *Give the comparison between interconnection structures.*
6. *Explain the basic building block for multistage switching network.*
7. *Draw and explain 1 to 8 way switching using basic 2 × 2 switch.*
8. *Write a short note on omega switching network.*
9. *Write a short note on hypercube interconnection.*

11.3 Interprocessor Arbitration

The processor, main memory and I/O devices can be interconnected by means of a common bus. A bus is a set of lines (wires) designed to transfer all bits of a word from a specified source to a specified destination. Thus bus provides a communication path for the transfer of data. The bus includes data lines, address lines and control lines. Such a bus is known as **system bus.** It also includes the lines needed to support interrupts

and arbitration as a part of control lines. Before going to see the arbitration we will see how data transfer takes place on the bus.

When processor, memory and I/O devices are connected in the system one particular unit acts as the bus master and supervises the use of the bus by the other units, the bus slaves. In most of the cases the processor is the bus master, while the memory and I/O interface circuits are slaves. However, IO controllers also serve as bus masters. Only a bus master can initiate data transfer, although slaves can request for data transfer. Once the data transfer is initiated, both master and slave participate equally in the data-transfer process.

The data transfer can be carried out in two ways :

- Synchronous
- Asynchronous

11.3.1 Synchronous Bus

In synchronous bus, all devices derive timing information from a common clock signal. The Fig. 11.3.1 shows the timing diagram for synchronous input/output transfer. At time t_0, the processor places the device address on the address lines of system bus and sets the control lines to perform the input operation. During time $t_0 - t_1$, addressed device gets the address and it recognizes that an input operation is requested. At time t_1, address device places its data on the data bus. At the end of bus cycle, that, is at time t_2, the processor reads the data lines and loads the data from data bus into its input buffer.

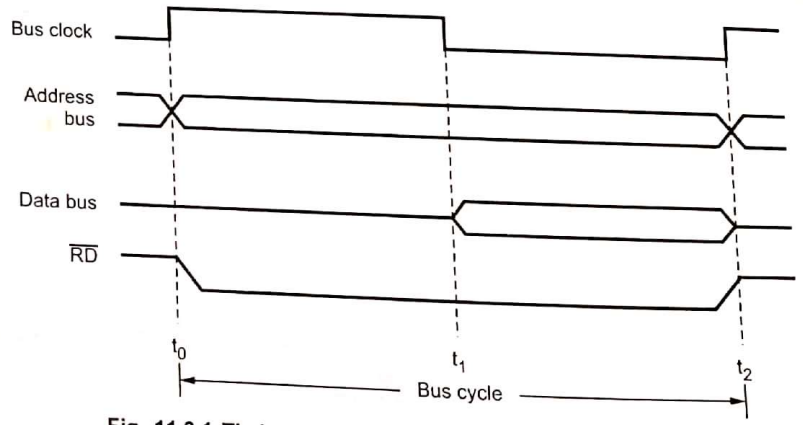


Fig. 11.3.1 Timing diagram for synchronous input transfer

The timing diagram for an output operation is shown in the Fig. 11.3.2. Here, processor places the output data on the data lines when it transmits the address and

control line informations. At time t_1, the addressed device reads the data lines and loads the data into its data buffer. The processor simply assumes that, at t_2, the output data have been received by the I/O device and the cycle ends.

In synchronous system, the clock pulse width should be chosen such that it is greater than the maximum propagation delay between the processor and any of the devices connected to the bus. During read operation, the clock pulse width must be large enough to allow all devices to decode the address and control signals so that the addressed device can be ready to respond at time t_1.

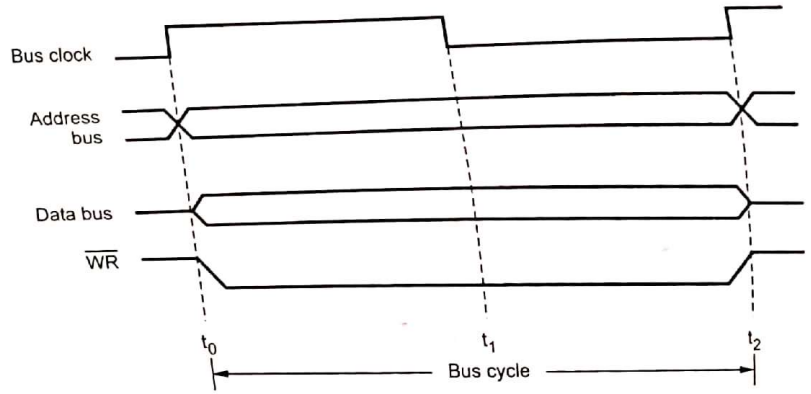


Fig. 11.3.2 Timing diagram for synchronous output transfer

11.3.2 Asynchronous Bus

In asynchronous bus, the common clock is eliminated and data transfer on the system bus is achieved by the use of a hand shake between the processor and the device being addressed. Here, the clock line is replaced by two control signals Ready and Accept.

For input operation, processor places the address and sets the control lines to perform the input operation at t_0. It then indicates at t_1 to the addressed device that it is ready to accept data by activating the Ready line. When the addressed device receives the Ready signal, it places the data on the data bus at t_2 and then informs the processor that it has done so by activating the Accept line. The processor waits for accept line. Upon activation of Accept line processor reads data from the data bus and loads it into the input buffer the timing diagram for input operation is as shown in Fig. 11.3.3. At t_4 processor removes the address and controls information from bus. When the device interface receives the 1 to 0 transition of master ready signal, it removes the data and slave-ready signal from the bus. This completes the input data transfer.

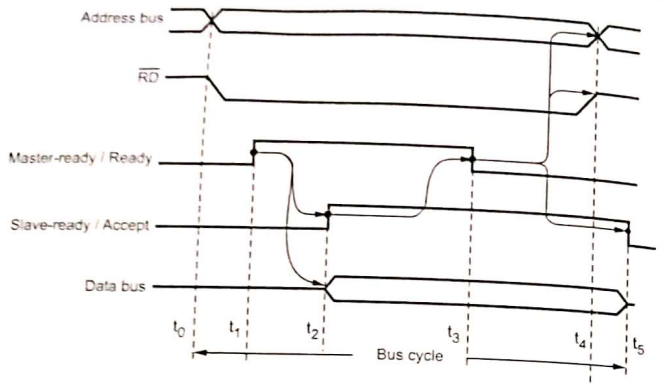


Fig. 11.3.3 Timing diagram for asynchronous input transfer

For output operation, processor places the address, data and sets the control lines to perform the output operation. It then indicates to the addressed device that it has send data by activating Ready signal. The addressed device reads the data into its input buffer when it receives Ready signal. It then indicates that it has done so by activating the Accept signal. The timing diagram for output operation is as shown in Fig. 11.3.4.

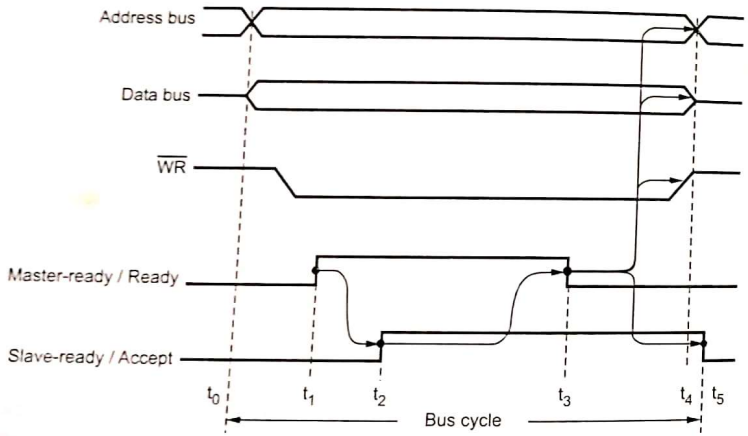


Fig. 11.3.4 Timing diagram for asynchronous output transfer

11.3.3 Bus Arbitration

The device that is allowed to initiate data transfers on the bus at any given time is called the **bus master**. In a computer system there may be more than one bus master such as processor, DMA controller etc. They share the system bus. When current master relinquishes control of the bus, another bus master can acquire the control of the bus. Bus arbitration is the process by which the next device to become the bus master is selected and bus mastership is transferred to it. The selection of bus master is usually done on the priority basis.

11.3.4 Serial Arbitration

The serial priority resolving technique is obtained from a daisy-chain connection of bus arbitration circuit. It is a simple and cheap method. The Fig. 11.3.5 shows the system connections for daisy chaining method. As shown in Fig. 11.3.5, here the processors connected to the system bus are assigned priority according to their position along the bus grant line. The device closest to the bus grant line in assigned the highest priority.

All processors make use of the same line for bus request. In response to a bus request the controller sends a bus grant if the bus is free. The bus grant signal serially propagates through each processor until it encounters the first one that is requesting access to the bus. This processor blocks the propagation of the bus grant signal, activates the busy line and gains control of the bus. Therefore any other requesting module will not receive the grant signal and hence cannot get the bus access.

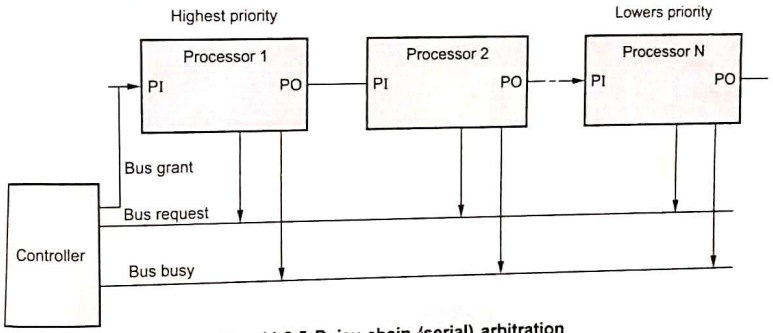


Fig. 11.3.5 Daisy-chain (serial) arbitration

Each processor has its own bus arbiter logic. Let us assume that the bus arbiter logic have Priority-In (PI) and Priority-Out (PO) lines as bus grant signals. The PO line of each arbiter is connected to the PI line of the next lower priority arbiter. The PO output for a particular arbiter is equal to 1 if its PI input is equal to 1 and the processor

associated with the arbiter logic is not requesting the system bus. In this way the priority is propagated from higher priority processor to lower priority processor.

Advantages

1. It is a simple and cheaper method.
2. It requires the least number of lines and this number is independent of the number of processors in the system.

Disadvantages

1. The propagation delay of bus grant signal is proportional to the number of processors in the system. This makes arbitration time slow and hence limits the number of processors in the system.
2. The priority of the processor is fixed by its physical location.
3. Failure of any one processor causes the whole system to fail.

11.3.5 Parallel Arbitration

The Fig. 11.3.6 shows parallel bus arbitration technique. This technique is also known as **independent request.** Here, each bus arbiter has three signals : bus request, bus acknowledge and bus busy. The processor can request for the system bus using bus request line. It can take the bus control only when its acknowledge input is activated in response to the bus request. The external priority encoder and decoder are used to resolve the priority and generate the appropriate acknowledge signal. The priority encoder generates a code which represents the highest priority processor among those requesting the system bus and decoder activates the corresponding acknowledgement line to grant bus access to the highest priority processor.

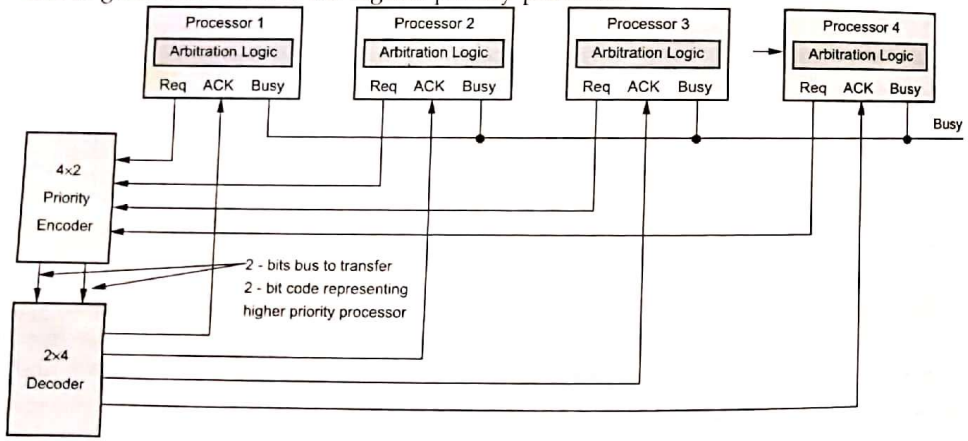


Fig. 11.3.6 Independent request (Parallel) arbitration

Advantage

1. Due to separate pairs of bus request and bus grant signals, arbitration is fast and is independent of the number of processors in the system.

Disadvantage

1. It requires more bus request and grant signals ($2 \times n$ signals for n modules).

11.3.6 Dynamic Arbitration

In serial and parallel techniques, priorities of the processors connected in the system are fixed i.e. **static.** Let us see some techniques which use dynamic priority allocation. The dynamic priority allocation has an ability to change the priorities of the processor while the system is in operation.

Polling technique

The system connections for polling technique are shown in Fig. 11.3.7. In this the controller is used to generate the addresses for the processors. Number of address lines required depends on the number of processors connected in the system. For example, if there are 8 processors connected in the system, at least three address lines are required. In response to a bus request controller generates a sequence of processor addresses. When the requesting processor recognizes its address, it activates the busy line and begins to use the bus.

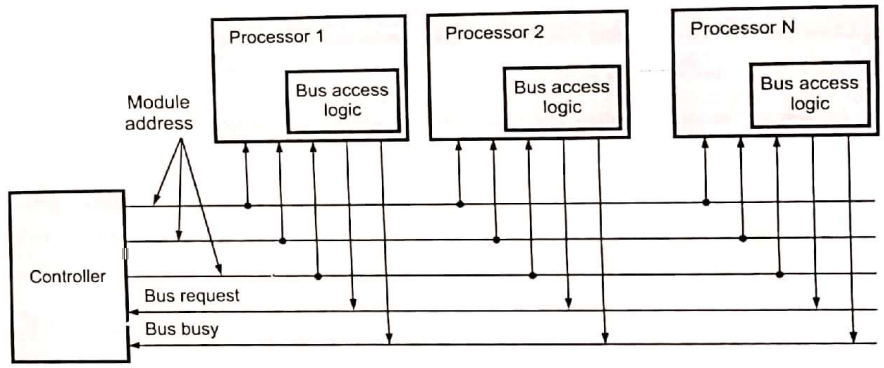


Fig. 11.3.7 Polling technique

Advantages

1. The priority can be changed by altering the polling sequence stored in the controller.
2. If the one module fails entire system does not fail.

Time slice technique

In time slice technique, a fixed length time slice of system bus is allocated to each processor in the system, in a round-robin fashion. Since allocated time for each processor is same, no preference is given to any particular device.

LRU technique

LRU stands for Least Recently Used. This gives the highest priority to the requesting device that has not used the bus for the longest period of time. In this technique priorities are dynamically changed. The processor just used system bus is assigned with lowest priority.

FIFO technique

FIFO stands for first-in first-out. This means that the processor requests are served in the order they have received. To implement this technique the bus controller maintains an queue of processor requests.

Review Questions

1. *What is bus ?*
2. *What is protocol ?*
3. *Draw and explain the timing diagram for synchronous input data transfer.*
4. *Draw and explain the timing diagram for synchronous output data transfer.*
5. *What do you mean by bus arbitration ?*
6. *Explain the serial arbitration technique with the help of neat diagram. State its advantages and disadvantages.*
7. *Explain the parallel arbitration technique with the help of neat diagram.*
8. *What is dynamic arbitration ?*
9. *Explain various techniques used for providing dynamic arbitration.*

11.4 Interprocessor Communication and Synchronization

In a multiprocessing environment processors implies parallelism by concurrent processing. The concurrent processing requires sharing of resources between the processors and interprocessor communication. Parallelism depends on how interprocess communication (IPC) is implemented. The interprocessor communication can be established through common input-output channels. Basically, there are two way by which the interprocessor communication is achieved

- Using shared variables
- Using message passing

11.4.1 IPC using Shared Variables

In a shared memory systems interprocessor communication is achieved using shared variables. In such systems, the shared variables are stored in the common memory which is accessible to all processors in the system. This is illustrated in Fig. 11.4.1.

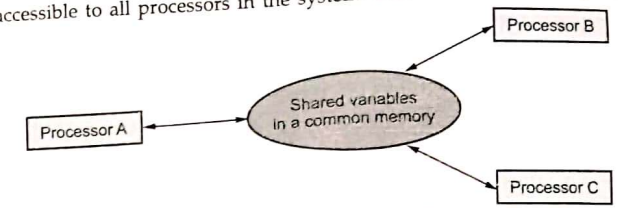


Fig. 11.4.1 Interprocessor communication using shared variables

While sharing common resources or shared variables conflict problem may arise. It is necessary to prevent conflicting use of shared resources by several processors. This task is done by operating system. In multiprocessor systems, the design of operating system can be categorized as master-slave configuration operating system, separate operating system and distributed operating system.

Master-slave operating system - In this processor that executes the operating system functions is denoted as **master**. The other processor do not perform operating system functions and they are denoted as **slaves**. When slave needs operating system service it requests it by interrupting the master.

Separate operating system - In this each processor has entire copy of operating system and can execute operating system functions. This organisation is more suitable for loosely couple systems.

Distributed operating system - In this operating system, routines are distributed among the available processors. The operating system routines float from one processor to another processor whenever they are required. Thus, such type of operating system is also known as **floating operating system.**

11.4.2 IPC using Message Passing

In multiprocessor systems with no shared memory we use message passing mechanism to perform interprocess communication. The communication between processors is achieved by message passing through I/O channels. This is illustrated in Fig. 11.4.2.

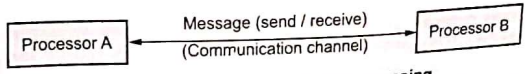


Fig. 11.4.2 IPC using message passing

When processor wants to communicate with another processor it uses a special procedure which initiates communication. It identifies the destination processor and once source and destination processors are identified a communication channel is established. A message is then sent through the communication channel.

11.4.3 Interprocessor Synchronization

At the higher level of exploiting parallelism, the program is partitioned into process that are executed on different processors. This technique is called **concurrent processing**. During concurrent processing when two or more processors need the same resource at a time, the contention problem arise. Such problem can be solved by synchronization. Therefore, in multiprocessors, in order to synchronize concurrent processes, special synchronizing software tools are used that are usually supported by hardware mechanisms. In large-scale or high contention situations, synchronization can become a performance bottleneck because contention results additional delays.

To achieve synchronization a set of hardware primitives are used to atomically read and modify a memory location without any interruption between the two operations.

Such a mechanism is necessary to protect data from being changed simultaneously by two or more processors. This mechanism is known as **mutual exclusion**. A program sequence which accesses the shared resources, one begun, must complete execution before another processor accesses the same shared resources. Thus a program sequence which accesses the shared memory is known as **critical section** of the program. Let us discuss two commonly used methods for interprocessor synchronization.

Mutual exclusion with a semaphore

In synchronization operation, a binary flag or variable called **semaphore** is stored in the shared memory to indicate whether or not a processor is executing a critical section. A semaphore is a software control flag. When it is equal to 1, it means that a processor is executing a critical section, so that the shared memory is not available to other processors. When it is equal to 0, the shared memory is available to any requesting processor. Therefore, the processor gets access of shared memory when he found semaphore = 0. When processor gets access of shared memory semaphore is set to 1. This is performed by the exchange instruction. The exchange instruction exchanges the value of semaphore with a value 1 in the register. The value returned in the register by the exchange instruction is 1 if some other processor had already claimed the access and 0 otherwise. In the later case processor gets access to shared memory and semaphore set to 1 by exchange instruction preventing any competing processor to get the access of shared memory.

In 8086, the XCHG instruction along with the LOCK prefix can be used to set or reset semaphore. Here is the program sequence.


```
                MOV AL, 01H
LOCK CHAG :     XCHG semaphore, AL
                CMP AL,00H
                JNZ CHAG
                .
                .   Critical region in which program
                .   access the shared resources
                .
                MOV semaphore, 0
```

The XCHG semaphore, AL instruction exchanges the contents of the AL register with the contents of the memory location in which semaphore is stored. The XCHG instruction requires two bus cycles. During the XCHG instruction, it is necessary to restrict other processor from accessing semaphore. This is achieved by LOCK prefix in the 8086. LOCK prefix activates the $\overline{\text{LOCK}}$ output pin during the execution of the instruction that follows the prefix. During the execution of XCHG instruction, the $\overline{\text{LOCK}}$ output pin is in the active state which does not allow other processor to get control of the system bus. A processor generated signal (in this case $\overline{\text{LOCK}}$) that serves to prevent other processors from using the system bus as long as the signal is active is called **Hardware Lock**. After execution of the XCHG instruction the status of the semaphore is available in the AL register. By checking the contents of the AL register it is possible to decide whether to enter into critical region of program or to check semaphore again.

Mutual exclusion using load linked and store conditional instructions

In some recent microprocessors, slightly different approach is provided to perform read and update operation. Because implementing a single atomic memory operation introduces some challenges, since it requires both a memory read and write in a single, uninterruptible instruction. This requirement complicates the implementation if coherence. Thus in an alternative approach a pair of a instructions where the second instruction returns a value from which it can be deduced whether the pair of instructions was executed as if the instruction were atomic. The pair of instructions are said to be atomic if all other operations executed by any processor occurred before or after the pair. The first instruction in the pair is a special load instruction called **load linked** or **load locked** and the second instruction in the pair is called a **store conditional**. These instructions are always used in sequence. The store operation is performed only when the condition is true. If the contents of the memory location specified by the load linked are changed before the store conditional to the same address occurs, then the store conditional fails. The store conditional also fails if processor does a context switch between the two instructions. The store conditional instruction return 1 if successful and 0 otherwise. The instruction sequence to implement an atomic exchange is as given below.

```
TRY     MOV R1, data    ; Move value to be exchanged in R1
        LL R2 (R0)      ; Load linked (load the contents of
                        ; memory location specified by R0 in R2)
        SC (R0), R1     ; Store conditional
        BEQZ R1, TRY    ; Branch if store conditional fails
        MOV R1, R2      ; Stores the load value in R1
```

After this instruction sequence the contents of R_1 and the memory location specified by R_1 have been automatically exchanged. If any other processor modifies the contents of memory location pointed by R_0 between the LL and SC instructions, the store conditional fails and it returns 0 in R_1, resulting the instruction sequence of try again.

The register used to specify the memory address in the LL register is called the **link register**. In case of any interrupt or invalidation of the same cache block, the link register is cleared. The SC instruction simply checks that its address matches that in the link register. If so SC succeeds; otherwise it fails.

An advantages of load linked/ store conditional mechanism is that it can be used to build other synchronization primitives such as fetch and increment. Let us see the instruction sequence for fetch and increment.

```
TRY     LL R2, (R0)     ; Load linked (load the contents of
                        ; memory location specified by R0 in R2)
        MOV R1, R2      ; store loaded value in R1
        INC R1          ; increment
        SC (R0), R1     ; store conditional
        BEQZ R1, TRY    ; Branch if store conditional fails
```

In this instruction sequence two instruction are added between LL and SC instructions to implement fetch and increment primitive. To maximize the probability of getting successful SC we have restrict the number of instructions between LL and SC. Further more, there instructions should be preferably register-to-register instructions; otherwise, it is possible to create dead lock situations where the processor can never complete SC.

Review Questions

1. Explain the interprocessor communication using shared variables.
2. Explain the interprocessor communication using message passing.
3. What is synchronization ?
4. What is mutual exclusion and critical section of program ? Explain the process of mutual exclusion with a semaphore.
5. Explain the process of mutual exclusion using load link and store conditional instructions.

11.5 Cache Coherence

We know that, cache memories are introduced in computers in order to bring data closer to the processor and hence to reduce memory latency. In multiprocessor systems, several processors require a copy of the same memory block and they store a copy of the same memory block in their individual caches. Now, if the processors are allowed to update the data in the cached memory block in its individual cache, an inconsistent view of memory can result. This problem is known as 'cache coherence problem'.

In a coherent multiprocessor system, the caches provide both migration and replication of shared data items. **Replication** allows that multiple copies of the same data item reside simultaneously in different local caches, in order to reduce both latency of access and contention for a read shared data item. On the other hand, **migration** implies a single copy of a data item that must be moved to the accessing site for exclusive use; this reduces the latency to access a shared data item that is allocated remotely.

In a multiprocessor system supporting both migration and replication of shared data items, coherence is maintained by introducing hardware based or software based protocols. The protocols used to maintain coherence for multiple processors are called **cache coherence protocols**. Software based protocols usually introduce some restrictions on the cacheability of data in order to prevent cache coherence problems. On the other hand, the hardware based protocols provide general solutions to the problems of cache coherence without any restrictions on the cachability of data. However, they need sophisticated hardware mechanisms to support cache coherence. Hardware-based protocols can be classified according to their :

- Memory update policy
- Cache coherence protocol and
- Interconnection scheme

Memory Update Policy : In cache system, two copies of same data may exist at a time, one in cache and one in main memory. If one copy is altered and other is not, two different sets of data become associated with the same address. To prevent this memory system uses two types of memory update policies : **Write-through policy** and **write-back policy**.

In write through policy, the cache controller copies data to the main memory immediately after it is written to the cache. Due to this main memory always contains a valid data, and any block in the cache can be overwritten immediately without data loss. Therefore, the write-through policy maintains consistency between the main memory and caches. In write-back policy, the alter bit in the tag field is used to keep information of the new data. Memory contents are not updated immediately. Memory contents are

required to read the data. ... time is

Table 11.5.1

Interconnection Scheme : Depending on the nature of the interconnection network, hardware-based protocols are further classified into three basic classes :

- **Snoopy cache coherence protocol :** Snoopy cache coherence protocols are very popular in shared bus multiprocessors due to their relative simplicity. In this, every cache that has a copy of the data from a block of physical memory also has a copy of the sharing status of the block, and no centralized state is kept. This protocol is typically used in single bus-based shared memory system where consistency commands (invalidate or update commands) are broadcast via the bus, and all cache controllers monitor or snoop on the bus to determine whether or not they have a copy of a block that is requested on the bus.
- **Directory-based cache coherence protocol :** Large interconnection networks (for example, multistage networks) cannot support broadcasting efficiently and therefore a mechanism is needed that can directly forward consistency command to those caches that contain a copy of the updated data. For this purpose the sharing status of a block of physical memory is kept in just one location, called the directory.
- **Hierarchical cache coherence protocol :** This protocol tries to avoid the application of the costly directory protocol but still provide high scalability. It proposes multiple-bus networks with the application of hierarchical cache coherence protocols that are generalized or extended version of the single bus-based snoopy cache protocol.
- The Fig. 11.5.1 summarises the design space and classification of hardware-based protocols. It is important to note that, for any hardware based protocols there are three choices and can be used in any combination.

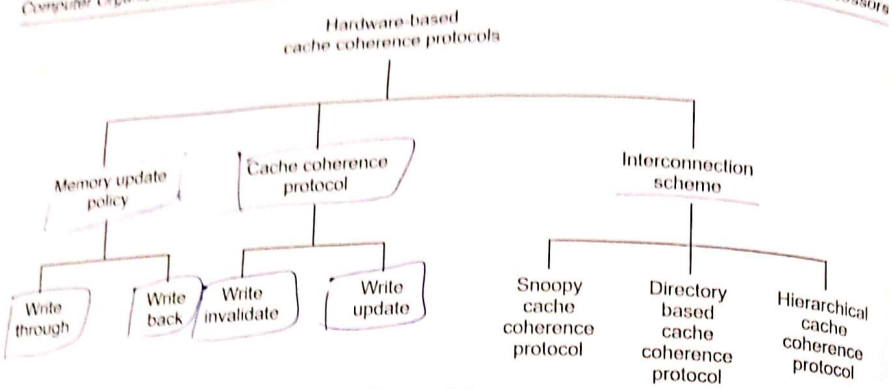


Fig. 11.5.1

Review Questions

1. *What is cache coherence ?*
2. *Explain memory update policies to prevent cache coherence problem.*
3. *What is cache coherence protocol ?*
4. *Explain cache coherence protocols.*
5. *Give the classification of cache coherence protocol depending on the nature of the interconnection network.*

11.6 Shared Memory Multiprocessors

Existing MIMD multiprocessors fall into two classes, depending on the number of processors involved, which in turn dictate a memory organisation and interconnect strategy. We know these multiprocessors by their memory organisations, because the type of memory organization decides whether a small or large number of processors in one multiprocessor. According to memory organization the MIMD multiprocessors can be classified as follows.

- **Centralized shared-memory architectures :**

Multiprocessors with small number of processors and processors in the multiprocessor can share single centralized memory.

- **Distributed shared-memory architectures :**

Multiprocessors with large number of processors and physical distributed memory among the processors.

11.6.1 Centralized Shared Memory Architectures

In centralized shared memory architectures with a small number of processors, it is possible for the processors to share a single centralized memory and to interconnect the processors and memory by a bus. In such systems, large caches, the bus and the single memory can satisfy the memory demands of a small number of processors. A single *main* memory used in the system has uniform access time from each processor. These systems are therefore also referred to as UMAs for **uniform memory access**. The Fig. 11.6.1 shows the architecture of multiprocessor system with centralized shared memory.

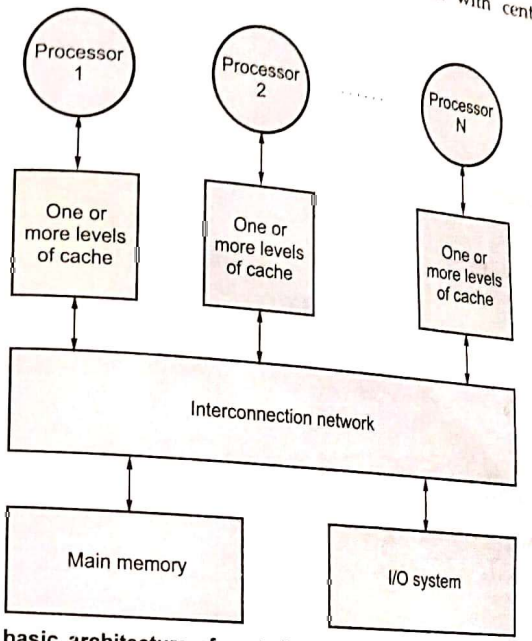


Fig. 11.6.1 The basic architecture of centralized shared memory multiprocessor system

The architecture shown in the Fig. 11.6.1 supports the caching of both shared and private data. Private data is used by a single processor, while shared data is used by multiple processors. When a private data item is cached, its location is migrated to the cache, reducing the average access time as well as the memory bandwidth required. Since no other processor uses this data, the program behaviour is identical to that in a uniprocessor system. On the other hand, when shared data are cached, the shared data items may be replicated in multiple caches. This replication provides a reduction in contention that may exists for shared data items that are being read by multiple processors simultaneously, in addition to the reduction in access latency and required

memory bandwidth. However, caching of shared data items introduces a new problem called **cache coherence**. This problem is discussed in section 11.5.

11.6.2 Distributed-Shared Memory Architectures

Distributed shared memory architectures consist of multiprocessors with physically distributed memory. With rapid increase in processor performance and the associated increase in a processor's bandwidth requirements, the centralized shared memory architecture is not suitable for large number of processors in the multiprocessor system. Hence, to support large number of processors in the multiprocessor system, memory must be distributed among the processors rather than centralized. The Fig. 11.6.2 shows the basic architecture of multiprocessor system with distributed-shared memory.

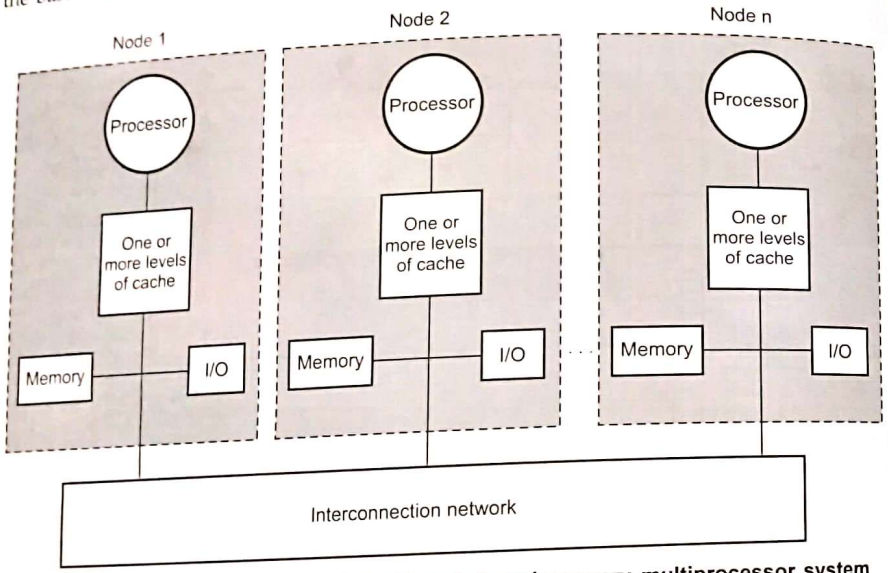


Fig. 11.6.2 The basic architecture of distributed shared memory multiprocessor system

As shown in the Fig. 11.6.3 the basic architecture of distributed shared memory (DSM) multiprocessor consists of individual nodes containing a processor with cache, some memory, typically some I/O and an interface called interconnection network to connection all the nodes. Therefore, the DSM system can be generally viewed as a set of nodes or clusters, connected by an interconnection network.

In typical DSM system architecture, individual nodes may contain a small number of processors, which may be interconnected by a small bus or a different interconnection technology. The Fig. 8.6.3 shows the typical DSM system architecture.

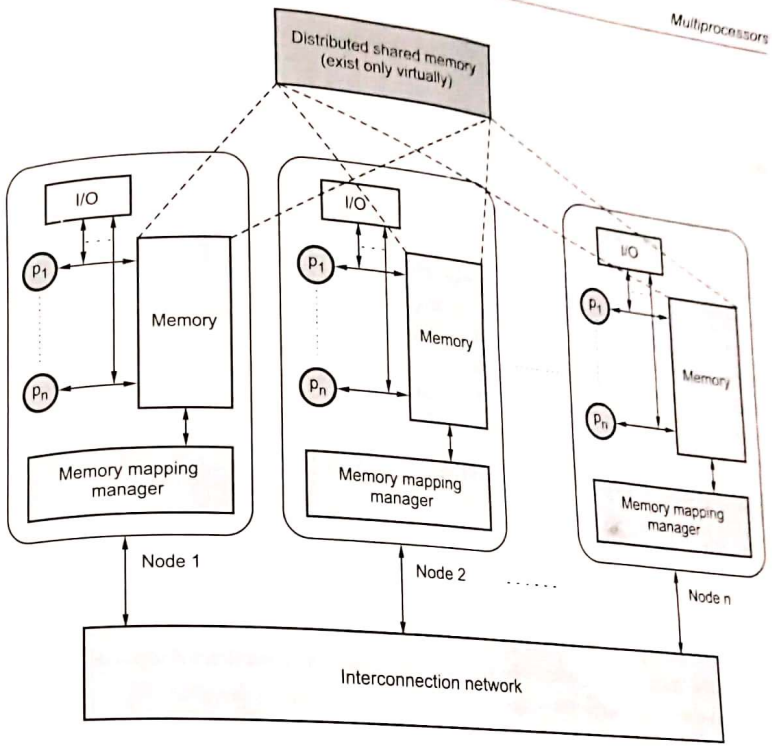


Fig. 11.6.3

Distributing the memory among the nodes has two major advantages :

1. It is a very cost-effective way to scale the memory bandwidth, if most of the accesses are to the local memory in the node.
2. It reduces the latency for accesses to the local memory.

The major disadvantage of a distributed memory architecture is that communicating data between processors becomes somewhat more complex and has higher latency because the processors no longer share a single centralized memory.

When a process on a node accesses some data from a memory block of the shared-memory space, the local memory-mapping manager takes charge of its request. If the memory block containing the accessed data from the local memory. Otherwise, a network block fault is generated and the control is passed to the operating system. The operating system (software layer) then sends a message to the node on which the desired memory block is migrated from the remote node to the requested processor's

node and the operating system maps it into the application address space. The faulting instruction is then restarted and can now complete. Therefore, the scenario is that data blocks keep migrating from one node to another on demand but no communication is visible to the user processes. Copies of data cache in the local memory eliminate network traffic for a memory access on cache hit, that is access to an address whose data is stored in the cache. Therefore, network traffic is significantly reduced if applications shown a high degree of locality of data accesses.

Distributed shared memory systems could choose to exclude or include cache coherence. The simplest scheme for hardware is to exclude cache coherence. These systems have caches, but to prevent coherence problems, shared data are marked as uncacheable and only private data are kept in the caches. However, software can still explicitly cache the value of the shared data by copying the data from the shared portion of the address space to the local private portion of the address space that is cached. In such systems coherence is controlled by software.

This approach has several disadvantages :

- Compiler mechanisms for transparent software cache coherence are very limited and they have significant overhead arising from explicitly copying data. The compiler based cache coherence is currently impractical for irregular problems or problems involving dynamic data structures and pointers.
- In such systems, for each reference only single word is fetched from the remote memory instead of cache block. Due to this the benefits of spatial locality in shared data cannot be leveraged.
- Such systems have large latency of access to remote memory versus a local cache.

For handling coherence problems in DSM systems it is necessary to have special protocol called **directory protocol**. A directory keeps the state of every block that may be cached. Information in the directory includes a set of pointers to the caches holding a valid copy of the block, the state of the cached block and so on.

Review Questions

1. *Explain the structural difference between centralized and distributed shared memory architecture.*
2. *Draw and explain the centralized shared memory architecture.*
3. *Discuss the cache coherence in the centralized shared memory architecture.*
4. *Draw and explain the distributed shared memory architecture.*
5. *List the advantages of distributed share memory architecture.*
6. *Discuss the cache coherence related to distributed shared memory architectures.*

❑❑❑

Computer Organization Tutorial

Tutorial 1 : **A digital computer has a common bus system for 16 registers of 32 bits each. The bus is constructed with multiplexers.**
How many selection inputs are there in each multiplexer ?
What size of multiplexers are needed ?
How many multiplexers are there in the bus ?

Ans. : Refer Example 2.3.1

Tutorial 2 : **The following transfer statements specify a memory. Explain the memory operation in each case.**
R2 ← M[AR]
M[AR] ← R3
R5 ← M[R5]

Ans. :

R_2 ← M[AR] : This statement causes a transfer of information from the memory word M selected by the address in AR into register R2.

M[AR] ← R3 : This statement causes a transfer of information from register R3 into the memory word M selected by the address in AR.

R5 ← M[R5] : This statement causes a transfer of information from the memory word M selected by the address in register R5 into register R5.

Tutorial 3 : **The adder-subtractor circuit of Fig. 2.4.6 (a) has the following values for input mode M and data inputs A and B. In each case, determine the values of the outputs : S_3, S_2, S_1, S_0 and C_4.**

	M	A	B
a.	0	0111	0110
b.	0	1000	1001
c.	1	1100	1000
d.	1	0101	1010
e.	1	0000	0001

Ans. : For the circuit shown in the Fig. 2.4.6(a) when M = 0 circuit performs operation $S \leftarrow A + B$ and when M = 1 circuit performs operation $S \leftarrow A + \overline{B} + 1$

a) M = 0, A = 0111, B = 0110 ∴ Operation : $S \leftarrow A + B$

	1	1				
	0	1	1	1	A	∴ $S_3S_2S_1S_0$= 1101
+	0	1	1	0	B	C_4 = 0
	1	1	0	1		

b) M = 0, A = 1000, B = 1001 ∴ Operation : $S \leftarrow A + B$

1						
	1	0	0	0	A	∴ $S_3S_2S_1S_0$= 0001
+	1	0	0	1	B	C_4 = 1
1	0	0	0	1		

c) M = 1, A = 1100, B = 1000 ∴ Operation : $S \leftarrow A + \overline{B} + 1$

$\overline{B}$ = 0111

1	1	1	1			
+	1	1	0	0	A	∴ $S_3S_2S_1S_0$= 0100
+	0	1	1	1	$\overline{B}$	C_4 = 1
+				1	1	
1	0	1	0	0		

d) M = 1, A = 0101, B = 1010 ∴ Operation : $S \leftarrow A + \overline{B} + 1$

$\overline{B}$ = 0101

1		1				
	0	1	0	1	A	∴ $S_3S_2S_1S_0$= 1011
+	0	1	0	1	$\overline{B}$	C_4 = 0
+				1	1	
1	0	1	1			

e) M = 1, A = 0000, B = 0001 ∴ Operation : $S \leftarrow A + \overline{B} + 1$

$\overline{B}$ = 1110

0	0	0	0	A	∴ $S_3S_2S_1S_0$= 1111	
1	1	1	0	$\overline{B}$	C_4 = 0	
			1	1		
1	1	1	1			

Tutorial 4 : Design a 4-bit combinational circuit decrementer using four full-adder circuits.

Ans. :

The decrement microoperation subtracts one from a number in a register. Fig. 1 shows a 4-bit combinational circuit decrementer. One of the inputs of the least significant Half-Subtractor (HS) is connected to logic-1 and the other input is connected to the least significant bit of the number to be decremented.

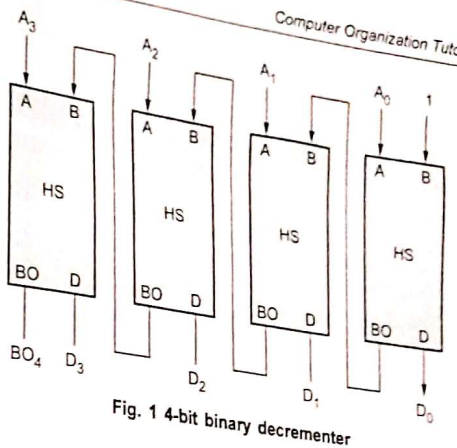
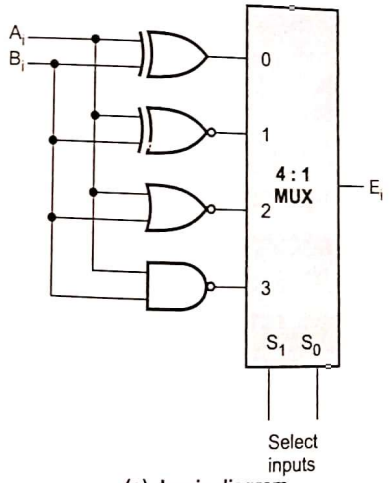


Fig. 1 4-bit binary decrementer

The Output Borrow (BO) from one half-subtractor is connected to one of the inputs of the next-higher order half-subtractor. The circuit receives the four bits from A_0 through A_3, subtracts one from it, and generates the decremented output in D_0 through D_3. The output Borrow, BO_4 will be 1 only after decrementing binary 0000. This also causes outputs D_0 through D_3 to go to 1111.

Tutorial 5 : Design a digital circuit that performs the four logic operations of exclusive-OR, exclusive-NOR, NOR and NAND. Use two selection variables. Show the logic diagram of one typical stage.

Ans. :



(a) Logic diagram

S_1 S_0	Output	Operation
0 0	E = A⊕B	XOR
0 1	E = A⊙B	XNOR
1 0	E = $\overline{A \vee B}$	NOR
1 1	E = $\overline{A \wedge B}$	NAND

(b) Function table

Fig. 2

Tutorial 6 : Register A holds the 8 bit binary 11011001. Determine the B operand and the logic microoperation to be performed in order to change the value in A to :
f. 01101101
g. 11111101

Ans. : A = 11011001 Operation = XOR

f = 01101101 Operand B = 10110100

	1	1	0	1	1	0	0	1	A
XOR	1	0	1	1	0	1	0	0	B
	0	1	1	0	1	1	0	1	f

A = 11011001 operation = OR

g = 11111101 B = XX1XX10X

1	1	0	1	1	0	0	1	A
X	X	1	X	X	1	0	X	B
1	1	1	1	1	1	0	1	

Note X : Don't care
X : Bit may be 0 or 1

Tutorial 7 : The 8 bit registers AR, BR, CR and DR initially have the following values :
AR = 11110010
BR = 11111111
CR = 10111001
DR = 11101010
Determine the 8 bit values in each register after the execution of the following sequence of microoperations.
AR ← AR + BR
CR ← CR ^ DR, BR ← BR + 1
AR ← AR – CR

Ans. : AR ← AR + BR

∴ AR = 11110001

CR ← CR^ DR, BR ← BR + 1

	1	1	1	1	1	1	1		
		1	1	1	1	0	0	1	0	AR
+		1	1	1	1	1	1	1	1	BR
	1	1	1	1	1	0	0	0	1	AR modified

	1	0	1	1	1	0	0	1	CR
AND	1	1	1	0	1	0	1	0	DR
	1	0	1	0	1	0	0	0	CR modified

1	1	1	1	1	1	1	1		
	1	1	1	1	1	1	1	1	BR
+								1	1
0	0	0	0	0	0	0	0		BR modified

AR ← AR – CR

$\overline{CR}$ = 01010111

∴ AR ← AR + $\overline{CR}$ + 1

1	1	1	1		1	1	1		
+	1	1	1	1	0	0	0	1	AR
+	0	1	0	1	0	1	1	1	$\overline{CR}$
								1	1
	0	1	0	0	1	0	0	1	AR modified

Final values

AR = 0100 1001

BR = 0000 0000

CR = 1010 1000

DR = 1110 1010

Tutorial 8 : An output program resides in memory starting from address 2300. It is executed after the computer recognizes an interrupt when FGO becomes a 1 (while IEN = 1).
a. What instruction must be placed at address 1 ?
b. What must be the last two instructions of the output program ?

Ans. : a) Instruction must be placed at address 1 is BUN SRV.

b) Last two instruction of output program are : ION

BUN ZROI

Tutorial 9 : Write an assembly level program for the following pseudocode.
SUM = 0
SUM = SUM + A + B
DIF = DIF – C
SUM = SUM + DIF

Ans. : Refer example 4.3.1.

Tutorial 10 : Write a program loop using a pointer and a counter to clear the contents of hex locations 500 to 5FF with 0.

Ans. : Refer example 4.5.1.

Tutorial 11 : Write an ALP to add two double-precision numbers.

Ans. : Refer example 4.6.2.

Tutorial 12 : Write a program that evaluates the logic EX-OR of two logic operands.

Ans. : Assuming basic computer instructions : AND, CMA and CLA, the program for logic EX-OR of two logic operands is as shown below.

EX-OR : $A\overline{B} + \overline{A}B = \overline{\overline{A\overline{B}} + \overline{\overline{A}B}} = \overline{\left(A\overline{B}\right) \bullet \left(\overline{\overline{A}B}\right)}$

```
    LDA B     ; Load second operand B
    CMA       ; Complement to get B̄
    AND A     ; AND with B̄ to get A • B̄
    CMA       ; Complement to get A B̄
    STA TMP   ; Store A B̄
    LDA A     ; Load first operand A
    CMA       ; Complement to get Ā
    AND B     ; AND with Ā to get Ā • B
    CMA       ; Complement to get A B̄
    AND TMP   ; AND with A B̄ to get (A B̄) • (Ā B)
    CMP       ; Complement to get (A B̄) • (Ā B)
```

Tutorial 13 : Write a program for the arithmetic shift-left operation. Branch to OVF if an overflow occurs.

Ans. :

```
    LDA  OP   ; Load operand
    CLE       ; Clear E.
    CIL       ; Make least significant bit = 0 and
              ; load sign bit in E
    SZE       ; Check E
    BUN ONE   ; Branch if E = 1
    SPA       ; Skip next operation if AC = 0
    BUN OVF   ; Branch to OVF
    BUN EXT   ; Branch to EXT
ONE, SNA      ; Skip next operation if AC = 1
    BUN OVF   ; Branch to OVF
EXT, HLT      ; Stop
```

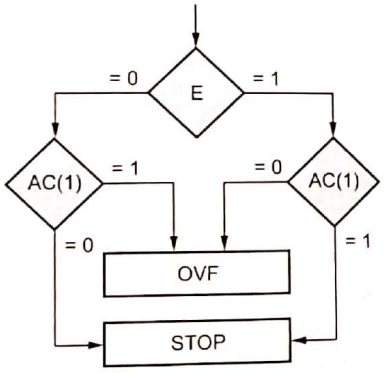


Fig. 3

***Tutorial* 14 : For the given program below :**

1. **Explain in words what the program accomplishes when it is executed. What is the value of location CTR when the computer *halts* ?**

2. **List the address symbol table obtained during the first pass of the assembler.**

```
        ORG 100
        CLE
        CLA
        STA CTR
        LDA WRD
        SZA
        BUN ROT
        BUN STP
ROT,    CIL
        SZE
        BUN AGN
        BUN ROT
AGN,    CLE
        ISZ CTR
        SZA
        BUN ROT
STP,    HLT
CTR,    HEX 0
WRD,    HEX 62C1
```

Ans. : **(a)** The program counts the number of 1's in the number stored in location WRD. Since WRD = $(62C1)_{16}$ = $(0110\ 0010\ 1100\ 0001)_2$

Number of 1's is 6 ; so CTR will have $(0006)_{16}$

(b)

```
                ORG     100     ; Origin of program is location 100
100  7400       CLE             ; Clear E
101  7800       CLA             ; Clear AC
102  3110       STA     CTR     ;  Initialize counter to zero
103  2111       LDA     WRD     ; Load hex word 62C1
104  7004       SZA             ; Skip if AC is zero
105  4107       BUN     ROT     ; Branch to ROT
106  410F       BUN     STP     ; Word is zero, stop with CTR = 0.
107  7040   ROT, CIL             ; Bring bit to E
108  7002       SZE             ; Skip if E is zero
109  410B       BUN     AGN     ; Bit = 1, Go to count it
10A  4107       BUN     ROT     ; Bit = 0, repeat
10B  7400   AGN, CLE             ; clear E
```

```
10C  6110        ISZ     CTR     , Increment counter
10D  7004        SZA             , Check if remaining bits = 0
10E  4107        BUN     ROT     , No; rotate again
10F  7001        STP, HLT         , Yes ; Stop
110  0000        CTR, HEX  0
111  62C1        WRD, HEX  62C1
                 END
```

Tutorial 15 : Write a subroutine to subtract two numbers. In the calling program, the BSA instruction is followed by the subtrahend and minuend. The difference is returned to the main program in the third location following the BSA instruction.

Ans. :

Calling program

```
BSA  SUB  ; Branch and save return address

HEX  1234 ; subtrahend
HEX  4321 ; minuend
HEX  0    ; difference
```

Subroutine

```
SUB, HEX  0
     LDA  SUB I ; Load subtrahend
     CMA        ; Complement subtrahend
     INC        ; Increment by one
     ISZ  SUB   ; Increment pointer
     ADD SUB I  ; Add minuend to 2's
                ; Complement of
                ; Subtrahend
     ISZ SUB    ; Increment pointer
     STA SUB    ; Store the difference
     ISZ SUB    ; Increment pointer
     BUN SUB I  ; Return  to calling
                ; program
```

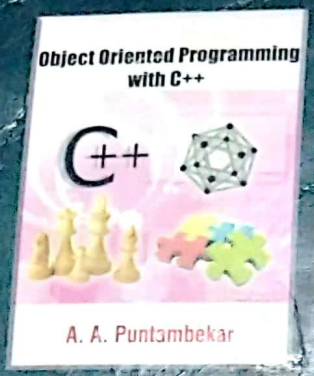
Tutorial 16 : Convert the following into reverse polish notation.
1) A+B*[C*D+E*(F+G)]
2) A*[B+C*(D+E)] / [F+G*(H+I)]

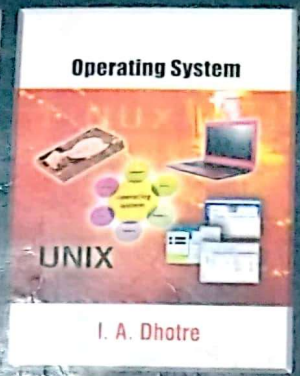
Ans. : Refer Example 6.3.3.

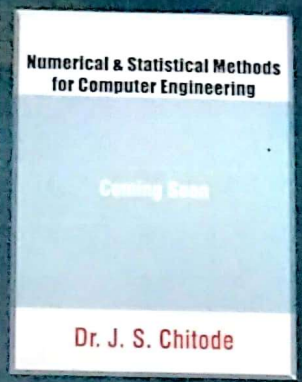
Tutorial 17 : Explain stack and evaluate the following expression using stack (3+4)*[10(2+6)+8]

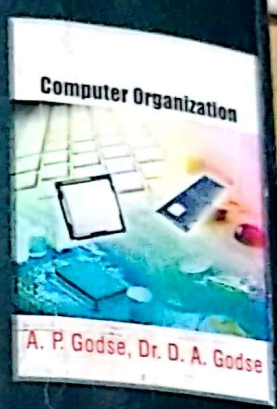
Ans. : Refer Example 6.3.4.

For Semester - IV (CE / CSE / IT / I&CT) . . .











Distributors

- **UBS Publishers Pvt Ltd.**
  Shop No.133-134, Daryapur Gate, Ahmedabad,
  Gujarat-6, Ph: (079) 22160373,
  email: ubspdahm@ahm.ubspd.com
- **Jaico Publishing House**
  2, Mill Officers Colony, Ashram Road, Ahmedabad.
  email: jaicoahm@yahoo.co.in
- **Mahajan Book Depot**
  Gandhi Road, Ahmedabad - 1.
  Ph: (079)25356031.
- **Patel Book Agency**
  1st Floor, Mahavir Smruti Complex, Gandhi Road,
  Ahmedabad-1, Ph: (079)25324741.
- **Atul Agencies**
  Fernandis Bridge, Gandhi Road,
  Ahmedabad - 1, Ph: (079)22160475.

Or Contact

- **Ajay Pachegaonkar (Technical Publications)**
  Mkt. Executive (Gujarat)
  Mob No.:09687118452,
  email: ajay@vtubooks.com

Scan this QR Code & visit us at
www.technicalpublications.org

Price : ₹ 399/-
ISBN 978-93-332-0067-7

TECHNICAL PUBLICATIONS
An Up-Thrust for Knowledge

https://www.facebook.com/technicalpublications
https://twitter.com/p_technical
https://plus.google.com/102423981228640599931/posts
http://www.linkedin.com/company/technical-publications
http://www.pinterest.com/technicalebooks