

A
Course File
On
“COMPUTER ORGANIZATION AND ARCHITECTURE”

Submitted by
Dr. Nidamanuru Srinivasa Rao
Associate Professor

In the Department of
Computer Science and Engineering



NARSIMHA REDDY ENGINEERING COLLEGE (UGC-AUTONOMOUS)

(Affiliated to J.N.T.U, HYDERABAD)

MAISAMMGUDA (V), DHULAPALLY (P), MEDCHAL (M) SECUNDERABAD-500100
(2022-2023)

COURSE FILE

Program Name : B.Tech- Computer Science and Engineering
Name of the Course : COMPUTER ORGANIZATION AND ARCHITECTURE
Course Code : CS2104PC
Year & Semester : II– B.Tech- I SEM

S.NO	CONTENTS	Included
1	Department Vision & mission, PEOs ,PSOs and POs	YES
2	Academic Calendar	YES
3	Syllabus	YES
4	CO/PO Mapping	YES
5	Nominal Rolls of the Students	YES
6	Time Table	YES
7	Lesson Plan	YES
8	Unit Wise question Bank	YES
9	Old Question Papers	YES
10	Question Papers (CIA & SEE)	YES
11	Tutorial Sheets	YES
12	Learning Methodologies: Experiential learning (Industrial Visits, Internships, Mini Projects, Academic Projects, Guest Lectures, Students workshops etc), Problem Solving Methodologies (Assignmnet, Quiz, Case study etc.)	YES
13	Subjects Notes/PPTs/Self study material.	YES
14	Feedback on curriculum design and development	YES
15	CO/PO attainment, analysis and action taken report	YES
Recommendation/ Remarks :		

1. Department Vision & Mission

Vision of the Department:

To evolve as a center of excellence with international reputation by adapting the rapid advancements in the computer specialization fields.

Mission of the Department:

1. To provide a strong theoretical and practical background in the area of computer science with an emphasize on software development
2. To inculcate Professional behavior, strong ethical values, leadership qualities, research capabilities and lifelong learning.
3. To educate students to be effective problem solvers, apply knowledge with social sensitivity for the betterment of the society and humanity as a whole.

2. List of PEOs, POs & PSOs

PEOs:

1. PEO-I: To provide students with a solid foundation in mathematics, engineering, basic science fundamentals required to solve computing problems and also to pursue higher studies and research.
2. PEO-II To train students with good Computer Science and Engineering breadth so as to comprehend, analyze, design and create innovative computing products and solutions for real life problems.
3. PEO-III To inculcate in students professional and ethical attitude, communication skills, teamwork skills, multi-disciplinary approach and an ability to relate computer engineering issues with social awareness.

POs:

1	PO1. Engineering knowledge: Apply the knowledge of basic sciences and fundamental engineering concepts in solving engineering problems.
2	PO2. Problem analysis: Identify and define engineering problems, conduct experiments and investigate to analyze and interpret data to arrive at substantial conclusions.
3	PO3. Design/development of solutions: Propose an appropriate solution for engineering problems complying with functional constraints such as economic, environmental, societal, ethical, safety and sustainability.
4	PO4. Conduct investigations of complex problems: Perform investigations, design and conduct experiments, analyze and interpret the results to provide valid conclusions.
5	PO5. Modern tool usage: Select or create and apply appropriate techniques and IT tools for the design & analysis of the systems.
6	PO6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7	PO7. Environment and sustainability: Demonstrate professional skills and contextual

	reasoning to assess environmental or societal issues for sustainable development.
8	PO8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9	PO9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multi-disciplinary situations.
10	PO10. Communication: Communicate effectively among engineering community, being able to comprehend and write effectively reports, presentation and give / receive clear instructions.
11	PO11. Project management and finance: Demonstrate and apply engineering & management principles in their own / team projects in multidisciplinary environment.
12	PO12. Life-long learning: Recognize the need for, and have the ability to engage in independent and lifelong learning.

PSOs:

1. PSO1: To provide effective and efficient real time solutions using acquired knowledge in various domains to crack problem using suitable mathematical analysis, data structure and suitable algorithm
2. PSO2: To develop environmental and sustainable engineering solution having global and societal context using modern IT tools.
3. PSO3: To exhibit professional and leadership skills with ethical values dealing diversified projects with excellent communication and documentation qualities.



2. Academic Calendar:



NARSIMHA REDDY ENGINEERING COLLEGE

(UGC-AUTONOMOUS)

(Sponsored by Jakkula Educational Society)

Maisammaguda (V), Dhulapally Post, Near Kompally, Secunderabad - 500 100 Telangana
Affiliated to JNTU, Approved by AICTE, New Delhi, Courses Accredited by NBA, NAAC with 'A' Grade, An ISO 9001:2015 Certified Institution

ACADEMIC CALENDAR FOR B.TECH II YEAR I SEMESTER FOR THE AY 2022-23

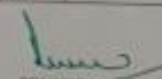
S.No.	Description	Duration		Duration (Weeks)
		From	To	
1	Commencement of I Semester class work	28-09-2022		
2	1 st Spell of Instructions (Including Dussera Vacation)	28-09-2022	26-11-2022	9
3	First Mid Term Examinations	28-11-2022	03-12-2022	1
4	Submission of Mid-I Marks	07-12-2022		
5	Parent-Teacher Meeting	10-12-2022		
6	2 nd Spell of Instructions	05-12-2022	28-01-2023	8
7	Second Mid Term Examinations	30-01-2023	04-02-2023	1
8	Submission of Mid-II Marks	07-02-2023		
9	Preparation Holidays & Lab Examinations	06-02-2023	11-02-2023	1
10	End Semester Examinations	13-02-2023	25-02-2023	2

ACADEMIC CALENDAR FOR B.TECH II YEAR II SEMESTER FOR THE AY 2022-23

S.No.	Description	Duration		Duration (Weeks)
		From	To	
1	Commencement of II Semester class work	27-02-2023		
2	1 st Spell of Instructions	27-02-2023	22-04-2023	8
3	First Mid Term Examinations	24-04-2023	29-04-2023	1
4	Submission of Mid-I Marks	03-05-2023		
5	Parent-Teacher Meeting	06-05-2023		
6	2 nd Spell of Instructions (Including 2 Week Summer Vacation)	01-05-2023	08-07-2023	10
7	Second Mid Term Examinations	10-07-2023	15-07-2023	1
8	Submission of Mid-II Marks	19-07-2023		
9	Preparation Holidays & Lab Examinations	17-07-2023	22-07-2023	1
10	End Semester Examinations	24-07-2023	05-08-2023	2

Copy to:

1. Chairman
2. IQAC
3. All HODs
4. Administrative Officer
5. Account officer
6. Web Portal I/C
7. ERP I/C
8. Library
9. Student Notice Boards


PRINCIPAL

NARSIMHA REDDY ENGINEERING COLLEGE
Survey No. 519, Maisammaguda (V), Dhulapally (P)
Medchal (M), Medchal Dist, Hyderabad-500100

3. SYLLABUS:

COMPUTER ORGANIZATION AND ARCHITECTURE

B.Tech. II Year I Semester								
Course Code	Category	Hours / Week			Credits	Maximum Marks		
CS2104PC	Core	L	T	P	C	CIA	SEE	Total
		3	0	0	3	30	70	100
Contact classes: 60	Tutorial Classes : NIL	Practical classes : NIL			Total Classes :60			
Prerequisites: No Prerequisites								

Course Objectives:

- The purpose of the course is to introduce principles of computer organization and the basic architectural concepts.
- It begins with basic organization, design and programming of a simple digital computer and introduces simple register transfer language to specify various computer operations.
- Topics include computer arithmetic, instruction set design, micro programmed control unit, pipelining and vector processing, memory organization and I/O systems and multiprocessors

Course Outcomes:

- Understand the basics of instructions sets and their impact on processor design.
- Demonstrate an understanding of the design of the functional units of a digital computer system.
- Evaluate cost performance and design trade-offs in designing and constructing a computer processor including memory.
- Design a pipeline for consistent execution of instructions with minimum hazards.
- Recognize and manipulate representations of numbers stored in digital computers

COURSE SYLLABUS

MODULE- I

Digital Computers: Introduction, Block diagram of Digital Computer, Definition of Computer Organization, Computer Design and Computer Architecture.

Register Transfer Language and Micro operations: Register Transfer language, Register Transfer, Bus and memory transfers, Arithmetic Micro operations, logic micro operations, shift micro operations, Arithmetic logic shift unit.

Basic Computer Organization and Design: Instruction codes, Computer Registers Computer instructions, Timing and Control, Instruction cycle, Memory Reference

Instructions, Input – Output and Interrupt.

MODULE- II

Microprogrammed Control: Control memory, Address sequencing, micro program example, design of control unit.

Central Processing Unit: General Register Organization, Instruction Formats, Addressing modes, Data Transfer and Manipulation, Program Control.

MODULE- III

Data Representation: Data types, Complements, Fixed Point Representation, Floating Point Representation.

Computer Arithmetic: Addition and subtraction, multiplication Algorithms, Division Algorithms, Floating–point Arithmetic operations. Decimal Arithmetic unit, Decimal Arithmetic operations.

MODULE- IV

Input-Output Organization: Input-Output Interface, Asynchronous data transfer, Modes of Transfer, Priority Interrupt Direct memory Access.

Memory Organization: Memory Hierarchy, Main Memory, Auxiliary memory, Associate Memory, Cache Memory.

MODULE- V

Reduced Instruction Set Computer: CISC Characteristics, RISC Characteristics.

Pipeline and Vector Processing: Parallel Processing, Pipelining, Arithmetic Pipeline, Instruction Pipeline, RISC Pipeline, Vector Processing, Array Processor.

MultiProcessors: Characteristics of Multiprocessors, Interconnection Structures, Interprocessor arbitration, Interprocessor communication and synchronization, cache Coherence.

TEXT BOOK:

1. Computer System Architecture–M.MorisMano, Third Edition, Pearson/PHI.

REFERENCE BOOKS:

1. Computer Organization–Carl Hamacher, Zvonks Vranesic, Safea Zaky, VthEdition, McGrawHill.
2. Computer Organization and Architecture – William Stallings Sixth Edition, Pearson/PHI.
3. Structured Computer Organization – Andrew S.Tanenbaum, 4thEdition, PHI/Pearson.

4. List of COs (Action verbs as per Bloom's Taxonomy)

Course Name: (CS2104PC)

Course Code.CO No	Course Outcomes (CO's)
At the end of the course student will be able to	
CS2104PC	Understand the basics of instructions sets and their impact on processor design.
CS2104PC	Demonstrate an understanding of the design of the functional units of a digital computer system.
CS2104PC	Evaluate cost performance and design trade-offs in designing and constructing a computer processor including memory.
CS2104PC	Design a pipeline for consistent execution of instructions with minimum hazards.
CS2104PC	Recognize and manipulate representations of numbers stored in digital computers

Course Outcome (CO)-Program Outcome (PO) Matrix: (2022-2023)

Course Name: CS2104PC

	PO1	PO[2]	PO[3]	PO[4]	PO[5]	PO[6]	PO[7]	PO[8]	PO[9]	PO[10]	PO[11]	PO[12]
CO[1]	2	2	3	2	3						2	2
CO[2]	2	3	1	1	2						3	2
CO[3]	1	2	2	2	1						1	3
CO[4]	2	1	3	2	2						2	2
CO[5]	2	2	1	3	3						1	2

Mapping of course outcomes with PSO's

CO PSO Mapping (2022-2023)

	PSO[1]	PSO[2]	PSO[3]
CO[1]	2	2	
CO[2]		2	
CO[3]	1	3	
CO[4]		1	2
CO[5]	2	2	1

5. Nominal Rolls of the Students:

II B.Tech – I Sem (2021 Batch):

CSE-A :

S.NO	ROLL NO	NAME OF THE STUDENT
1	21X01A0501	AEDULLA NIKHIL
2	21X01A0502	ALAKUNTA ARJUN
3	21X01A0503	AKIRI HEMANTHSAI
4	21X01A0504	AMBALA HARSHITH KUMAR
5	21X01A0505	BANDARUPALLI SRI SAI HARSHITHA
6	21X01A0506	BADDAM PRASHANTH REDDY
7	21X01A0507	BADDAM SHIVA NANDU REDDY
8	21X01A0508	BADUGU SANJAY KUMAR
9	21X01A0509	BANATHI NITHIN
10	21X01A0510	CHAKALA SUNIL
11	21X01A0511	CHALLA UDAY KIRAN
12	21X01A0512	CHILUKURI SUNIL
13	21X01A0513	DAMERA RATHAN PAUL
14	21X01A0514	DHARAVATH GANESH
15	21X01A0515	EDLA SAKETH
16	21X01A0516	GADARI ANUDEEP
17	21X01A0517	GADDE AKHIL
18	21X01A0518	GAJAGATTLA PRAVEEN
19	21X01A0519	GANGARAMAINA NITHIN
20	21X01A0520	GANGIDI AKANKSHA
21	21X01A0521	GODEPALLY POOJA
22	21X01A0522	JADA NAVEEN
23	21X01A0523	JANAGAMA VAMSHIKRISHNA
24	21X01A0524	K SWETHA
25	21X01A0525	KALAKUNTALA BHANUPRASAD
26	21X01A0526	KAMALAPURAM MEGHANA
27	21X01A0527	KAMSALI SURYATEJA ACHARI
28	21X01A0528	KAMUNI SAI SHASHINDRA
29	21X01A0529	KOTTADA ELIZABETH RANI
30	21X01A0530	KURELLI AKSHAYA
31	21X01A0531	KURRA SATISH KUMAR
32	21X01A0532	LACHULAGARI VAMSHI KRISHNA
33	21X01A0533	LOKA PUNITH REDDY
34	21X01A0534	M INDUSREE KATYAYANI
35	21X01A0535	M SHIVARAM

36	21X01A0536	MACHA MANOJ KUMAR
37	21X01A0537	MADDINENI HEMA SUNDAR
38	21X01A0538	MADISHETTY GAYATHRI
39	21X01A0539	MALGARI SAMPATH REDDY
40	21X01A0540	MAMIDI RUCHITHA
41	21X01A0541	M.V.SURYA
42	21X01A0542	MARRI MONIKA REDDY
43	21X01A0543	N V MANOJ
44	21X01A0544	NAGULA RAVI KIRAN
45	21X01A0545	NAGIGE SAITHEJA
46	21X01A0546	ORAGANTI GANESH
47	21X01A0547	P VIJAY
48	21X01A0548	PADALWAR SHRUTHI GOUD
49	21X01A0549	PANTHAGANI RATHNAM
50	21X01A0550	PANJALA SANGEETHA GOUD
51	21X01A0551	PESSU SNEHITH REDDY
52	21X01A0552	R VEENA
53	21X01A0553	S R M R T RATHNAKUMAR
54	21X01A0554	SADANANDE SHRUTHI
55	21X01A0555	SAKETH KANTE
56	21X01A0556	SANGU KRANTHI
57	21X01A0557	TAHOORA RAFI
58	21X01A0558	THALLAPALLY MAHESH
59	21X01A0559	TEKULAPALLY TARUN REDDY
60	21X01A0560	UDGIRE SAMEER SHADUL
61	21X01A0561	VEMIREDDY SRAVANTHI
62	21X01A0562	VEMIREDDY LIKITHA REDDY
63	21X01A0563	YELDI SRIVYBHAV
64	21X01A0564	YANNE ADAM BASHA
65	21X01A0565	YELLAIAHGARI ANUSHA
66	21X01A0566	YEDDANDI MANISHA REDDY
67	21X01A0567	YEDMALA PRATHYUSHA
68	22X05A0501	BANJA POOJA
69	22X05A0502	BASAVARAJU DURGA BHANUPRASAD
70	22X05A0503	DEERAVATH GANESH NAYAK

CSE- B

1	21X01A0568	AMBIDI AKSHITHA
2	21X01A0569	AMDHIPOOR VARUNGOUD
3	21X01A0570	AREPALLI MANOJ
4	21X01A0571	BATHULA SAI KIRAN
5	21X01A0572	BOOJALA SOURYA

6	21X01A0573	BARKAM RISHIK
7	21X01A0574	BHEEMANATHI HARINI
8	21X01A0575	BHUKYA JASVANTH
9	21X01A0576	BODA DIVYASRI
10	21X01A0577	CHALLA NAGALAKSHMI
11	21X01A0578	CHINTHALAPELly SAHASRA REEDY
12	21X01A0579	CHIRRA BALAKRISHNA REDDY
13	21X01A0580	DRAKSHARAM SOWMYA
14	21X01A0581	DUNGU SUBROTO CHAKRAVARTHY
15	21X01A0582	GOGULA LAXMI PRASANNA
16	21X01A0583	GOPISETTI VIVEK SAI
17	21X01A0584	GOSULA RAJKUMAR
18	21X01A0585	GUDA MEGHANA
19	21X01A0586	GOLI SAI KIRAN
20	21X01A0587	GOLKONDA PRANAY
21	21X01A0588	ISLAVATH ABHISHEK NAYAK
22	21X01A0589	JANGA MAHESH
23	21X01A0590	JARPULA SRAVAN KUMAR
24	21X01A0591	KANNABATHULA VENKAT SAI
25	21X01A0592	KANUGANTI HARI PRIYA
26	21X01A0593	KANUKULA STANLY
27	21X01A0594	KARINGU SHIVASHANKAR
28	21X01A0595	KARINGULA KAVYA SRI
29	21X01A0596	KASHISH PAREKH
30	21X01A0597	KOMMULA MARUTHI
31	21X01A0598	KOMPELLA VENKATA SUBRAHMANYA SHARAT CHANDRA
32	21X01A0599	KONDE CHANDANA
33	21X01A05A0	MAMIDIPALLY ASHWITHA
34	21X01A05A1	MANDHA NAVEEN
35	21X01A05A2	MANTRI SOWMYA
36	21X01A05A3	MASKURI SINDHU
37	21X01A05A4	MIDDELA GOPI SAGAR
38	21X01A05A5	MOHAMMAD SHAREEF
39	21X01A05A6	MOHAMMED NAUSHAD MOHIUDDIN
40	21X01A05A7	MATHYA HEPSIBHA
41	21X01A05A8	MUKUL REDDY ANAGANDULA
42	21X01A05A9	NARENDRAPURAPU TEJASWI
43	21X01A05B0	NEDUNURI AVINASH
44	21X01A05B1	NAKIRI VENKATESH
45	21X01A05B2	OTARKAR SAI KUMAR

46	21X01A05B3	PASHAM VAMSHI
47	21X01A05B4	PATHIREDDY ABINAYA
48	21X01A05B5	PATNAM VAMSHI
49	21X01A05B6	PODILAPU HARADEEP
50	21X01A05B7	POWER NAVEEN
51	21X01A05B8	RAPARTHI SAI KIRTHI
52	21X01A05B9	RAVALKOL PAVAN KUMAR
53	21X01A05C0	SAMALA VYSHNAVI
54	21X01A05C1	SAMPANGI VILAKAR
55	21X01A05C2	SANGAM SRINIDHI
56	21X01A05C3	SATTI GOWTHAM RAVINDRA REDDY
57	21X01A05C4	THATI BHARATH
58	21X01A05C5	TUNGALA NEERAJ
59	21X01A05C6	THALLURI CHAKRAVARTHI
60	21X01A05C7	VEMIREDDY RAM DINESH REDDY
61	21X01A05C8	VELPULA ABHINAV
62	21X01A05C9	VADLA LAXMI NARASIMHA
63	21X01A05D0	VELMA SAI CHARAN REDDY
64	21X01A05D1	YEMULA SRICHARAN
65	21X01A05D2	YENAGANDULA RAHUL
66	21X01A05D3	YELLAMMALA RAMYA
67	21X01A05D4	YERRA SANJANA
68	22X05A0504	DUSSA GANESH
69	22X05A0505	GATTU SAI PRAKASH
70	22X05A0506	GOLLAPALLI DINESH KUMAR GOUD

CSE- C

S.NO	ROLL NO	NAME OF THE STUDENT
1	21X01A05D5	ANEDLA AKSHITHA
2	21X01A05D6	ANUMAS MOUNIKA
3	21X01A05D7	ATLA SRUJANA
4	21X01A05D8	BADDIPADIGA DURGA BHAVANI
5	21X01A05D9	BOIN RAMU
6	21X01A05E0	BOLLEPELLE VIGNESH
7	21X01A05E1	BURSU KAMESHWARA RAO
8	21X01A05E2	BURUJU SIDDARTHA REDDY
9	21X01A05E3	CHEDE RUCHITHA
10	21X01A05E4	CHILUVERI JAISH
11	21X01A05E5	CHITAMANENI SAI TANUJ
12	21X01A05E6	CHITYALA KOUSHIK REDDY
13	21X01A05E7	CHUKKA RAJENDRA

14	21X01A05E8	DYAGARI PRIYA
15	21X01A05E9	GUGULOTH MADHU CHANDANA
16	21X01A05F0	GUMPULA NAVEEN
17	21X01A05F1	GUNDE RAKESH
18	21X01A05F2	GUNDEBOINA SNEHA
19	21X01A05F3	GUNTAKULAM SAI NIKITHA
20	21X01A05F4	GOURU SAI PAVAN
21	21X01A05F5	
22	21X01A05F6	JILAKARA ADITHYA
23	21X01A05F7	JAKKULA GANGAMANI
24	21X01A05F8	KASUKURTHI SUPRIYA
25	21X01A05F9	KONDAIAHGARI MANISHA
26	21X01A05G0	KOTAGIRI PRANAY KUMAR
27	21X01A05G1	KOTTALA RAJITHA
28	21X01A05G2	KURRA ASRITHA
29	21X01A05G3	KUSUMA RAMYA
30	21X01A05G4	KAGGA GOPICHAND
31	21X01A05G5	KATARI BHAVANI PRASAD VARMA
32	21X01A05G6	KATTA ASHRITHA REDDY
33	21X01A05G7	MUMMANENI SREETHARATHNAM
34	21X01A05G8	MOULIK PATEL
35	21X01A05G9	MUDAVATH SANTHOSH
36	21X01A05H0	MUDDADA RAVI SANKAR
37	21X01A05H1	MULGI RUDRAKSHA
38	21X01A05H2	MUNFED ALI
39	21X01A05H3	MUTHYAM NIKSHITHA
40	21X01A05H4	MUTTUM UMESH
41	21X01A05H5	NIMMAKANTI MADHUSAGAR
42	21X01A05H6	NIRUDI DINESH
43	21X01A05H7	NIRUDI VAISHNAVI
44	21X01A05H8	NIMMALA MANOJ
45	21X01A05H9	BOLLA SRIKAR
46	21X01A05I0	PAYYAVULA DINESH KUMAR
47	21X01A05I1	PERUPOGU SANDEEP
48	21X01A05I2	PS VENNALA VEDASHINEE
49	21X01A05I3	PUPPALA RISHIKESH
50	21X01A05I4	ROUTHU PRABHAKAR
51	21X01A05I5	SHAIK AKASH
52	21X01A05I6	SHAIK SOHEL
53	21X01A05I7	SIDDIQUA TABASSUM
54	21X01A05I8	SIRIMALLA SIRI CHANDANA

55	21X01A05I9	SIRIPURAM VINAY KUMAR
56	21X01A05J0	SHIVARATHRI VIGNESHWAR
57	21X01A05J1	SNEHA TRIPATHY
58	21X01A05J2	TURKANI LALAPPA
59	21X01A05J3	THOTAKURA SHRAVYA
60	21X01A05J4	KADMURI KAUSHIK KUMAR VARMA
61	22X05A0507	GSANDEEP
62	22X05A0508	KAGNALI NANDINI
63	22X05A0509	KAITHALAPURAM HEMANTH
64	22X05A0510	KATAM ASHWINI
65	22X05A0511	KORPATHI HEMALATHA
66	22X05A0512	MAHESHWARAM JITHENDHER
67	22X05A0513	PODILA NAVA BHARATH KUMAR
68	22X05A0514	R SAI VARDHAN
69	22X05A0515	RODDA BHANU PRASAD
70	22X05A0516	S MANJUNADHA
71	22X05A0517	SATHU AJAY REDDY
72	22X05A0518	SHAIK SURAJ BABA
73	22X05A0519	SINAMGARAM CHANDU YADAV
74	22X05A0520	THUPAKULA VAMSHI YADAV



6. TIME TABLE (2022-2023)

NARSIMHA REDDY ENGINEERING COLLEGE
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
TIME TABLE

CLASS: II -YEAR CSE -A I-SEM (2022-2023)
CLASS INCHARGE: Dr. N. SRINIVAS RAO

ROOM NUMBER: 111
II YEAR INCHARGE: Dr. N. SRINIVAS RAO

WEF: 28-09-2022

	1	2	3	4	12:50PM - 1:40PM	5	6	7
HOUR/DAY	9:30AM - 10:20AM	10:20AM - 11:10AM	11:10AM - 12:00PM	12:00PM - 12:50PM		1:40PM - 2:30PM	2:30PM - 3:20PM	3:20PM - 4:10PM
MON	ADE	COA	C++	C++	L U N C H	DS LAB		
TUE	COA	COA	COSM	COSM		DS	ADE	ADE
WED	COSM	DS	DS	COA		C++ LAB		
THU	C++	DS	DS	ADE		ITWS LAB		
FRI	C++	ADE LAB				COSM	COSM	SPORTS
SAT	GENDER SENSITIZATION					LIBRARY		SPORTS

S.NO	COURSE CODE	COURSE TITLE	FACULTY
1	EC2101ES	Analog and Digital Electronics	K.ANNAMMA
2	CS2102PC	Data Structures	Dr. U. MOHAN SRINIVAS
3	MA2103BS	Computer Oriented Statistical Methods	G.SHANDHIYA RANI
4	CS2104PC	Computer Organization and Architecture	Dr. N. SRINIVAS RAO
5	CS2105PC	Object Oriented Programming using C++	D. GNANAPRASANNA
6	EC2106ES	Analog and Digital Electronics Lab	K.ANNAMMA
7	CS2107PC	Data Structures Lab (Room no. 007)	Dr. N. SRINIVAS RAO, DR. R. RAJAGOPAL, P.RAMPRASAD, S. SHAKINA
8	CS2108PC	IT Workshop lab (Room No. 107)	A.BARKATHULLA, P.RAMPRASAD
9	CS2109PC	C++ Programming Lab (Room no. 108)	D. GNANAPRASANNA, S. SHAKINA, DR. R. RAJAGOPAL
10	MC2002*	Gender Sensitization Lab	S. SHAKINA, P.RAMPRASAD

Time Table Coordinator(s)

HOD

PRINCIPAL

DR. N. SRINIVAS RAO

Survey No: 515, Mahasaraksethoda (V),

09/10/2022

Time Table Coordinator(s)

HOD

PRINCIPAL
NARSIMHA REDDY ENGINEERING COLLEGE
Survey No. 518, Malsammoguda (V),
Medchal (M), Medchal Dist. Hyderabad

NARSIMHA REDDY ENGINEERING COLLEGE
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
TIME TABLE

CLASS: II-YEAR CSE -B I-SEM (2022-2023)

CLASS INCHARGE: Dr. R.RAJAGOPAL

ROOM NUMBER: 112

II YEAR INCHARGE: Dr. N. SRINIVAS RAO

WEF: 28.09.2022

	1	2	3	4	12:50PM - 1:40PM	5	6	7
HOUR/DAY	9:30AM - 10:20AM	10:20AM - 11:10AM	11:10AM - 12:00PM	12:00PM - 12:50PM		1:40PM - 2:30PM	2:30PM - 3:20PM	3:20PM - 4:10PM
MON	COSM	COSM	DS	DS	L U N C H	ADE	C++	C++
TUE	C++	C++ LAB				COA	COA	COSM
WED	ADE	COA	C++	C++		ITWS LAB		
THU	COSM	ADE LAB				COA	DS	SPORTS
FRI	ADE	ADE	COSM	DS		DS LAB		
SAT	GENDER SENSITIZATION					LIBRARY		SPORTS

S.NO	COURSE CODE	COURSE TITLE	FACULTY
1	EC2101ES	Analog and Digital Electronics	K.ANNAMMA
2	CS2102PC	Data Structures	Dr. U. MOHAN SRINIVAS
3	MA2103BS	Computer Oriented Statistical Methods	G.SHANDHIYA RANI
4	CS2104PC	Computer Organization and Architecture	Dr. N. SRINIVAS RAO
5	CS2105PC	Object Oriented Programming using C++	D. GNANAPRASANNA
6	EC2106ES	Analog and Digital Electronics Lab	K.ANNAMMA
7	CS2107PC	Data Structures Lab (Room no. 007)	Dr. N. SRINIVAS RAO, DR. R. RAJAGOPAL, P.RAMPRASAD, S. SHAKINA
8	CS2108PC	IT Workshop lab (Room No. 107)	A.BARKATHULLA, P.RAMPRASAD
9	CS2109PC	C++ Programming Lab (Room no. 108)	D. GNANAPRASANNA, S. SHAKINA, A.BARKATHULLA
10	MC2002*	Gender Sensitization Lab	S. SHAKINA, P.RAMPRASAD

(Signature)
Time Table Coordinator(s)

(Signature)
HOD

PRINCIPAL
(Signature)
 09/10/2022

Time Table Coordinator(s)

HOD

PRINCIPAL
NARSIMHA REDDY ENGINEERING COLLEGE
Survey No. 518, Malsammoguda (V),
Medchal (M), Medchal Dist. Hyderabad

NARSIMHA REDDY ENGINEERING COLLEGE
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

TIME TABLE

CLASS: II-YEAR CSE -C I-SEM (2022-2023)

ROOM NUMBER: 207

WEEK: 28-09-2022

CLASS INCHARGE: D. GNANAPRASANNA

II YEAR INCHARGE: Dr. N. SRINIVAS RAO

GENS ENLIGHTENED - GAINING KNOWLEDGE					12:50PM – 1:40PM L U N C H	5	6	7
HOUR/DAY	1 9:30AM - 10:20AM	2 10:20AM - 11:10AM	3 11:10AM - 12:00PM	4 12.00PM- 12.50PM		1:40PM - 2:30PM	2:30PM - 3:20PM	3:20PM - 4:10PM
MON	C++	DS	COA	COSM		ADE	COSM	DS
TUE	COSM	DS	DS	ADE		ITWS LAB		
WED	C++ ADE LAB					COSM	COSM	ADE
THU	COA	COA	C++	C++		C++ LAB		
FRI	COA	DS LAB				ADE	DS	SPORTS
SAT	GENDER SENSITIZATION					LIBRARY		SPORTS

S.NO	COURSE CODE	COURSE TITLE	FACULTY
1	EC2101ES	Analog and Digital Electronics	D.YESHASREE
2	CS2102PC	Data Structures	Dr. U. MOHAN SRINIVAS
3	MA2103BS	Computer Oriented Statistical Methods	G.SHANDHIYA RANI
4	CS2104PC	Computer Organization and Architecture	Dr. N. SRINIVAS RAO
5	CS2105PC	Object Oriented Programming using C++	D. GNANAPRASANNA
6	EC2106ES	Analog and Digital Electronics Lab	D.YESHASREE
7	CS2107PC	Data Structures Lab (Room no. 007)	Dr. N. SRINIVAS RAO, DR. R.RAJAGOPAL, P.RAMPRASAD, S. SHAKINA
8	CS2108PC	IT Workshop lab (Room No. 107)	A.BARKATHULLA, P.RAMPRASAD
9	CS2109PC	C++ Programming Lab (Room no. 108)	D. GNANAPRASANNA, S. SHAKINA, DR. R.RAJAGOPAL
10	MC2002*	Gender Sensitization Lab	S.SHAKINA, P.RAMPRASAD

Time Table Coordinator(s)

HOD

09/10/2022 11:45
NARSIMHA REDDY ENGINEERING COLLEGE
Survey No: 518, Maisarancha, N. Taluk, Nellore, A.P.

7. LECTURE PLAN (2022-2023)

S.No	Tentative Date	Topics as per Syllabus	Topic Actually Covered	Suggested Book	Method of Teaching BB/PPT
UNIT - I					
1	29/09/2022	Digital Computers: Introduction	Digital Computers: Introduction,	T1	BB/PPT
2	30/09/2022	Block diagram of Digital Computer	Block diagram of Digital Computer	T1	BB/PPT
3	10/10/2022	Definition of Computer Organization	Definition of Computer Organization	T1	BB/PPT
4	11/10/2022	Computer Design and Computer Architecture.	Computer Design and Computer Architecture.	T1	BB/PPT
5	11/10/2022	Register Transfer	Register Transfer	T1	BB/PPT

		Language and Micro operations	Language and Micro operations		
6	13/10/2022	Register Transfer language,	Register Transfer language,	T1	BB/PPT
7	14/10/2022	Register Transfer	Register Transfer	T1	BB/PPT
8	17/10/2022	Bus and memory transfers	Bus and memory transfers	T1	BB/PPT
9	18/10/2022	Arithmetic Micro operations	Arithmetic Micro operations	T1	BB/PPT
10	18/10/2022	logic micro operations	shift micro operations	T1	BB/PPT
11	20/10/2022	shift micro operations	shift micro operations	T1	BB/PPT
12	21/10/2022	Arithmetic logic shift unit	Arithmetic logic shift unit	T1	BB/PPT
13	25/10/2022	Basic Computer Organization and Design	Basic Computer Organization and Design	T1	BB/PPT
14	25/10/2022	Instruction codes, Computer Registers, Computer instructions,	Instruction codes, Computer Registers, Computer instructions,	T1	BB/PPT
15	27/10/2022	Timing and Control, Instruction cycle,	Timing and Control, Instruction cycle,	T1	BB/PPT
16	28/10/2022	Memory Reference Instructions, Input – Output and Interrupt.	Memory Reference Instructions, Input – Output and Interrupt.	T1	BB/PPT
UNIT - II					
17	31/10/2022	Micro programmed Control: Control memory	Micro programmed Control: Control memory,	T1	BB/PPT

18	01/11/2022	Address sequencing,	Address sequencing ,	T1	BB/PPT
19	01/11/2022	micro program example	micro program example	T1	BB/PPT
20	03/11/2022	design of control unit	design of control unit	T1	BB/PPT
21	04/11/2022	Central Processing Unit: General Register Organization	Central Processing Unit: General Register Organization	T1	BB/PPT
22	07/11/2022	Instruction Formats, Addressing modes	Instruction Formats, Addressing modes	T1	BB/PPT
23	10/11/2022	Data Transfer and Manipulation, Program Control.	Data Transfer and Manipulation, Program Control.	T1	BB/PPT
UNIT - III					
24	11/11/2022	Data Representation: Data types, Complements	Data Representation: Data types, Complements	T1	BB/PPT
25	14/11/2022	Fixed Point Representation	Fixed Point Representation	T1	BB/PPT
26	15/11/2022	, Floating Point Representation	, Floating Point Representation	T1	BB/PPT
27	15/11/2022	Computer Arithmetic: Addition and subtraction	Computer Arithmetic : Addition and subtraction	T1	BB/PPT
28	17/11/2022	multiplication Algorithms	multiplication Algorithms	T1	BB/PPT
29	18/11/2022	multiplication Algorithms	multiplication Algorithms	T1	BB/PPT
30	21/11/2022	multiplication Algorithms	multiplication Algorithms	T1	BB/PPT

31	22/11/2022	Division Algorithms	Division Algorithms	T1	BB/PPT
32	22/11/2022	Floating-point Arithmetic operations	Floating-point Arithmetic operations	T1	BB/PPT
33	24/11/2022	Decimal Arithmetic unit	Decimal Arithmetic unit	T1	BB/PPT
34	25/11/2022	Decimal Arithmetic operations	Decimal Arithmetic operations	T1	BB/PPT
35	05/12/2022	Decimal Arithmetic operations	Decimal Arithmetic operations	T1	BB/PPT
39	06/12/2022	Decimal Arithmetic operations	Decimal Arithmetic operations	T1	BB/PPT
UNIT - IV					
40	06/12/2022	Input-Output Organization: Input-Output Interface	Input-Output Organization: Input-Output Interface	T1	BB/PPT
41	08/12/2022	Asynchronous data transfer	Asynchronous data transfer	T1	BB/PPT
42	09/12/2022	Modes of Transfer,	Modes of Transfer,	T1	BB/PPT
43	12/12/2022	Priority Interrupt	Priority Interrupt	T1	BB/PPT
44	13/12/2022	Direct memory Access.	Direct memory Access.	T1	BB/PPT
45	13/12/2022	Memory Organization: Memory Hierarchy	Memory Organization: Memory Hierarchy	T1	BB/PPT
46	15/12/2022	Main Memory,	Main Memory,	T1	BB/PPT
47	16/12/2022	Auxiliary memory	Auxiliary memory	T1	BB/PPT
48	19/12/2022	Associate Memory,	Associate Memory,	T1	BB/PPT
49	20/12/2022	Cache Memory.	Cache Memory.	T1	BB/PPT
UNIT V					
50	22/12/2022	Reduced Instruction Set Computer: CISC	Reduced Instruction Set	T1	BB/PPT

		Characteristics,	Computer: CISC Characteristics,		
51	23/12/2022	RISC Characteristics.	RISC Characteristics.	T1	BB/PPT
52	26/12/2022	Pipeline and Vector Processing: Parallel Processing	Pipeline and Vector Processing: Parallel Processing	T1	BB/PPT
53	27/12/2022	Pipelining, Arithmetic Pipeline,	Pipelining, Arithmetic Pipeline,	T1	BB/PPT
54	29/12/2022	Instruction Pipeline	Instruction Pipeline	T1	BB/PPT
55	02/01/2023	RISC Pipeline	RISC Pipeline	T1	BB/PPT
56	03/01/2023	Vector Processing,	Vector Processing,	T1	BB/PPT
57	05/01/2023	Vector Processing	Vector Processing	T1	BB/PPT
58	02/01/2023	Array Processor.	Array Processor.	T1	BB/PPT
59	06/01/2023	Array Processor.	Array Processor.	T1	BB/PPT
60	09/01/2023	MultiProcessors: Characteristics of Multiprocessors	MultiProcessors: Characteristics of Multiprocessors	T1	BB/PPT
61	10/01/2023	Interconnection Structures	Interconnection Structures	T1	BB/PPT
62	12/01/2023	Interconnection Structures	Interconnection Structures	T1	BB/PPT
63	16/01/2023	Interprocessor arbitration	Interprocessor arbitration	T1	BB/PPT
64	17/01/2023	Interprocessor arbitration	Interprocessor arbitration	T1	BB/PPT
65	19/01/2023	Interprocessor communication and synchronization, cache Coherence.	Interprocessor communication and synchronization, cache Coherence.	T1	BB/PPT
66	20/01/2023	Unit Test 1	Unit Test 1	T1	BB/PPT
67	23/01/2023	Unit Test 2	Unit Test 2		

68	24/01/2023	Unit Test 3	Unit Test 3		
69	24/01/2023	Unit Test 4	Unit Test 4		
70	27/01/2023	Unit Test 5	Unit Test 5		

TEXT BOOK:

1. Computer System Architecture–M.MorisMano, Third Edition, Pearson/PHI.

REFERENCE BOOKS:

1. Computer Organization–Carl Hamacher, Zvonks Vranesic, Safea Zaky, VthEdition, McGrawHill.
2. Computer Organization and Architecture – William Stallings Sixth Edition, Pearson/PHI.
3. Structured Computer Organization – Andrew S.Tanenbaum, 4thEdition, PHI/Pearson.

8. University Questions / Question Bank

Unit 1:

1. What is RTL?
2. What is Micro operation? Give examples.
3. Define Computer Architecture
4. Define Computer Organization
5. Show the block diagram of the hard ware that implements the following register transfer statement: $yT2: R2 \rightarrow R1, R1 \leftarrow R2$.
6. What are the basic symbols for register transfers.
7. What is control function?
8. What is a bus? What are different buses in a CPU?
- 9.. What is memory and what are the operations of memory.
10. Give RTL statements for memory transfers.
11. Explain about the following operation with an example.
 - a. arithmetic micro operation
 - b. Logic micro operation
 - c. shift micro operation
12. Draw the circuit diagram of 4-bit binary adder.

13. Draw the circuit diagram of 4-bit adder/subtractor.
14. Draw the circuit diagram of 4-bit binary incrementer.
15. Draw the circuit diagram of 4-bit binary decrements.
16. What is an arithmetic circuit ?
17. List and the applications of logic micro operations.
18. Define selective set with an example.
19. Define selective clear with an example.
20. Define selective complement with an example.
21. Define selective mask with an example.
22. Discuss an arithmetic shift micro operation.
23. Explain the logical shift micro operation.
24. What is circular shift micro operation.
25. Define effective address.

Unit 2:

1. What are the ways of designing a control unit? Explain
2. Distinguish between hardwired and micro programmed control.
3. What size of decoder is used for designing of control unit?
4. What is the purpose of control memory?
7. What is the purpose of CAR(control address register) ,CDR(control data register). And SBR (subroutine register)
8. What is the function of micro program sequencer.
9. What is the purpose of pipe line register?
10. What is a micro routine OR what is control store?
11. Define mapping process in address sequencing.
12. Define conditional branching.
13. What is branch logic hardware?
14. Draw the diagram for mapping instruction code to microinstruction address.

15. Give the micro instruction code format.
16. What is a condition field?
17. Give some symbols and micro instruction code for micro instruction fields.
18. Define the functions of JMP, CALL, RET, MAP instructions.
19. What are the differences between the main memory and control memory?
20. How many micro instructions are needed for Fetch and decode routine.
21. What are the address sequencing capabilities required in a control memory.
22. What are the different branching techniques used in control unit.
23. What is vertical organization and horizontal organization?
24. What is a computer instruction?
25. What is an instruction code?
26. What is an operation code?
27. Define the effective address (EA)
28. What is a stack?
29. What is a stack pointer?
30. What are the operations of a stack?
31. What is push and pop?
32. What are the types of stack?
33. What is the polish notation?
34. What is the reverse polish notation (RPN)?
35. What are the fields of an instruction format?
36. What is a register address?
37. What are the types of CPU organizations?
38. What is an addressing mode?
39. What is the use of addressing modes in computers?

Unit 3:

1. Explain the data types and complements.

2. How floating point numbers are represented? Give example.
3. Discuss the floating Point representation.
4. What is a hexadecimal number system?
5. What is the role of the Accumulator?
6. Perform the subtraction with the following unsigned binary number by taking the 2's complement of the subtrahend

11010-10000

7. What is the advantage of using Booth algorithm?
8. Explain the functions of CPU.
9. What kind of number system does computer use?
10. Write overflow conditions for addition and subtraction.
11. Give a brief note on Division Algorithms.
12. Give the format of floating point numbers.
13. Draw the diagram for hardware for signed magnitude addition and subtraction.
14. What is the need of Booth's multiplication algorithm.
15. Give the procedure for Booth's multiplication algorithm.
16. What is 2-bit by 2-bit array multiplier.
17. Explain Floating-point Arithmetic operations.
18. Discuss the decimal Arithmetic operations

Unit 4:

1. Give the block diagram of interface between a processor and peripheral devices and explain its operations.
2. Explain Daisy-chaining method of establishing priority with the help of diagram.
3. List four peripheral devices that produce an acceptable output for a person to understand.
4. Discuss the different methods of data transfer between the CPU and I/O devices.
5. Design parallel priority interrupt hardware for a system with eight interrupt sources.
6. How many characters per second can be transmitted over a 1200 baud line in each of the following modes?

a)Synchronous serial transmission

b)Asynchronous serial transmission with two stop bits.

c)Asynchronous serial transmission with one stop bits

7.Why are the read and write control lines in a DMA controller bidirectional? Under what condition and for what purpose are they are they used as inputs? Under what condition and for what purpose are they are they used as outputs?

8. The access time of a cache memory is 100 ns and that of main memory is 1000 ns. It is estimated that 80% of the memory requests are for read and the remaining 20% are for write. The hit ratio for read accesses only is 0.9. A write-through procedure is used.

(i). What is the average access time of the system considering only memory read cycles?

(ii). What is the average access time of the system for both read and write requests?

Unit 5:

1.Compare and contrast

i. Multiprocessors and multicomputer systems.

ii. Tightly coupled and loosely coupled multiprocessors.

iii. Synchronous and asynchronous bus

2.Differentiate between parallel processing and pipeline processing with suitable examples.

3.Construct a diagram for a 4 x 4 omega switching network. Show the switch setting required to connect input 3 to output 1.

4.“Multiprocessing can improve performance by decomposing a program into parallel executable tasks.” Explain how this is achieved.

5.What are the various physical forms available for establishing an interconnection network? Explain

6.Draw and explain the structure of general purpose multicomputer.

7.Explain the characteristics of CISC and RISC.

8.What is an instruction pipeline? What are the difficulties that cause the instruction pipeline to deviate from its normal operation?

9.Draw a space time diagram for six segment pipeline showing the time it takes to process eight tasks.

10Explain four possible hardware schemes that can be used in an instruction pipeline in order to minimize the performance degradation caused by instruction branching.

11.Consider the multiplication of two 40*40 matrices using a vector processor.

a) .How many product terms are there in each inner product and how many inner products must be evaluated?

b) How many multiply add operations are needed to calculate the product matrix?

12. Explain the parallel processing architecture and its uses.

9. Old Question Papers:

IV B.Tech I Semester Supplementary Examinations, October/November-2019

COMPUTER ARCHITECTURE AND ORGANIZATION

(Common to Electronics and Communication Engineering and Electronics and Instrumentation Engineering)

Time: 3 hours

Max. Marks: 70

Question paper consists of Part-A and Part-B

Answer ALL sub questions from Part-A

Answer any THREE questions from Part-B

PART-A (22 Marks)

1. a) What is the difference between the restoring and non-restoring method of division? [4]
- b) What are the types of micro operations? [3]
- c) What is a control word? [3]
- d) What is Memory system? Define Memory refreshing. [4]
- e) Define intra segment and inter segment communication. [4]
- f) What is bus arbitration? Explain. [4]

PART-B (3x16 = 48 Marks)

2. a) Explain the architecture of a basic Computer. [6]
- b) Distinguish between multiprocessors and multi computers. [4]
- c) Explain the Booth's algorithm for multiplication of signed two's complement numbers. [6]
3. a) Explain the Differences between CISC and RISC. [8]
- b) Discuss about Memory Reference Instructions. [8]
4. a) Explain the basic organization of a micro programmed control unit and the generation of control signals using micro program. [8]
- b) Describe the control unit organization with a separate Encoder and Decoder functions in a hardwired control. [8]
5. a) What do you mean by virtual memory? Discuss how paging helps in implementing virtual memory. [8]
- b) Discuss any six ways of improving the cache performance. [8]
6. a) Discuss about priority interrupt. [8]
- b) Explain about Input-output interface. [8]

10. Question Papers (CIA & SEE)

Mid exam question papers:



ACCREDITED BY NBA & NAAC WITH A-GRADE
NARSIMHA REDDY ENGINEERING COLLEGE
PERMANENTLY AFFILIATED TO JNTUH, HYDERABAD - APPROVED BY AICTE, NEW DELHI
AN ISO 9001 : 2008 CERTIFIED INSTITUTE



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
II-B.TECH I SEMESTER – I MID EXAMINATION
SET - B

SUBJECT: Computer Organization and Architecture
MAX. MARKS: 10

DATE:
TIME: 2.00PM-3.30PM

ANSWER ANY TWO QUESTIONS

2*5=10M

S.No	Question	CO	BL	POs
1.	a) Draw the figure to show how functional units are interconnected using a bus and explain.	1	3	PO1,PO3,PO11
	b) List and explain the functions of various components	1	2	PO2,PO2,PO5
2.	a) Explain about Stack Organization in detail.	1	4	PO3,PO1,PO12
	b) Discuss the generic Instruction types present in a computer system.	1	3	PO2,PO5,PO4
3	a) Describe the Data Transfer and Manipulation.	1	2	PO2,PO4,PO5
	b) Explain the Instruction Formats.	2	3	PO1,PO3,PO6
4.	a) Elaborate the Floating Point Representation.	2	4	PO1,PO2,PO5
	b) Illustrate the Fixed Point Representation.	2	2	PO3,PO4,PO12

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
II-B.TECH I SEMESTER – II MID EXAMINATION
SET - B

SUBJECT: Computer Organization and Architecture

DATE:

MAX. MARKS: 10

TIME: 10.00AM-11.30PM

ANSWER ANY TWO QUESTIONS

2*5=10M

S.No	Question	CO	BL	POs
1.	a) Compare horizontal and vertical organization. Give their advantages and disadvantages.	3	3	PO1,PO2,PO12
	b) What do you understand by symbolic micro instruction? Give the typical field distribution of a symbolic micro instruction and explain the significance of each field.	3	2	PO2,PO1,PO5
2.	a) When a device interrupt occurs, how does the processor determine which device issued the interrupt? Explain.	4	4	PO3,PO2,PO11
	b) Explain the Decimal Arithmetic unit.	4	3	PO1,PO5,PO4
3.	a) Discuss the CISC Characteristics.	4	2	PO2,PO4,PO6
	b) List and explain the RISC Characteristics.	5	3	PO2,PO4,PO6
4.	a) Elaborate the Vector Processing and Array Processor.	5	4	PO1,PO2,PO5
	b) Discuss the Characteristics of Multiprocessors.	5	2	PO3,PO4,PO12

11. Tutorial Sheets

12. I. Assignment Questions (2022-2023)



ACCREDITED BY NBA & NAAC WITH A-GRADE
NARSIMHA REDDY ENGINEERING COLLEGE
PERMANENTLY AFFILIATED TO JNTUH, HYDERABAD - APPROVED BY AICTE, NEW DELHI
AN ISO 9001 : 2008 CERTIFIED INSTITUTE



DEPARTMENT OF CSE

II-B.TECH I SEMESTER- ASSIGNMENT: I

SUBJECT: COMPUTER ORGANIZATION AND ARCHITECTURE

S.No		Question	CO	BL	POs
1.	a)	Define computer. Specify the different types of computers and their characteristics.	1	4	PO2,PO3,PO11
	b)	Explain how the floating-point numbers are represented and used in digital arithmetic operations. Give an example.	1	3	PO2,PO3,PO5
2.	a)	What is a bus? Draw the figure to show how functional units are interconnected using a bus and explain.	1	2	PO4,PO2,PO12
	b)	List and explain the functions of various components	1	3	PO1,PO5,PO6
3	a)	Explain about Stack Organization in detail.	1	2	PO2,PO4,PO5
	b)	Discuss the generic Instruction types present in a computer system.	2	4	PO4,PO3,PO6
4.	a)	Describe the Data Transfer and Manipulation.	2	4	PO3,PO2,PO6
	b)	Explain the Instruction Formats.	2	2	PO3,PO4,PO11
5	a)	Elaborate the Floating Point Representation.	3	2	PO4,PO3,PO12
	b)	Illustrate the Fixed Point Representation.	3	3	PO2,PO4,PO7

DEPARTMENT OF CSE

II-B.TECH I SEMESTER- ASSIGNMENT: 2

SUBJECT: COMPUTER ORGANIZATION AND ARCHITECTURE

S.No		Question	CO	BL	POs
1.	a)	When a device interrupt occurs, how does the processor determine which device issued the interrupt? Explain.	3	2	PO2,PO2,PO11
	b)	A DMA module is transferring the characters to memory using cycle stealing, from a device transmitting at 9600 bps. The processor is fetching instructions at the rate of 1MIPS. By how much will the processor be slowed down due to DMA activity?	3	2	PO2,PO2,PO5
2.	a)	Compare horizontal and vertical organization. Give their advantages and disadvantages.	3	3	PO4,PO2,PO12
	b)	What do you understand by symbolic micro instruction? Give the typical field distribution of a symbolic micro instruction and explain the significance of each field.	4	3	PO1,PO5,PO6
3	a)	When a device interrupt occurs, how does the processor determine which device issued the interrupt? Explain.	4	2	PO2,PO4,PO5
	b)	Explain the Decimal Arithmetic unit.	4	3	PO4,PO3,PO6
4.	a)	Discuss the CISC Characteristics.	4	4	PO3,PO2,PO6
	b)	List and explain the RISC Characteristics.	5	2	PO3,PO4,PO11
5	a)	Elaborate the Vector Processing and Array Processor.	5	2	PO4,PO3,PO12
	b)	Discuss the Characteristics of Multiprocessors.	5	3	PO2,PO4,PO7

- I. One Day Online workshop on Computer Organization on 28/12/2022 by Dr. Vijaya Bhaskar Reddy , Professor, LBRCE,NTR District
- II. One Day Guest Lecturer on Architecture on 2/01/2023 by Dr. J. Kiran Kumar , IBM, Hyderabad .
- III. **Case Tools:**

Thread Level Parallelism – SMT and CMP:

The objectives of this module are to discuss the drawbacks of ILP and the need for exploring other types of parallelism available in application programs and exploit them. We will discuss what is meant by thread level parallelism and discuss the concepts of Simultaneous Multi Threading and Chip Multi Processors.

So far, we have looked at various hardware and software techniques to exploit ILP. The ideal CPI that we can expect in a pipelined implementation is only 1. We looked at different techniques to avoid or minimize the stalls associated with the various hazards. The performance of a pipelined implementation can be improved by deepening the pipeline or widening the pipeline. Deepening the pipeline increases the number of in-flight instructions and decreases the gap between successive independent instructions. However, it increases the gap between dependent instructions. There is an optimal pipeline depth depending on the ILP in a program and it is a design issue. It may be tough to pipeline some structures and there may be an increase in the cost of bypassing. Increasing the width of the pipeline, as in the case of multiple issue processors also has its own problems and difficulties. It may be difficult to find more than a few, say, four independent instructions to issue and it may be difficult to fetch more than six instructions and there is also an increase in the number of ports per structure.

In order to reduce the stalls associated with fetch, we may have to employ better branch prediction methods with novel ways to index/update and avoid aliasing and also cascade branch predictors. The other option is to use a *trace cache*. Instead of limiting the instructions in a static cache block to spatial locality, a trace cache finds a dynamic sequence of instructions including taken branches to load into a cache block. The name comes from the cache blocks containing dynamic traces of the executed instructions as determined by the CPU rather than containing static sequences of instructions as determined by memory. Hence, the branch prediction is folded into cache, and must be validated along with the addresses to have a valid fetch. The Intel Netburst microarchitecture, which is the foundation of the Pentium 4 and its successors, uses a trace cache. The trace cache has a lot of shortcomings, but is very useful in handling the limitations of the fetch unit. In Intel processors, the trace cache stores the pre-decoded instructions.

In spite of all the hardware and software techniques employed to exploit ILP, there is a limit to how much we can exploit ILP. First of all, there is a limitation with the hardware that we use. The number of virtual registers that we actually have is limited, not infinite, to do the renaming process. The branch predictors and jump predictors that we use may not be perfect. Similarly, we may not be able to resolve memory address disambiguities always. In short, we do not have an idealistic processor, limited only by true data dependences and without any control, WAR and WAW hazards.

Doubling issue rates above today's 3-6 instructions per clock, say to 6 to 12 instructions, probably requires a processor to issue 3 or 4 data memory accesses per cycle, resolve 2 or 3 branches per

cycle, rename and access more than 20 registers per cycle, and fetch 12 to 24 instructions per cycle. The complexity of implementing these capabilities is likely to mean sacrifices in the maximum clock rate. For example, one of the widest issue processors is the Itanium 2, but it also has the slowest clock rate, despite the fact that it consumes the most power. Most techniques for increasing performance also increase the power consumption. Multiple issue processors techniques all are energy inefficient. Issuing multiple instructions incurs some overhead in logic that grows faster than the growth in issue rate. There is also a growing gap between the peak issue rates and sustained performance, which leads to increasing energy per unit of performance.

Exploiting other types of parallelism: The above discussion clearly shows that ILP can be quite limited or hard to exploit in some applications. More importantly, it may lead to increase in power consumption. Furthermore, there may be significant parallelism occurring naturally at a higher level in the application that cannot be exploited with the approaches used to exploit ILP. For example, an online transaction processing system has natural parallelism among the multiple queries and updates that are presented by requests. These queries and updates can be processed mostly in parallel, since they are largely independent of one another. This higher level parallelism is called *thread level parallelism* because it is logically structured as separate threads of execution. A *thread* is a separate process with its own instructions and data. A thread may represent a process that is part of a parallel program consisting of multiple processes, or it may represent an independent program on its own. Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute. Unlike instruction level parallelism, which exploits implicit parallel operations within a loop or straight-line code segment, thread level parallelism is explicitly represented by the use of multiple threads of execution that are inherently parallel.

Thread level parallelism is an important alternative to instruction level parallelism, primarily because it could be more cost-effective to exploit than instruction level parallelism. There are many important applications where thread level parallelism occurs naturally, as it does in many server applications. Similarly, a number of applications naturally exploit *data level parallelism*, where the same operation can be performed on multiple data. We shall discuss about exploiting data level parallelism in a later module.

Since ILP and TLP exploit two different types of parallel structure in a program, it is a natural option to combine these two types of parallelism. The datapath that has already been designed has a number of functional units remaining idle because of the insufficient ILP caused by stalls and dependences. This can be utilized to exploit TLP and thus make the functional units busy. There are predominantly two strategies for exploiting TLP along with ILP – *Multithreading* and its variants, viz., *Simultaneous Multi Threading (SMT)* and *Chip Multi Processors (CMP)*. In the case of SMT, multiple threads share the same large processor which reduces under-utilization and does efficient resource allocation. In the case of CMPs, each thread executes on its own mini processor, which results in a simple design and low interference between threads. We will discuss about both these approaches.

Multithreading: Multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion. In order to enable this, the processor duplicates the independent state of each thread – a separate copy of the register file, a separate PC, and a separate page table. The memory itself can be shared through the virtual memory mechanisms, which already support multiprogramming. In addition, the hardware must support the ability to change to a different thread relatively quickly; in particular, a thread switch should be much more efficient than a process switch, which typically requires hundreds to thousands of processor cycles.

There are two main approaches to multithreading – Fine grained and Coarse grained. ***Fine-grained multithreading*** switches between threads on each instruction, causing the execution of multiple threads to be interleaved. This interleaving is normally done in a round-robin fashion, skipping any threads that are stalled at that time. In order to support this, the CPU must be able to switch threads on every clock cycle. The main advantage of fine-grained multithreading is that it can hide the throughput losses that arise from both short and long stalls, since instructions from other threads can be executed when one thread stalls. But it slows down the execution of the individual threads, since a thread that is ready to execute without stalls will be delayed by instructions from other threads.

Coarse-grained multithreading switches threads only on costly stalls, such as level two cache misses. This allows some time for thread switching and is much less likely to slow the processor down, since instructions from other threads will only be issued, when a thread encounters a costly stall. Coarse-grained multithreading, however, is limited in its ability to overcome throughput losses, especially from shorter stalls. This limitation arises from the pipeline start-up costs of coarse-grain multithreading. Because a CPU with coarse-grained multithreading issues instructions from a single thread, when a stall occurs, the pipeline must be emptied or frozen and then fill in instructions from the new thread. Because of this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of high cost stalls, where pipeline refill is negligible compared to the stall time.

Simultaneous Multithreading: This is a variant on multithreading. When we only issue instructions from one thread, there may not be enough parallelism available and all the functional units may not be used. Instead, if we issue instructions from multiple threads in the same clock cycle, we will be able to better utilize the functional units. This is the concept of simultaneous multithreading. We try to use the resources of a multiple issue, dynamically scheduled superscalar to exploit TLP on top of ILP. The dynamically scheduled processor already has many HW mechanisms to support multithreading –

- a large set of virtual registers that can be used to hold the register sets of independent threads
- register renaming to provide unique register identifiers, so that instructions from multiple threads can be mixed in the data-path without confusing sources and destinations across threads and
- out-of-order completion that allows the threads to execute out of order, and get better utilization of the HW.

Thus, with register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to the dependences among them. The resolution of the dependences will be handled by the dynamic scheduling capability. We need to add a renaming table per thread and keep separate PCs. The independent commitment of each thread can be supported by logically keeping a separate reorder buffer for each thread. Figure 24.1 shows the difference between the various techniques.

In the superscalar approach without multithreading support, the number of instructions issued per clock cycle is dependent on the ILP available. Additionally, a major stall, such as an instruction cache miss, can leave the entire processor idle. In the fine-grained case, the interleaving of threads eliminates fully empty slots. Because only one thread issues instructions in a given clock cycle,

however, ILP limitations still lead to a significant number of idle slots within individual clock cycles. In the coarse-grained multithreaded superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor. Although this reduces the number of completely idle clock cycles, within each clock cycle, the ILP limitations still lead to idle cycles. Furthermore, in a coarse-grained multithreaded processor, since thread switching only occurs when there is a stall and the new thread has a start-up period, there are likely to be some fully idle cycles. In the SMT case, TLP and ILP are exploited simultaneously, with multiple threads using the issue slots in a single clock cycle. Ideally, the issue slot usage is limited by imbalances in the resource needs and resource availability over multiple threads. In practice, other factors—including how many active threads are considered, finite limitations on buffers, the ability to fetch enough instructions from multiple threads, and practical limitations of what instruction combinations can issue from one thread and from multiple threads—can also restrict how many slots are used.



13. Subject notes/PPTs/self study material

UNIT 1:

Digital Computers: Introduction, Block diagram of Digital Computer, Definition of Computer Organization, Computer Design and Computer Architecture.

Register Transfer Language and Micro operations: Register Transfer language, Register Transfer, Bus and memory transfers, Arithmetic Micro operations, logic micro operations, shift micro operations, Arithmetic logic shift unit.

Basic Computer Organization and Design: Instruction codes, Computer Registers Computer instructions, Timing and Control, Instruction cycle, Memory Reference Instructions, Input – Output and Interrupt.

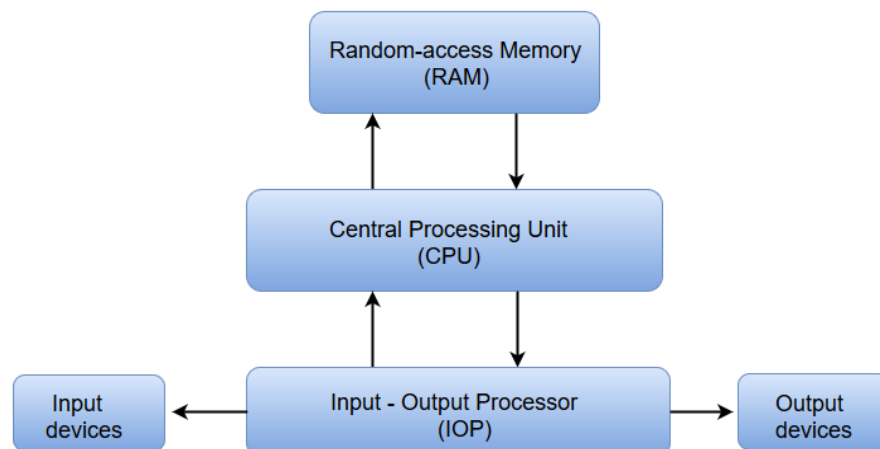
DIGITAL COMPUTERS

A Digital computer can be considered as a digital system that performs various computational tasks.

The first electronic digital computer was developed in the late 1940s and was used primarily for numerical computations. By convention, the digital computers use the binary number system, which has two digits: 0 and 1. A binary digit is called a bit. A computer system is subdivided into two functional entities: Hardware and Software.

The hardware consists of all the electronic components and electromechanical devices that comprise the physical entity of the device. The software of the computer consists of the instructions and data that the computer manipulates to perform various data-processing tasks.

Block diagram of a digital computer:



- The Central Processing Unit (CPU) contains an arithmetic and logic unit for manipulating data, a number of registers for storing data, and a control circuit for fetching and executing instructions.

- The memory unit of a digital computer contains storage for instructions and data.
- The Random Access Memory (RAM) for real-time processing of the data.
- The Input-Output devices for generating inputs from the user and displaying the final results to the user.
- The Input-Output devices connected to the computer include the keyboard, mouse, terminals, magnetic disk drives, and other communication devices.

BASIC COMPUTER ORGANIZATION:

Most of the computer systems found in automobiles and consumer appliances to personal computers and main frames have some basic organization. The basic computer organization has three main components:

- CPU
- Memory subsystem
- I/O subsystem.

The generic organization of these components is shown in the figure below.

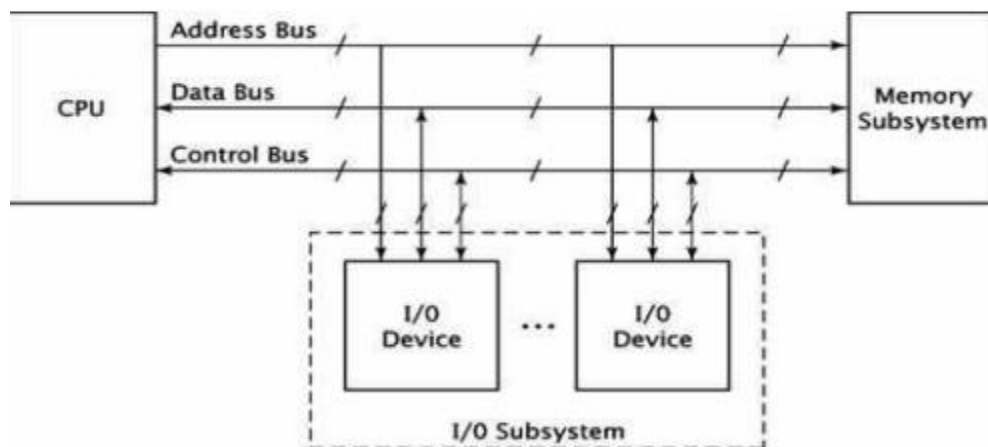


Fig 1.1 Generic computer Organization

Computer organization: Computer organization is the knowing, what the functional components of a computer are, how they work and how their performance is measured and optimized. Computer

Organization refers to the level of abstraction above the digital logic level, but below the operating system level.

Computer design and architecture:

Computer design is concerned with the determination of what hardware should be used and how the parts should be connected. This aspect of computer hardware is sometimes referred to as computer implementation. Computer architecture is concerned with the structure and behavior of the computer as seen by the user.

Register Transfer Language and Micro Operations:

Register Transfer language:

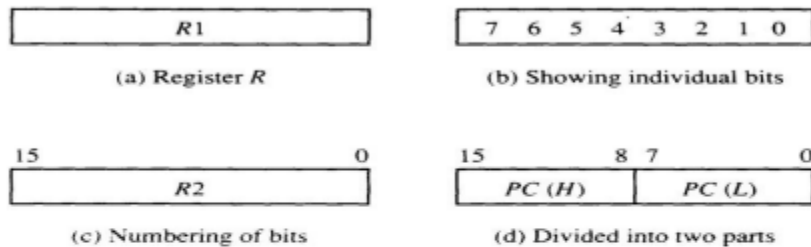
- ☐ Digital systems are composed of modules that are constructed from digital components, such as registers, decoders, arithmetic elements, and control logic
- ☐ The modules are interconnected with common data and control paths to form a digital computer system
- ☐ The operations executed on data stored in registers are called micro operations
- ☐ A micro operation is an elementary operation performed on the information stored in one or more registers
- ☐ Examples are shift, count, clear, and load
- ☐ Some of the digital components from before are registers that implement micro operations
- ☐ The internal hardware organization of a digital computer is best defined by specifying
 - o The set of registers it contains and their functions
 - o The sequence of micro operations performed on the binary information stored
 - o The control that initiates the sequence of micro operations
- ☐ Use symbols, rather than words, to specify the sequence of micro operations
- ☐ The symbolic notation used is called a register transfer language
- ☐ A programming language is a procedure for writing symbols to specify a given computational process

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

Register Transfer

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

Figure 4-1 Block diagram of register.

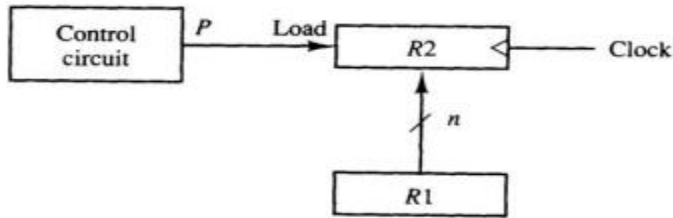


- Designate information transfer from one register to another by $R2 \leftarrow R1$
- This statement implies that the hardware is available
 - The outputs of the source must have a path to the inputs of the destination
 - The destination register has a parallel load capability
- If the transfer is to occur only under a predetermined control condition, designate it by
 - If $(P = 1)$ then $(R2 \leftarrow R1)$
 or,
 - $P: R2 \leftarrow R1,$

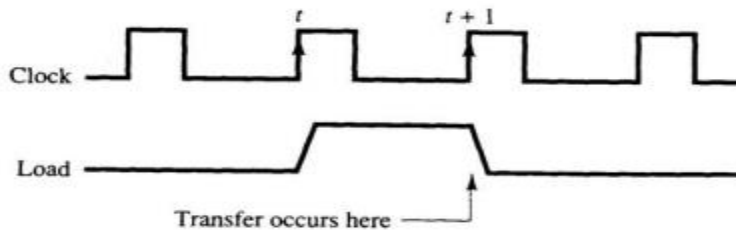
where P is a control function that can be either 0 or 1

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

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



(a) Block diagram



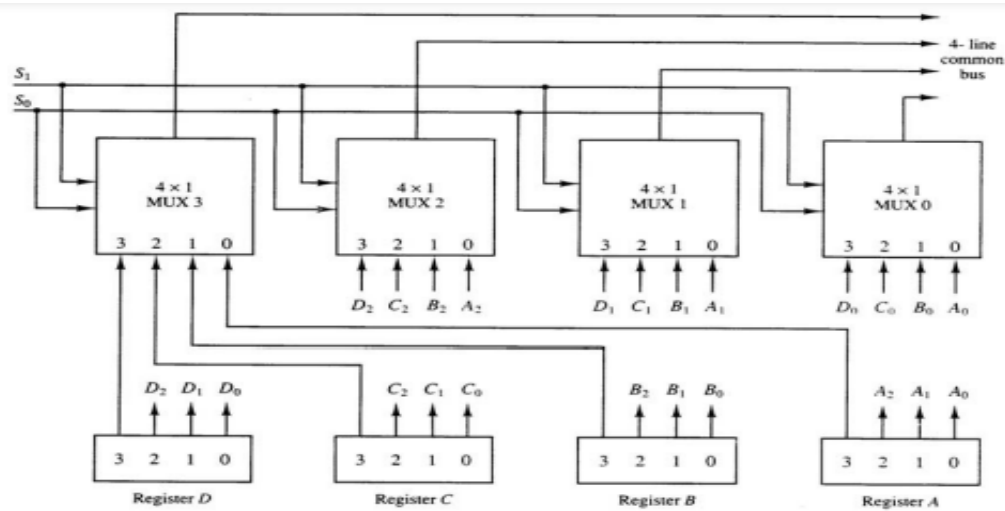
- It is assumed that all transfers occur during a clock edge transition
- All microoperations written on a single line are to be executed at the same time T : $R2 \leftarrow R1, R1 \leftarrow R2$

TABLE 4-1 Basic Symbols for Register Transfers

Symbol	Description	Examples
Letters (and numerals)	Denotes a register	$MAR, R2$
Parentheses ()	Denotes a part of a register	$R2(0-7), R2(L)$
Arrow \leftarrow	Denotes transfer of information	$R2 \leftarrow R1$
Comma ,	Separates two microoperations	$R2 \leftarrow R1, R1 \leftarrow R2$

Bus and Memory Transfers

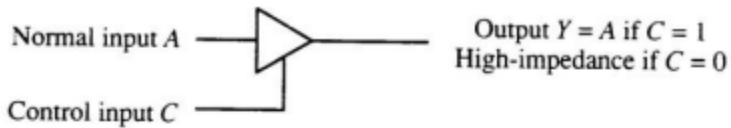
- Rather than connecting wires between all registers, a common bus is used
- A bus structure consists of a set of common lines, one for each bit of a register
- Control signals determine which register is selected by the bus during each transfer



- In general, a bus system will multiplex k registers of n bits each to produce an n -line common bus
- This requires n multiplexers – one for each bit
- The size of each multiplexer must be $k \times 1$
- The number of select lines required is $\log k$
- To transfer information from the bus to a register, the bus lines are connected to the inputs of all destination registers and the corresponding load control line must be activated
- Rather than listing each step as
 BUS \square C, R1 \square BUS,
 use R1 \square C, since the bus is implied



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



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

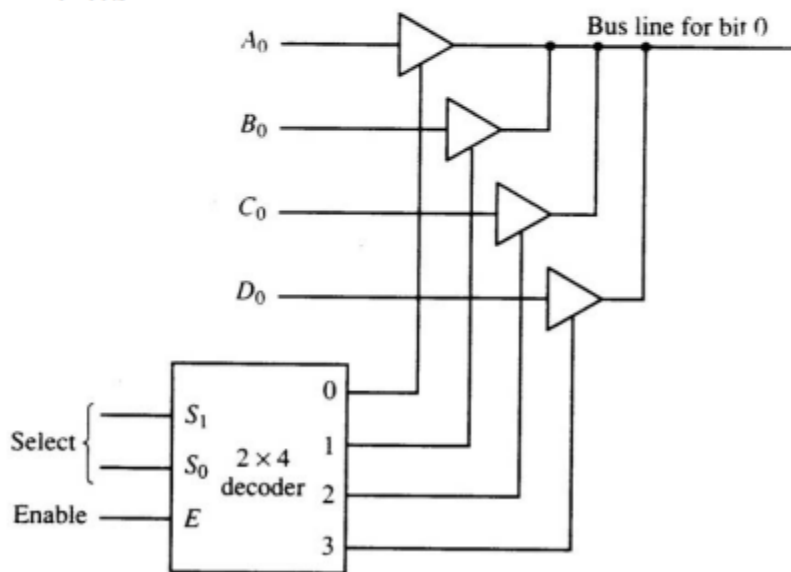


Figure 4-5 Bus line with three state-buffers.

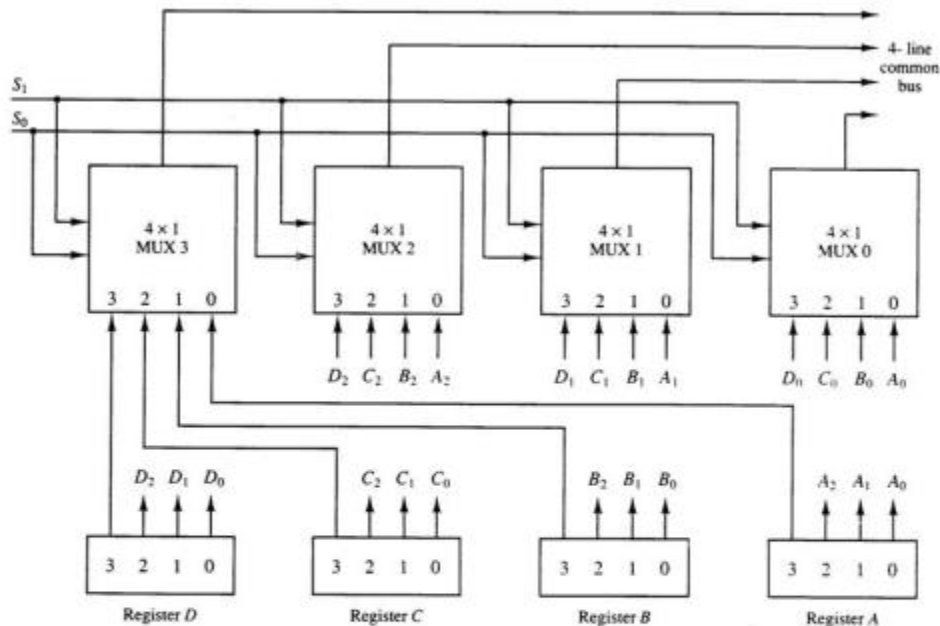
- Designate the address register by AR and the data register by DR
- The read operation can be stated as: Read: DR \leftarrow M[AR]
- The write operation can be stated as:
Write: M[AR] \leftarrow R1

Arithmetic Microoperations

- There are four categories of the most common microoperations:
 - Register transfer: transfer binary information from one register to another
 - Arithmetic: perform arithmetic operations on numeric data stored in registers
 - Logic: perform bit manipulation operations on non-numeric data stored in registers
 - Shift: perform shift operations on data stored in registers
- The basic arithmetic microoperations are addition, subtraction, increment, decrement, and shift
- Example of addition: R3 \leftarrow R1 + R2
- Subtraction is most often implemented through complementation and addition
- Example of subtraction: R3 \leftarrow R1 + ~~R2~~ + 1 (strikethrough denotes bar on top – 1's complement of R2)
- Adding 1 to the 1's complement produces the 2's complement
- Adding the contents of R1 to the 2's complement of R2 is equivalent to subtracting



Figure 4-3 Bus system for four registers.



- Multiply and divide are not included as microoperations
- A microoperation is one that can be executed by one clock pulse
- Multiply (divide) is implemented by a sequence of add and shift microoperations (subtract and shift)
- To implement the add microoperation with hardware, we need the registers that hold the data and the digital component that performs the addition
- A full-adder adds two bits and a previous carry
- An n-bit binary adder requires n full-adders

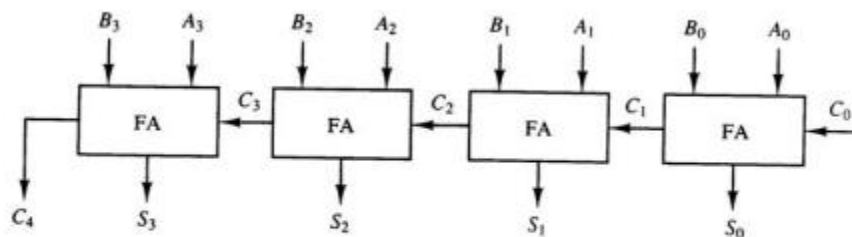


Figure 4-6 4-bit binary adder.

- The subtraction $A-B$ can be carried out by the following steps
 - Take the 1's complement of B (invert each bit)



- Get the 2's complement by adding 1
- Add the result to A
- The addition and subtraction operations can be combined into one common circuit by including an XOR gate with each full-adder

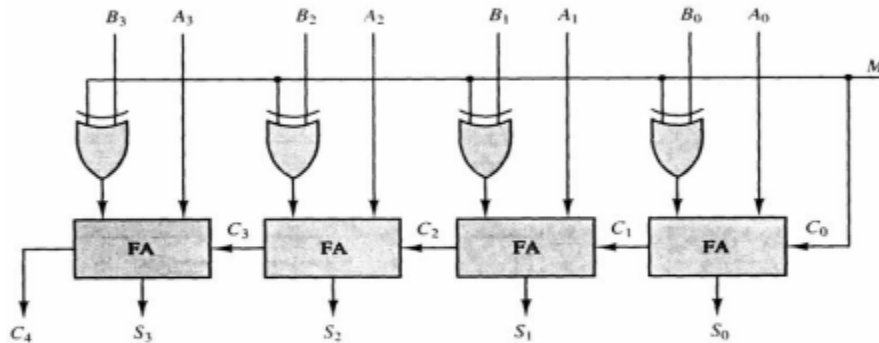


Figure 4-7 4-bit adder-subtractor.

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

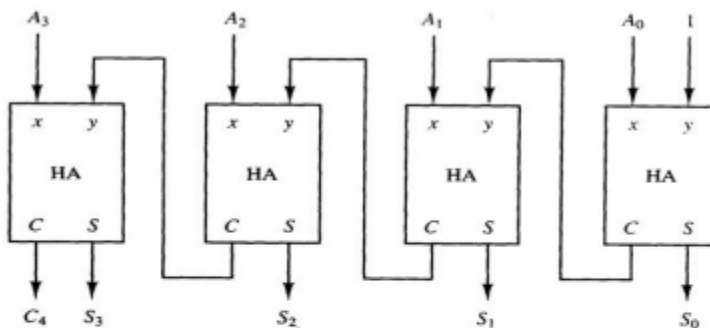


Figure 4-8 4-bit binary incrementer.

- Each of the arithmetic microoperations can be implemented in one composite arithmetic circuit
- The basic component is the parallel adder
- Multiplexers are used to choose between the different operations
- The output of the binary adder is calculated from the following sum: $D = A + Y + C_{in}$

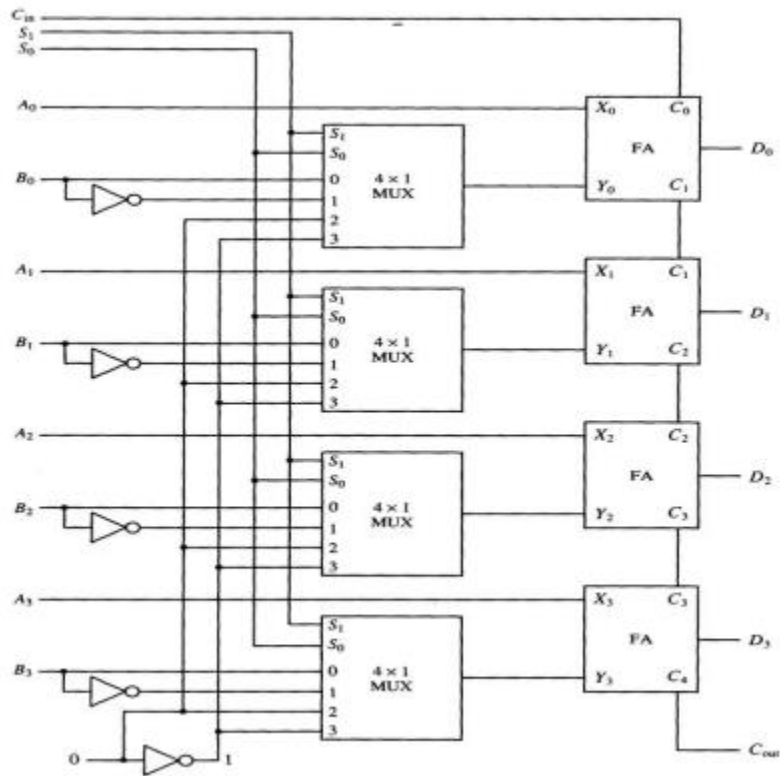


Figure 4-9 4-bit arithmetic circuit.



TABLE 4-4 Arithmetic Circuit Function Table

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

Logic Microoperations

- Logic operations specify binary operations for strings of bits stored in registers and treat each bit separately
- Example: the XOR of R1 and R2 is symbolized by

$$P: R1 \square R1 \oplus R2$$

- Example: R1 = 1010 and R2 = 1100

$$\begin{array}{ll} 1010 & \text{Content of R1} \\ \underline{1100} & \text{Content of R2} \end{array}$$

Logic Micro operations

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

□ Example: the XOR of R1 and R2 is symbolized by

$$P: R1 \square R1 \oplus R2$$

□ Example: R1 = 1010 and R2 = 1100

1010 Content of R1

1100 Content of R2

0110 Content of R1 after $P = 1$

- Symbols used for logical microoperations:
 - OR: \vee
 - AND: \wedge
 - XOR: \oplus
- The + sign has two different meanings: logical OR and summation
- When + is in a microoperation, then summation
- When + is in a control function, then OR
- Example:

$P + Q: R1 \leftarrow R2 + R3, R4 \leftarrow R5 \wedge R6$
- There are 16 different logic operations that can be performed with two binary variables

TABLE 4-5 Truth Tables for 16 Functions of Two Variables

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

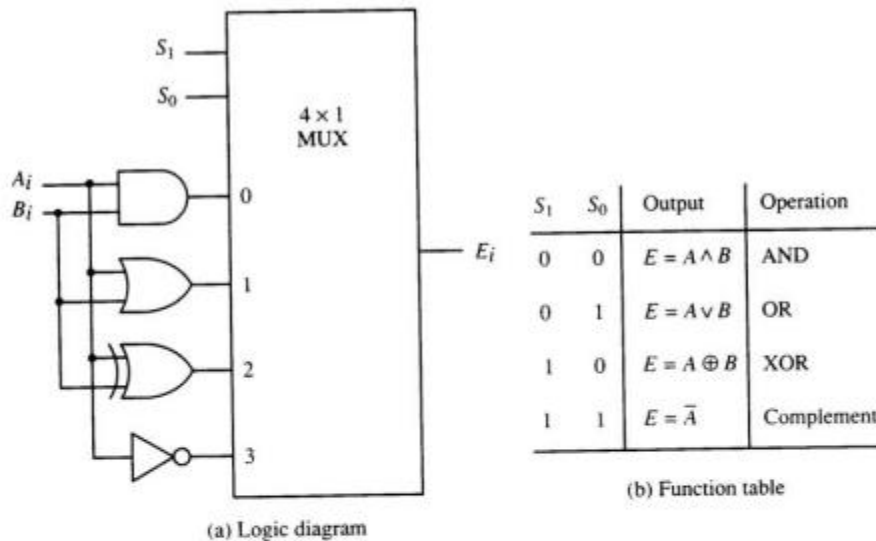
TABLE 4-6 Sixteen Logic Microoperations

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

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

- All 16 microoperations can be derived from using four logic gates

Figure 4-10 One stage of logic circuit.



- Logic microoperations can be used to change bit values, delete a group of bits, or insert new bit values into a register
- The selective-set operation sets to 1 the bits in A where there are corresponding 1's in B

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

$$A \leftarrow A \vee B$$

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

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

$$A \leftarrow A \oplus B$$

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

1010 A before
1100 B
 (logic operand)

operand)
 0010 A
 after
 $A \square A \square B$

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

1010 A before
1100 B
 (logic
 operand)
 1000 A
 after
 $A \square A \square B$

- The insert operation inserts a new value into a group of bits
- This is done by first masking the bits to be replaced and then Oring them with the bits to be inserted

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



```

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

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

```

1010  A
1010  B
-----
0000  A □ A ⊕ B
    
```

Shift Microoperations

- Shift microoperations are used for serial transfer of data
- They are also used in conjunction with arithmetic, logic, and other data-processing operations
- There are three types of shifts: logical, circular, and arithmetic
- A logical shift is one that transfers 0 through the serial input
- The symbols shl and shr are for logical shift-left and shift-right by one position $R1 \square \text{shl}R1$
- The circular shift (aka rotate) circulates the bits of the register around the two ends without loss of information
- The symbols cil and cir are for circular shift left and right



TABLE 4-7 Shift Microoperations

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

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

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

**Figure 4-11** Arithmetic shift right.

- A bi-directional shift unit with parallel load could be used to implement this
- Two clock pulses are necessary with this configuration: one to load the value and another to shift



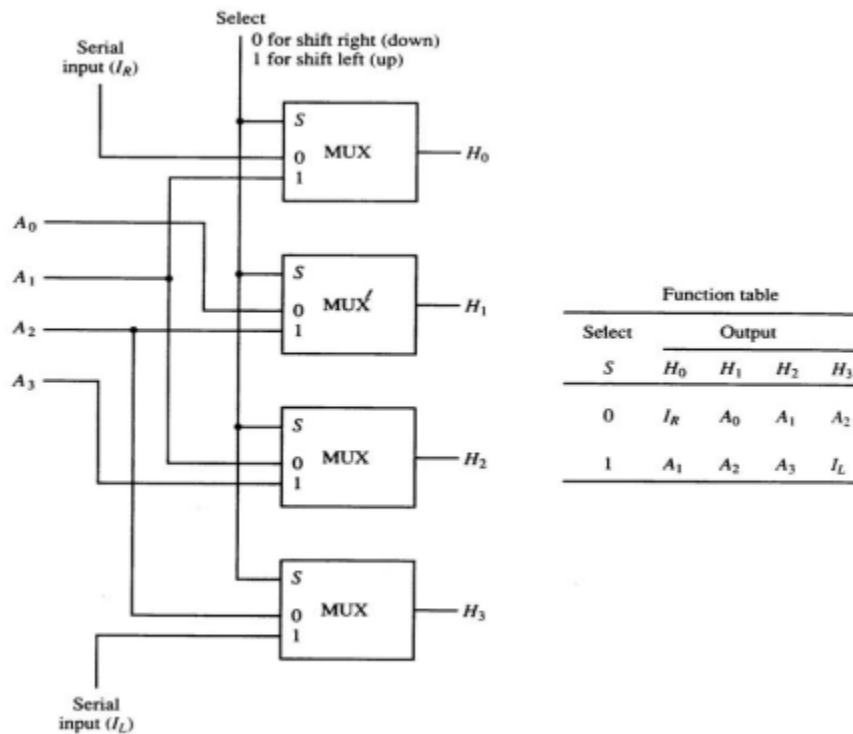


Figure 4-12 4-bit combinational circuit shifter.

Arithmetic Logic Shift Unit

- ☐ The arithmetic logic unit (ALU) is a common operational unit connected to a number of storage registers
- ☐ To perform a microoperation, the contents of specified registers are placed in the inputs of the ALU
- ☐ The ALU performs an operation and the result is then transferred to a destination register
- ☐ The ALU is a combinational circuit so that the entire register transfer operation from the source registers through the ALU and into the destination register can be performed during one clock pulse period.

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

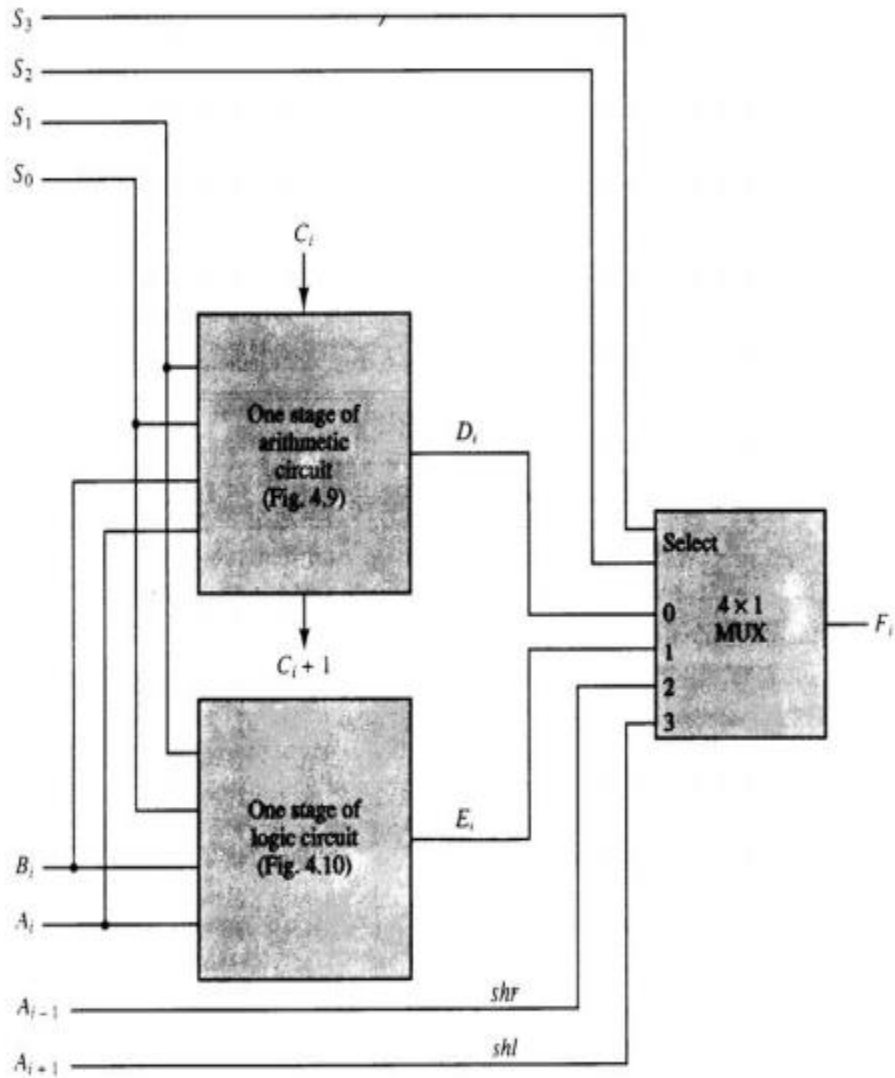


TABLE 4-8 Function Table for Arithmetic Logic Shift Unit

Operation select					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	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + \overline{B}$	Subtract with borrow
0	0	1	0	1	$F = A + \overline{B} + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	\times	$F = A \wedge B$	AND
0	1	0	1	\times	$F = A \vee B$	OR
0	1	1	0	\times	$F = A \oplus B$	XOR
0	1	1	1	\times	$F = \overline{A}$	Complement A
1	0	\times	\times	\times	$F = \text{shr } A$	Shift right A into F
1	1	\times	\times	\times	$F = \text{shl } A$	Shift left A into F

Instruction Formats:

A computer will usually have a variety of instruction code formats. It is the function of the control unit within the CPU to interpret each instruction code and provide the necessary control functions needed to process the instruction.

The format of an instruction is usually depicted in a rectangular box symbolizing the bits of the instruction as they appear in memory words or in a control register. The bits of the instruction are divided into groups called fields. The most common fields found in instruction formats are:

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

Other special fields are sometimes employed under certain circumstances, as for example a field that gives the number of shifts in a shift-type instruction.

The operation code field of an instruction is a group of bits that define various processor operations, such as add, subtract, complement, and shift.

The bits that define the mode field of an instruction code specify a variety of alternatives for choosing the operands from the given address.

Operations specified by computer instructions are executed on some data stored in memory or processor registers. Operands residing in processor registers are specified with a register address. A register address is a binary number of k bits that defines one of 2^k registers in the CPU. Thus a CPU with 16 processor registers R0 through R15 will have a register address field of four bits. The binary number 0101, for example, will designate register R5. Computers may have instructions of several different lengths containing varying number of addresses. The number of address fields in the instruction format of a computer depends on the internal organization of its registers. Most computers fall into one of three types of CPU organizations: 1 Single accumulator organization. 2 General register organization. 3 Stack organization. All operations are performed with an implied accumulator register. The instruction format in this type of computer uses one address field. For example, the instruction that specifies an arithmetic addition is defined by an assembly language instruction as ADD. Where X is the address of the operand. The ADD instruction in this case results in the operation $AC \leftarrow AC + M[X]$. AC is the accumulator register and $M[X]$ symbolizes the memory word located at address X. An example of a general register type of organization was presented in Fig. 7.1. The instruction format in this type of computer needs three register address fields. Thus the instruction for an arithmetic addition may be written in an assembly language as ADD R1, R2, R3 To denote the operation $R1 \leftarrow R2 + R3$. The number of address fields in the instruction can be reduced from three to two if the destination register is the same as one of the source registers. Thus the instruction ADD R1, R2 Would denote the operation $R1 \leftarrow R1 + R2$. Only register addresses for R1 and R2 need be specified in this instruction. Computers with multiple processor registers use the move instruction with a mnemonic MOV to symbolize a transfer instruction. Thus the instruction MOV R1, R2 Denotes the transfer $R1 \leftarrow R2$ (or $R2 \leftarrow R1$, depending on the particular computer). Thus transfer-type instructions need two address fields to specify the source and the destination.

General register-type computers employ two or three address fields in their instruction format. Each address field may specify a processor register or a memory word. An instruction symbolized by ADD R1, X Would specify the operation $R1 \leftarrow R + M[X]$. It has two address fields, one for register R1 and the other for the memory address X. The stack-organized CPU was presented in Fig. 8-4. Computers with stack organization would have PUSH and POP instructions which require an address field. Thus the instruction PUSH X Will push the word at address X to the top of the stack. The stack pointer is updated automatically. Operation-type instructions do not need an address field in stack-organized computers. This is because the operation is performed on the two items that are on top of the stack. The instruction ADD in a stack computer consists of an operation code only with no address field. This operation has the effect of popping the two top numbers from the stack, adding the numbers, and pushing the sum into the stack. There is no need to specify operands with an address field since all operands are implied to be in the stack. To illustrate the influence of the number of addresses on computer programs, we will evaluate the arithmetic statement $X = (A + B) * (C + D)$. Using zero, one, two, or three address instruction. We will use the symbols ADD, SUB,

MUL, and DIV for the four arithmetic operations; MOV for the transfer-type operation; and LOAD and STORE for transfers to and from memory and AC register. We will assume that the operands are in memory addresses A, B, C, and D, and the result must be stored in memory at address X. Three-Address Instructions Computers with three-address instruction formats can use each address field to specify either a processor register or a memory operand. The program in assembly language that evaluates $X = (A + B) * (C + D)$ is shown below, together with comments that explain the register transfer.

operation of each instruction.

ADD R1, A, B R1 \leftarrow

M [A] + M [B]

ADD R2, C, D R2 \leftarrow

M [C] + M [D]

MUL X, R1, R2 M [X]

\leftarrow R1 *R2

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

The advantage of the three-address format is that it results in short programs when evaluating arithmetic expressions. The disadvantage is that the binary-coded instructions require too many bits to specify three addresses. An example of a commercial computer that uses three-address instructions is the Cyber 170. The instruction formats in the Cyber computer are restricted to either three register address fields or two register address fields and one memory address field

Two-Address Instructions

Two address instructions are the most common in commercial computers. Here again each address field can specify either a processor register or a memory word. The program to evaluate $X = (A + B) * (C + D)$ is as

follows:

MOV R1, A R1 \leftarrow M [A]

ADD R1, B R1 \leftarrow R1 + M [B]

MOV R2, C R2 \leftarrow M [C]

ADD R2, D $R2 \leftarrow R2 + M[D]$

MUL R1, R2 $R1 \leftarrow R1 * R2$

MOV X, R1 $M[X] \leftarrow R1$

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

One-Address Instructions

One-address instructions use an implied accumulator (AC) register for all data manipulation. For multiplication and division there is a need for a second register. However, here we will neglect the second and assume that the AC contains the result of all operations. The program to evaluate $X =$

$(A + B) * (C + D)$ is

LOAD A $AC \leftarrow M[A]$

ADD B $AC \leftarrow AC + M[B]$

STORE T $M[T] \leftarrow AC$

LOAD C $AC \leftarrow M[C]$

ADD D $AC \leftarrow AC + M[D]$

MUL T $AC \leftarrow AC * M[T]$

STORE X $M[X] \leftarrow AC$

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

Zero-Address Instructions

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

written for a stack organized computer. (TOS stands for top of stack)

PUSH A $TOS \leftarrow A$

PUSH B TOS \leftarrow B

ADD

PUSH C

TOS \leftarrow (A + B)

TOS \leftarrow C

PUSH D TOS \leftarrow D

ADD TOS \leftarrow (C + D)

MUL

POP X

TOS \leftarrow (C + D) * (A + B)

M[X] \leftarrow TOS

To evaluate arithmetic expressions in a stack computer, it is necessary

to convert the expression into reverse Polish notation. The name “zeroaddress” is given to this type of computer because of the absence of an address field in the computational instructions.

Instruction Codes

A set of instructions that specify the operations, operands, and the sequence by which processing has to occur. An instruction code is a group of bits that tells the computer to perform a specific operation part.

Format of Instruction

The format of an instruction is depicted in a rectangular box symbolizing the bits of an instruction. Basic fields of an instruction format are given below:

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

Computers may have instructions of different lengths containing varying number of addresses. The number of address field in the instruction format depends upon the internal organization of its registers.

Addressing Modes

To understand the various addressing modes to be presented in this section, it is imperative that we understand the basic operation cycle of the computer.

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

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

There is one register in the computer called the program counter or PC that keeps track of the instructions in the program stored in memory. PC holds the address of the instruction to be executed next and is incremented each time an instruction is fetched from memory. The decoding done in step 2 determines the operation to be performed, the addressing mode of the instruction and the location of the operands. The computer then executes the instruction and returns to step 1 to fetch the next instruction in sequence. In some computers the addressing mode of the instruction is specified with a distinct binary code, just like the operation code is specified. Other computers use a single binary code that designates both the operation and the mode of the instruction. Instructions may be defined with a variety of addressing modes, and sometimes, two or more addressing modes are combined in one instruction.

1. The operation code specifies the operation to be performed. The mode field is used to locate the operands needed for the operation. There may or may not be an address field in the instruction. If there is an address field, it may designate a memory address or a processor register. Moreover, as discussed in the preceding section, the instruction may have more than one address field, and each address field may be associated with its own particular addressing mode.

Although most addressing modes modify the address field of the instruction, there are two modes that need no address field at all. These are the implied and immediate modes.

1 Implied Mode: In this mode the operands are specified implicitly in the definition of the instruction. For example, the instruction “complement accumulator” is an implied-mode instruction because the operand in the accumulator register is implied in the definition of the instruction. In fact, all register reference instructions that use an accumulator are implied-mode instructions.

Op code Mode Address

Figure 1: Instruction format with mode field

Zero-address instructions in a stack-organized computer are implied mode instructions since the operands are implied to be on top of the stack.

2 Immediate Mode: In this mode the operand is specified in the instruction itself. In other words, an immediate- mode instruction has an operand field rather than an address field. The operand field contains the actual operand to be used in conjunction with the operation specified in the instruction. Immediate-mode instructions are useful for initializing registers to a constant value.

It was mentioned previously that the address field of an instruction may specify either a memory word or a processor register. When the address field specifies a processor register, the instruction is said to be in the register mode.

3 Register Mode: In this mode the operands are in registers that reside within the CPU. The particular register is selected from a register field in the instruction. A kbit field can specify any one of 2^k registers.

4 Register Indirect Mode: In this mode the instruction specifies a register in the CPU whose contents give the address of the operand in memory. In other words, the selected register contains the address of the operand rather than the operand itself. Before using a register indirect mode instruction, the programmer must ensure that the memory address for the operand is placed in the processor register with a previous instruction. A reference to the register is then equivalent to specifying a memory address. The advantage of a register indirect mode instruction is that the address field of the instruction uses fewer bits to select a register than would have been required to specify a memory address directly.

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

However, because it is such a common requirement, some computers incorporate a special mode that automatically increments or decrements the content of the register after data access.

The address field of an instruction is used by the control unit in the CPU to obtain the operand from memory. Sometimes the value given in the address field is the address of the operand, but sometimes it is just an address from which the address of the operand is calculated. To differentiate among the various addressing modes it is necessary to distinguish between the address part of the instruction

and the effective address used by the control when executing the instruction. The effective address is defined to be the memory address obtained from the computation dictated by the given addressing mode. The effective address is the address of the operand in a computational-type instruction. It is the address where control branches in response to a branch-type instruction. We have already defined two addressing modes in previous chapter.

6 Direct Address Mode: In this mode the effective address is equal to the address part of the instruction. The operand resides in memory and its address is given directly by the address field of the instruction. In a branch-type instruction the address field specifies the actual branch address.

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

8 Relative Address Mode: In this mode the content of the program counter is added to the address part of the instruction in order to obtain the effective address. The address part of the instruction is usually a signed number (in 2's complement representation) which can be either positive or negative. When this number is added to the content of the program counter, the result produces an effective address whose position in memory is relative to the address of the next instruction. To clarify with an example, assume that the program counter contains the number 825 and the address part of the instruction contains the number 24. The instruction at location 825 is read from memory during the fetch phase and the program counter is then incremented by one to $826 + 24 = 850$. This is 24 memory locations forward from the address of the next instruction. Relative addressing is often used with branch-type instructions when the branch address is in the area surrounding the instruction word itself. It results in a shorter address field in the instruction format since the relative address can be specified with a smaller number of bits compared to the number of bits required to designate the entire memory address.

9 Indexed Addressing Mode: In this mode the content of an index register is added to the address part of the instruction to obtain the effective address. The index register is a special CPU register that contains an index value. The address field of the instruction defines the beginning address of a data array in memory. Each operand in the array is stored in memory relative to the beginning address. The distance between the beginning address and the address of the operand is the index value stored in the index register. Any operand in the array can be accessed with the same instruction provided that the index register contains the correct index value. The index register can be incremented to facilitate access to consecutive operands. Note that if an index-type instruction does not include an address field in its format, the instruction converts to the register indirect mode of operation. Some computers dedicate one CPU register to function solely as an index register. This register is involved implicitly when the index-mode instruction is used. In computers with many processor registers, any

one of the CPU registers can contain the index number. In such a case the register must be specified explicitly in a register field within the instruction format.

10 Base Register Addressing Mode: In this mode the content of a base register is added to the address part of the instruction to obtain the effective address. This is similar to the indexed addressing mode except that the register is now called a base register instead of an index register. The difference between the two modes is in the way they are used rather than in the way that they are computed. An index register is assumed to hold an index number that is relative to the address part of the instruction. A base register is assumed to hold a base address and the address field of the instruction gives a displacement relative to this base address. The base register addressing mode is used in computers to facilitate the relocation of programs in memory. When programs and data are moved from one segment of memory to another, as required in multiprogramming systems, the address values of the base register requires updating to reflect the beginning of a new memory segment.

Numerical Example

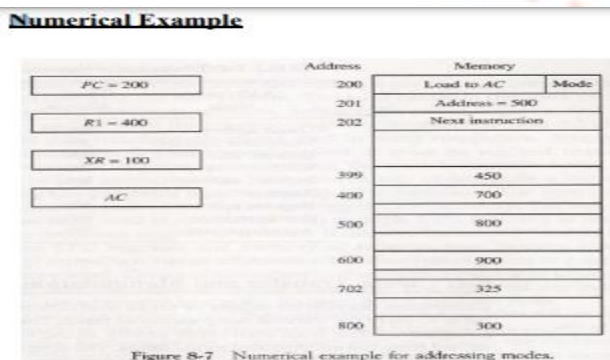


TABLE 8-4 Tabular List of Numerical Example

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

Computer Registers

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

☐ Data Register(DR) : hold the operand(Data) read from memory

☐ Accumulator Register(AC) : general purpose processing register

☐ Instruction Register(IR) : hold the instruction read from memory

☐ Temporary Register(TR) : hold a temporary data during processing

☐ Address Register(AR) : hold a memory address, 12 bit width

☐ Program Counter(PC) :

»hold the address of the next instruction to be read from memory after the current instruction is executed

»Instruction words are read and executed in sequence unless a branch instruction is encountered

»A branch instruction calls for a transfer to a nonconsecutive instruction in the program

»The address part of a branch instruction is transferred to PC to become the address of the next instruction

Input Register(INPR) : receive an 8-bit character from an input device

☐ Output Register(OUTR) : hold an 8-bit character for an output device

The following registers are used in Mano's example computer.

Register Number Register Register

symbol of bits name Function-----

DR 16 Data register Holds memory operands

AR 12 Address register Holds address for memory

AC 16 Accumulator Processor register

IR 16 Instruction register Holds instruction code

PC 12 Program counter Holds address of instruction

TR 16 Temporary register Holds temporary data

INPR 8 Input register Holds input character

OUTR 8 Output register Holds output character

Computer Instructions:

The basic computer has 16 bit instruction register (IR) which can denote either memory reference or register reference or input-output instruction.

1. Memory Reference – These instructions refer to memory address as an operand. The other operand is always accumulator. Specifies 12 bit address, 3 bit opcode (other than 111) and 1 bit addressing mode for direct and indirect addressing.

Example –

IR register contains = 0001XXXXXXXXXXXX, i.e. ADD after fetching and decoding of instruction we find out that it is a memory reference instruction for ADD operation.

Hence, $DR \leftarrow M[AR]$

$AC \leftarrow AC + DR$, $SC \leftarrow 0$

2. Register Reference – These instructions perform operations on registers rather than memory addresses. The IR(14-12) is 111 (differentiates it from memory reference) and IR(15) is 0 (differentiates it from input/output instructions). The rest 12 bits specify register operation.

Example –

IR register contains = 0111001000000000, i.e. CMA after fetch and decode cycle we find out that it is a register reference instruction for complement accumulator.

Hence, $AC \leftarrow \sim AC$

3. Input/Output – These instructions are for communication between computer and outside environment. The IR(14-12) is 111 (differentiates it from memory reference) and IR(15) is 1 (differentiates it from register reference instructions). The rest 12 bits specify I/O operation.

Example –

IR register contains = 1111100000000000, i.e. INP after fetch and decode cycle we find out that it is an input/output instruction for inputting character. Hence, INPUT character from peripheral device.

Timing and Control

All sequential circuits in the Basic Computer CPU are driven by a master clock, with the exception of the INPR register. At each clock pulse, the control unit sends control signals to control inputs of the bus, the registers, and the ALU.

Control unit design and implementation can be done by two general methods:

- ☐ A hardwired control unit is designed from scratch using traditional digital logic design techniques to produce a minimal, optimized circuit. In other words, the control unit is like an ASIC (application-specific integrated circuit).
- ☐ A microprogrammed control unit is built from some sort of ROM. The desired control signals are simply stored in the ROM, and retrieved in sequence to drive the microoperations needed by a particular instruction.

Instruction Cycle

The CPU performs a sequence of microoperations for each instruction. The sequence for each instruction of the Basic Computer can be refined into 4 abstract phases:

1. Fetch instruction
2. Decode
3. Fetch operand
4. Execute

Program execution can be represented as a top-down design:

1. Program execution

a. Instruction 1

i. Fetch instruction

ii. Decode

iii. Fetch operand

iv. Execute

b. Instruction 2

i. Fetch instruction

ii. Decode

iii. Fetch operand

iv. Execute

c. Instruction 3 ...

Program execution begins with:

$PC \leftarrow \text{address of first instruction}, SC \leftarrow 0$

After this, the SC is incremented at each clock cycle until an instruction is completed, and then it is cleared to begin the next instruction. This process repeats until a HLT instruction is executed, or until the power is shut off.

Instruction Fetch and Decode

The instruction fetch and decode phases are the same for all instructions, so the control functions and microoperations will be independent of the instruction code.

Everything that happens in this phase is driven entirely by timing variables T0, T1 and T2. Hence, all control inputs in the CPU during fetch and decode are functions of these three variables alone.

T0: $AR \leftarrow PC$

T1: $IR \leftarrow M[AR], PC \leftarrow PC + 1$

T2: $D0-7 \leftarrow \text{decoded } IR(12-14), AR \leftarrow IR(0-11), I \leftarrow IR(15)$

For every timing cycle, we assume $SC \leftarrow SC + 1$ unless it is stated that $SC \leftarrow 0$.

UNIT 2:

Microprogrammed Control: Control memory, Address sequencing, micro program example, design of control unit.

Central Processing Unit: General Register Organization, Instruction Formats, Addressing modes, Data Transfer and Manipulation, Program Control.

Micro Programmed Control:

Control Memory

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

- ☐ The control data register holds the microinstruction read from memory
- ☐ The microinstruction contains a control word that specifies one or more microoperations for the data processor
- ☐ The location for the next microinstruction may, or may not be the next in sequence
- ☐ Some bits of the present microinstruction control the generation of the address of the next microinstruction
- ☐ The next address may also be a function of external input conditions
- ☐ While the microoperations are being executed, the next address is computed in the next address generator circuit (sequencer) and then transferred into the CAR to read the next microinstructions
- ☐ Typical functions of a sequencer are:
 - o incrementing the CAR by one
 - o loading into the CAR and address from control memory
 - o transferring an external address
 - o loading an initial address to start the control operations
- ☐ A clock is applied to the CAR and the control word and next-address information are taken directly from the control memory
- ☐ The address value is the input for the ROM and the control word is the output
- ☐ No read signal is required for the ROM as in a RAM
- ☐ The main advantage of the microprogrammed control is that once the hardware configuration is established, there should be no need for h/w or wiring changes
- ☐ To establish a different control sequence, specify a different set of microinstructions for control memory

Address Sequencing

- ☐ Microinstructions are stored in control memory in groups, with each group specifying a routine
- ☐ Each computer instruction has its own microprogram routine to generate the microoperations
- ☐ The hardware that controls the address sequencing of the control memory must be capable of sequencing the microinstructions within a routine and be able to branch from one routine to another
- ☐ Steps the control must undergo during the execution of a single computer instruction:
 - o Load an initial address into the CAR when power is turned on in the computer. This address is usually the address of the first microinstruction that activates the instruction fetch routine – IR holds instruction

o The control memory then goes through the routine to determine the effective address of the operand – AR holds operand address o The next step is to generate the microoperations that

execute the instruction by considering the opcode and applying a mapping o After execution, control must return to the fetch routine by executing an unconditional branch

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

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

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

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

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

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

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

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

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

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

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

☐ This provides for each computer instruction a microprogram routine with a capacity of four microinstructions

☐ Subroutines are programs that are used by other routines to accomplish a particular task and can be called from any point within the main body of the microprogram

☐ Frequently many microprograms contain identical section of code

- ☐ Microinstructions can be saved by employing subroutines that use common sections of microcode
- ☐ Microprograms that use subroutines must have a provisions for storing the return address during a subroutine call and restoring the address during a subroutine return
- ☐ A subroutine register is used as the source and destination for the addresses

Microprogram Example

•In the block diagram four registers and ALU are associated with the processor unit. –DR, AR, PC, AC and–ALU•DR can receive information from AC, PC or memory (selected byMUX)•AR can receive information from PC or DR (selected by MUX)•PC can receive information only from AR.

➤The process of code generation for the control memory is called microprogramming. ➤Transfer of information among registers in the processor is through MUXs rather than a bus.

Design of Control Unit

The Control Unit is classified into two major categories:

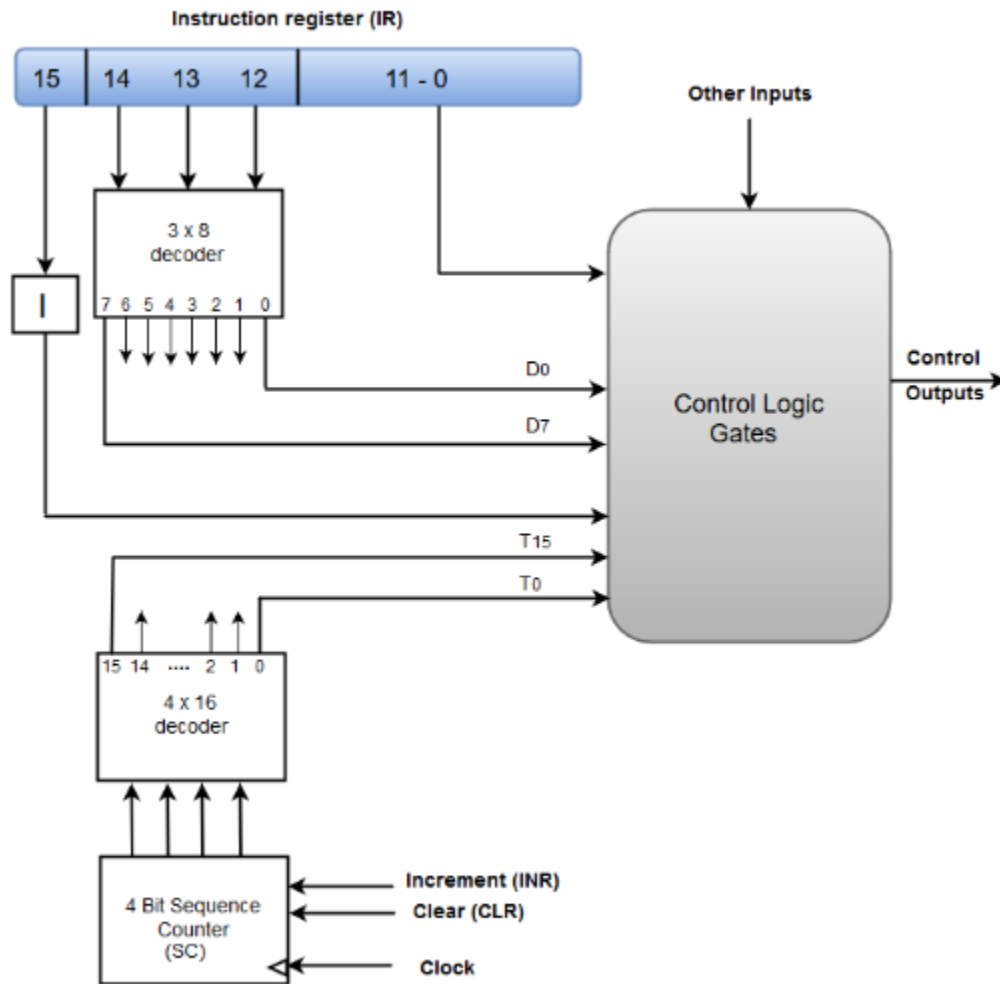
Hardwired Control

Microprogrammed Control

Hardwired Control

The Hardwired Control organization involves the control logic to be implemented with gates, flip-flops, decoders, and other digital circuits.

The following image shows the block diagram of a Hardwired Control organization.



- A Hard-wired Control consists of two decoders, a sequence counter, and a number of logic gates.
- An instruction fetched from the memory unit is placed in the instruction register (IR).
- The component of an instruction register includes; I bit, the operation code, and bits 0 through 11.
- The operation code in bits 12 through 14 are coded with a 3 x 8 decoder.
- The outputs of the decoder are designated by the symbols D0 through D7.
- The operation code at bit 15 is transferred to a flip-flop designated by the symbol I.
- The operation codes from Bits 0 through 11 are applied to the control logic gates.

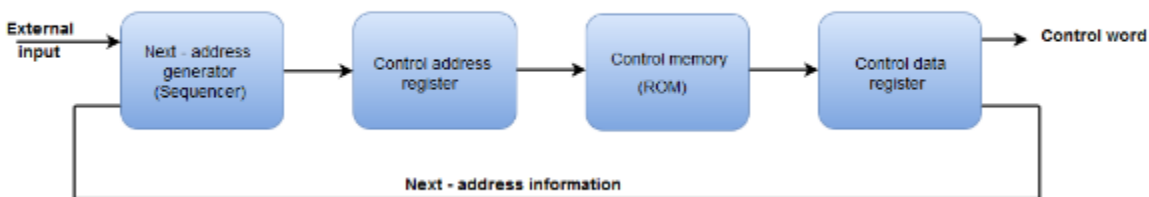
- The Sequence counter (SC) can count in binary from 0 through 15.

The Microprogrammed Control organization is implemented by using the programming approach.

In Microprogrammed Control, the micro-operations are performed by executing a program consisting of micro-instructions.

The following image shows the block diagram of a Microprogrammed Control organization.

Microprogrammed Control Unit of a Basic Computer:



- The Control memory address register specifies the address of the micro-instruction.
- The Control memory is assumed to be a ROM, within which all control information is permanently stored.
- The control register holds the microinstruction fetched from the memory.
- The micro-instruction contains a control word that specifies one or more micro-operations for the data processor.
- While the micro-operations are being executed, the next address is computed in the next address generator circuit and then transferred into the control address register to read the next microinstruction.
- The next address generator is often referred to as a micro-program sequencer, as it determines the address sequence that is read from control memory.

Central Processing Unit:

The operation or task that must perform by CPU is:

- Fetch Instruction: The CPU reads an instruction from memory.
- Interpret Instruction: The instruction is decoded to determine what action is required.

- Fetch Data: The execution of an instruction may require reading data from memory or I/O module.
- Process data: The execution of an instruction may require performing some arithmetic or logical operation on data.
- Write data: The result of an execution may require writing data to memory or an I/O module.

To do these tasks, it should be clear that the CPU needs to store some data temporarily. It must remember the location of the last instruction so that it can know where to get the next instruction. It needs to store instructions and data temporarily while an instruction is being executed. In other words, the CPU needs a small internal memory. These storage locations are generally referred as registers. The major components of the CPU are an arithmetic and logic unit (ALU) and a control unit (CU). The ALU does the actual computation or processing of data. The CU controls the movement of data and instruction into and out of the CPU and controls the operation of the ALU. The CPU is connected to the rest of the system through system bus. Through system bus, data or information gets transferred between the CPU and the other component of the system. The system bus may have three components: Data Bus: Data bus is used to transfer the data between main memory and CPU. Address Bus: Address bus is used to access a particular memory location by putting the address of the memory location. Control Bus: Control bus is used to provide the different control signal generated by CPU to different part of the system. As for example, memory read is a signal generated by CPU to indicate that a memory read operation has to be performed. Through control bus this signal is transferred to memory module to indicate the required operation.

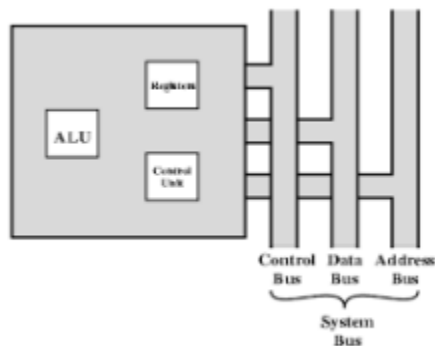
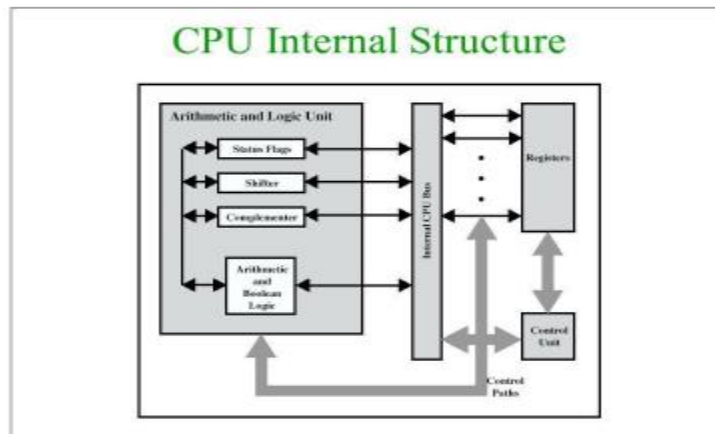


Figure 1: CPU with the system bus.



Stack Organization:

A useful feature that is included in the CPU of most computers is a stack or last in, first out (LIFO) list. A stack is a storage device that stores information in such a manner that the item stored last is the first item retrieved. The operation of a stack can be compared to a stack of trays. The last tray placed on top of the stack is the first to be taken off. The stack in digital computers is essentially a memory unit with an address register that can only (after an initial value is loaded in to it). The register that holds the address for the stack is called a stack pointer (SP) because its value always points at the top item in stack. Contrary to a stack of trays where the tray itself may be taken out or inserted, the physical registers of a stack are always available for reading or writing. The two operations of a stack are the insertion and deletion of items. The operation of insertion is called PUSH because it can be thought of as the result of pushing a new item on top. The operation of deletion is called POP because it can be thought of as the result of removing one item so that the stack pops up. However, nothing is pushed or popped in a computer stack. These operations are simulated by incrementing or decrementing the stack pointer register.

INSTRUCTION FORMATS:

We know that a machine instruction has an opcode and zero or more operands. Encoding an instruction set can be done in a variety of ways. Architectures are differentiated from one another by the number of bits allowed per instruction (16, 32, and 64 are the most common), by the number of operands allowed per instruction, and by the types of instructions and data each can process. More specifically, instruction sets are differentiated by the following features:

1. Operand storage in the CPU (data can be stored in a stack structure or in registers)
2. Number of explicit operands per instruction (zero, one, two, and three being the most common)
3. Operand location (instructions can be classified as register-to-register, register-to-memory or memory-to-memory, which simply refer to

the combinations of operands allowed per instruction) 4. Operations (including not only types of operations but also which instructions can access memory and which cannot) 5. Type and size of operands (operands can be addresses, numbers, or even characters) Number of Addresses: One of the characteristics of the ISA(Industrial Standard Architecture) that shapes the architecture is the number of addresses used in an instruction. Most operations can be divided into binary or unary operations. Binary operations such as addition and multiplication require two input operands whereas the unary operations such as the logical NOT need only a single operand. Most operations produce a single result. There are exceptions, however. For example, the division operation produces two outputs: a quotient and a remainder. Since most operations are binary, we need a total of three addresses: two addresses to specify the two input operands and one to specify where the result should go.

Three-Address Machines:

In three-address machines, instructions carry all three addresses explicitly. The RISC processors use three addresses. Table X1 gives some sample instructions of a threeaddress machine.

In these machines, the C statement

$$A = B + C * D - E + F + A$$

is converted to the following code:

mult T,C,D ; $T = C * D$

add T,T,B ; $T = B + C * D$

sub T,T,E ; $T = B + C * D - E$

add T,T,F ; $T = B + C * D - E + F$

add A,T,A ; $A = B + C * D - E + F + A$

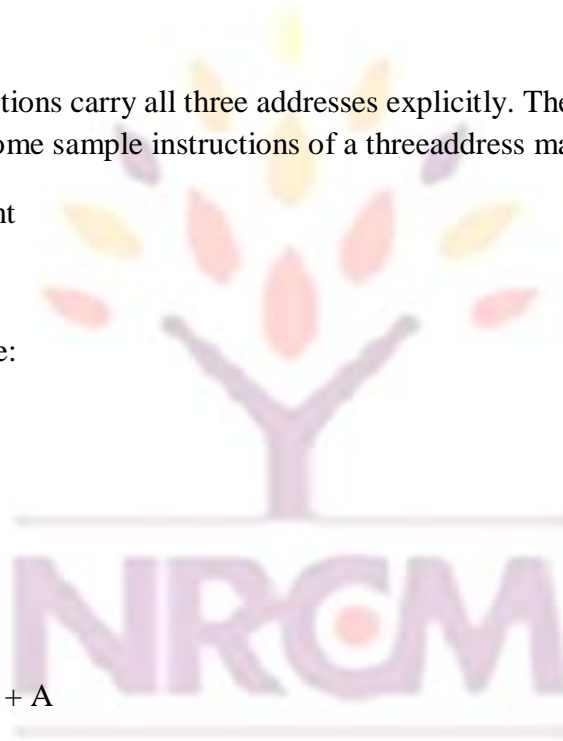


Table :T1 Sample three-address machine instructions

Instruction	Semantics
add dest,src1,src2	Adds the two values at src1 and src2 and stores the result in dest $M(dest) = [src1] + [src2]$
sub dest,src1,src2	Subtracts the second source operand at src2 from the first at src1 and stores the result in dest $M(dest) = [src1] - [src2]$
mult dest,src1,src2	Multiplies the two values at src1 and src2 and stores the result in dest $M(dest) = [src1] * [src2]$

We use the notation that each variable represents a memory address that stores the value associated with that variable. This translation from symbol name to the memory address is done by using a symbol table.

As you can see from this code, there is one instruction for each arithmetic operation. Also notice that all instructions, barring the first one, use an address twice. In the middle three instructions, it is the temporary T and in the last one, it is A. This is the motivation for using two addresses, as we show next.

Two-Address Machines : In two-address machines, one address doubles as a source and destination. Usually, we use dest to indicate that the address is used for destination. But you should note that this address also supplies one of the source operands. The Pentium is an example processor that uses two addresses. Sample instructions of a two-address machine On these machines, the C statement $A = B + C * D - E + F + A$ is converted to the following code: load T,C ; T = C mult T,D ; T = C*D add T,B ; T = B + C*D sub T,E ; T = B + C*D - E add T,F ; T = B + C*D - E + F add A,T ; A = B + C*D - E + F + A

Table :T2 Sample Two-address machine instructions:

Instruction	Semantics
load dest,src	Copies the value at src to dest $M(dest) = [src]$
add dest,src	Adds the two values at src and dest and stores the result in dest $M(dest) = [dest] + [src]$
sub dest,src	Subtracts the second source operand at src from the first at dest and stores the result in dest $M(dest) = [dest] - [src]$
mult dest,src	Multiplies the two values at src and dest and stores the result in dest $M(dest) = [dest] * [src]$

One-Address Machines : In the early machines, when memory was expensive and slow, a special set of registers was used to provide an input operand as well as to receive the result from the ALU. Because of this, these registers are called the accumulators. In most machines, there is just a single accumulator register. This kind of design, called accumulator machines, makes sense if memory is expensive. In accumulator machines, most operations are performed on the contents of the accumulator and the operand supplied by the instruction. Thus, instructions for these machines need to specify only the address of a single operand. There is no need to

store the result in memory: this reduces the need for larger memory as well as speeds up the computation by reducing the number of memory accesses. A few sample accumulator machine instructions are shown in Table X3. In these machines, the C statement $A = B + C * D - E + F + A$ is converted to the following code: load C ; load C into the accumulator mult D ; accumulator = $C * D$ add B ; accumulator = $C * D + B$ sub E ; accumulator = $C * D + B - E$ add F ; accumulator = $C * D + B - E + F$.

add A ; accumulator = $C * D + B - E + F + A$ store A ; store the accumulator contents in A

Table :T3 Sample ONE-address machine instructions

Instruction	addr	Semantics
load	addr	Copies the value at address addr into the accumulator $\text{accumulator} = [\text{addr}]$
store	addr	Stores the value in the accumulator at the memory address addr $M(\text{addr}) = \text{accumulator}$
add	addr	Adds the contents of the accumulator and value at address addr $\text{accumulator} = \text{accumulator} + [\text{addr}]$
sub	addr	Subtracts the value at memory address addr from the contents of the accumulator $\text{accumulator} = \text{accumulator} - [\text{addr}]$
mult	addr	Multiplies the contents of the accumulator and value at address addr $\text{accumulator} = \text{accumulator} * [\text{addr}]$

Zero-Address Machines : In zero-address machines, locations of both operands are assumed to be at a default location. These machines use the stack as the source of the input operands and the result goes back into the stack. Stack is a LIFO (last-in-first-out) data structure that all processors support, whether or not they are zero-address machines. As the name implies, the last item placed on the stack is the first item to be taken out of the stack. A good analogy is the stack of trays you find in a cafeteria. All operations on this type of machine assume that the required input operands are the top two values on the stack. The result of the operation is placed on top of the stack. Table X4 gives some sample instructions for the stack machines.

Table :T4 Sample Zero-address machine instructions

Instruction	Semantics
push addr	Places the value at address addr on top of the stack $\text{push}([\text{addr}])$
pop addr	Stores the top value on the stack at memory address addr $M(\text{addr}) = \text{pop}$
add	Adds the top two values on the stack and pushes the result onto the stack $\text{push}(\text{pop} + \text{pop})$
sub	Subtracts the second top value from the top value of the stack and pushes the result onto the stack $\text{push}(\text{pop} - \text{pop})$
mult	Multiplies the top two values in the stack and pushes the result onto the stack $\text{push}(\text{pop} * \text{pop})$

two are special instructions that use a single address and are used to move data between memory and stack.

All other instructions use the zero-address format. Let's see how the stack machine translates the arithmetic expression we have seen in the previous subsections. In these machines, the C statement

$$A = B + C * D - E + F + A$$

is converted to the following code:

```
push E ; <E>
push C ; <C, E>
push D ; <D, C, E>
mult ; <C*D, E>
push B ; <B, C*D, E>
add ; <B+C*D, E>
sub ; <B+C*D-E>
push F ; <F, B+C*D-E>
add ; <F+B+C*D-E>
push A ; <A, F+B+C*D-E>
add ; <A+F+B+C*D-E>
pop A ; <>
```

On the right, we show the state of the stack after executing each instruction.

The top element of the stack is shown on the left. Notice that we pushed E early because we need to subtract it from $(B+C*D)$.

Stack machines are implemented by making the top portion of the stack internal to the processor. This is referred to as the stack depth. The rest of the stack is placed in memory. Thus, to access the top values that are within the stack depth, we do not have to access the memory. Obviously, we get better performance by increasing the stack depth.

Addressing Modes

We have examined the types of operands and operations that may be specified by machine instructions. Now we have to see how is the address of an operand specified, and how are the bits of an instruction organized to define the operand addresses and operation of that instruction

Addressing Modes: The most common addressing techniques are

- Immediate
- Direct
- Indirect
- Register
- Register Indirect
- Displacement
- Stack

All computer architectures provide more than one of these addressing modes.

The question arises as to how the control unit can determine which addressing mode is being used in a particular instruction. Several approaches are used. Often, different opcodes will use different addressing modes. Also, one or more bits in the instruction format can be used as a mode field. The value of the mode field determines which addressing mode is to be used.

What is the interpretation of effective address. In a system without virtual memory, the effective address will be either a main memory address or a register. In a virtual memory system, the effective address is a virtual address or a register. The actual mapping to a physical address is a function of the paging mechanism and is invisible to the programmer. To explain the addressing modes, we use the following notation:

A	=	contents of an address field in the instruction that refers to a memory
R	=	contents of an address field in the instruction that refers to a register
EA	=	actual (effective) address of the location containing the referenced operand
(X)	=	contents of location X

Immediate Addressing:

The simplest form of addressing is immediate addressing, in which the operand is actually present in the instruction: $OPERAND = A$ This mode can be used to define and use constants or set initial values of variables. The advantage of immediate addressing is that no memory reference other than the instruction fetch is required to obtain the operand. The disadvantage is that the size of the

number is restricted to the size of the address field, which, in most instruction sets, is small compared with the word length.

Direct Addressing:

A very simple form of addressing is direct addressing, in which the address field contains the effective address of the operand:

$$EA = A$$

It requires only one memory reference and no special calculation.

Indirect Addressing:

With direct addressing, the length of the address field is usually less than the word length, thus limiting the address range. One solution is to have the address field refer to the address of a word in memory, which in turn contains a full-length address of the operand.

Register Addressing: Register addressing is similar to direct addressing. The only difference is that the address field refers to a register rather than a main memory address: $EA = R$

The advantages of register addressing are that only a small address field is needed in the instruction and no memory reference is required. The disadvantage of register addressing is that the address space is very limited.

The exact register location of the operand in case of Register Addressing Mode is shown in the Figure 34.4. Here, 'R' indicates a register where the operand is present.

Register Indirect Addressing:

Register indirect addressing is similar to indirect addressing, except that the address field refers to a register instead of a memory location. It requires only one memory reference and no special calculation.

$$EA = (R)$$

Register indirect addressing uses one less memory reference than indirect addressing. Because, the first information is available in a register which is nothing but a memory address. From that memory location, we use to get the data or information. In general, register access is much more faster than the memory access.

Displacement Addressing: A very powerful mode of addressing combines the capabilities of direct addressing and register indirect addressing, which is broadly categorized as displacement addressing: $EA = A + (R)$ Displacement addressing requires that the instruction have two address fields, at least one of which is explicit. The value contained in one address field (value = A) is used directly. The other address field, or an implicit reference based on opcode, refers to a register whose contents are

added to A to produce the effective address. The general format of Displacement Addressing is shown in the Figure 4.6. Three of the most common use of displacement addressing are: • Relative addressing • Base-register addressing • Indexing

Relative Addressing: For relative addressing, the implicitly referenced register is the program counter (PC). That is, the current instruction address is added to the address field to produce the EA. Thus, the effective address is a displacement relative to the address of the instruction. **Base-Register Addressing:** The reference register contains a memory address, and the address field contains a displacement from that address. The register reference may be explicit or implicit. In some implementation, a single segment/base register is employed and is used implicitly. In others, the programmer may choose a register to hold the base address of a segment, and the instruction must reference it explicitly. **Indexing:** The address field references a main memory address, and the reference register contains a positive displacement from that address. In this case also the register reference is sometimes explicit and sometimes implicit.

UNIT 3:

Data Representation: Data types, Complements, Fixed Point Representation, Floating Point Representation.

Computer Arithmetic: Addition and subtraction, multiplication Algorithms, Division Algorithms, Floating-point Arithmetic operations. Decimal Arithmetic unit, Decimal Arithmetic operations.

BASIC COMPUTER DATA TYPES:

- Binary information in digital computers is stored in memory or processor registers.
- The data types found in the registers of digital computers may be classified as being one of the following categories: (1) numbers used in arithmetic computations, (2) letters of the alphabet used in data processing, and (3) other discrete symbols used for specific purposes. All types of data, except binary numbers, are represented in computer registers in binary-coded form.
- A number system of base or radix r is a system of that uses distinct symbols for r digits
- The decimal number system in everyday use employs radix 10 system. The 10 symbols are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9; highest number being $r-1$
The binary number system uses the radix 2. The two digit symbols used are 0 and 1.
- The string of digits 101101 is interpreted to represent $1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 45$
- Besides the decimal and binary number systems,
- Octal (radix 8)- 0,1,2,3,4,5,6,7 and

- Hexadecimal (radix 16) – 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F • Octal can be converted to decimal as follows
 $(736.4)_8 = 7 \times 8^2 + 3 \times 8^1 + 6 \times 8^0 + 4 \times 8^{-1} = 7 \times 64 + 3 \times 8 + 6 \times 1 + 4/8 = (478.5)_{10}$

Basic computer data types

- Hexadecimal can be converted to decimal

$$\begin{aligned}(F3)_{16} &= F \times 16 + 3 \\ &= 15 \times 16 + 3 \\ &= (243)_{10}\end{aligned}$$

Octal and Hex can be obtained from Binary as shown below:

1	2	7	5	4	3	Octal									
1	0	1	0	1	1	1	0	1	1	0	0	0	1	1	Binary
A	F	6	3	Hexadecimal											

Complements:

- A binary code is a group of n bits that assume upto 2^n distinct combinations of 0s and 1s.
- A BCD code is a binary coded decimal i.e. binary coding decimal numbers.
- ASCII (American Standard Code for Information Interchange),
- which uses seven bits to code 128 characters is standard alphanumeric character code.
- Complements are used in digital computers for simplifying the subtraction operation and for logical manipulation. There are two types of complements
- For each base r system: the r's complement and the (r - 1)'s Complement

For binary base 2 system: the 2's complement and the 1's complement • For decimal base 10 system: the 10's complement and the 9's complement • The 9's complement of 546700 is $999999 - 546700 = 453299$ • The 9's complement of 12389 is $99999 - 12389 = 87610$ • The 1's complement of a binary number is formed by Changing 1's into 0's and 0's into 1's. • For example, the 1's complement of 1011001 is 0100110 and the 1's complement of 0001111 is 1110000.

The 10's complement of the decimal 2389 is $7610 + 1 = 7611$ and is obtained by adding 1 to the 9's complement value.

- The 2's complement of binary 101100 is $010011 + 1 = 010100$ and is obtained by adding 1 to the 1's complement value.

- For example, the 1's complement of 1011001 is 0100110 and the 1's complement of 0001111 is 1110000.

Solve: Find the 9's and 10's complement of 246700. Find the 1's and 2's complement of 1101100.

Fixed-Point Representation

- In computer binary systems it is customary to represent the sign with a bit placed in the leftmost position of the number.
- sign bit is equal to 0 for positive and to 1 for negative.
- a number may have a binary (or decimal) point.
- There are two ways of specifying the position of the binary point: by giving it a fixed position or by employing a floatingpoint representation.
- The fixed-point method assumes that the binary point is always. fixed in one position.
- The two positions most widely used are (1) a binary point in the extreme left of the register to make the stored number a fraction, and (2) a binary point in the extreme right of the register to make the stored number an integer. In either case, the binary point is not actually present.

Integer Representation for signed numbers

- signed-magnitude representation 1 0001110
- signed-1's complement representation 1 1110001
- signed-2's complement representation 1 1110010

Floating Point Representation:

The floating-point representation uses a second register to store a number that designates the position of the decimal point in the first register. • The floating-point representation of a number has two parts. The first part represents a signed, fixed-point number called the mantissa. The second part designates the position of the decimal (or binary) point and is called the exponent The fixedpoint mantissa may be a fraction or an integer. For example, • the decimal number +6132.789 is represented in floating-point with a fraction and an exponent as follows:

Fraction	Exponent
+0.6132789	+04

A floating-point binary number is represented in a similar manner except that it uses base 2 for the exponent.

- For example, the binary number +1001.11 is represented with an 8-bit fraction and 6-bit exponent as follows:

Fraction	Exponent
01001110	00010

Sign-Magnitude

used in every day arithmetic calculations .

Left most bit is sign bit- **0 – positive** **1 – negative**

- +18 = **00010010**

- -18 = **10010010**

- Problems

—Need to consider both sign and magnitude in arithmetic

—Two representations of zero (+0 and -0)

Addition and Subtraction:

Normal binary addition

- Monitor sign bit for overflow
- So we only need addition and complement

Circuits

Assume:-

-magnitudes of two numbers as A and B.

when those sign numbers are added we have eight different conditions depending on the sign bits and operations performed.

SIGNED MAGNITUDE ADDITION AND SUBTRACTION

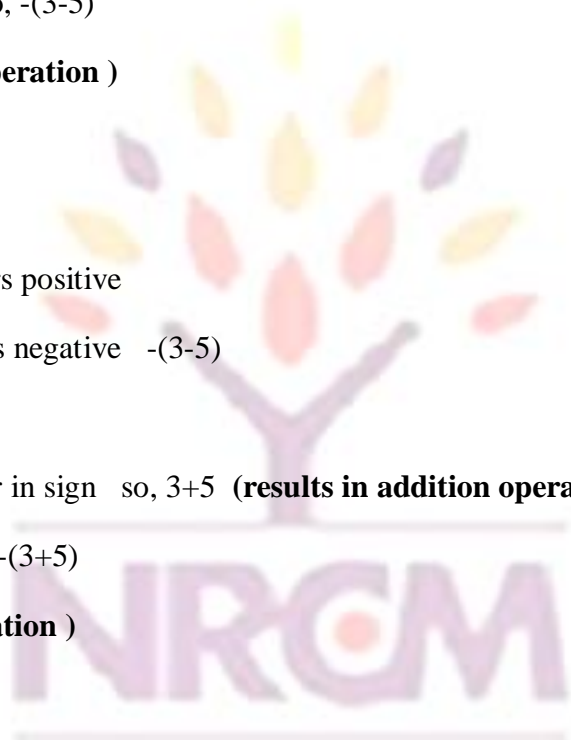
- Addition: $A + B$; A: Augend; B: Addend
- Subtraction: $A - B$; A: Minuend; B: Subtrahend

Case 1: addition

- **If signs are same:**
- $3+5$ both numbers positive
- $(-3) + (-5)$ both numbers negative $-(3+5)$
- **If signs differ:**
- $(+3) + (-5)$ numbers differ in sign so, $3-5$ (results in subtraction operation)
- $(-3) + (+5)$ entin signs so, $-(3-5)$
- (results in subtraction operation)

Case 2: Subtraction

- **If signs are same:**
- $3-5$ both numbers positive
- $(-3) - (-5)$ both numbers negative $-(3-5)$
- **If signs differ:**
- $(+3) - (-5)$ numbers differ in sign so, $3+5$ (results in addition operation)
- $(-3) - (+5)$ entin signs so, $-(3+5)$
- (results in addition operation)

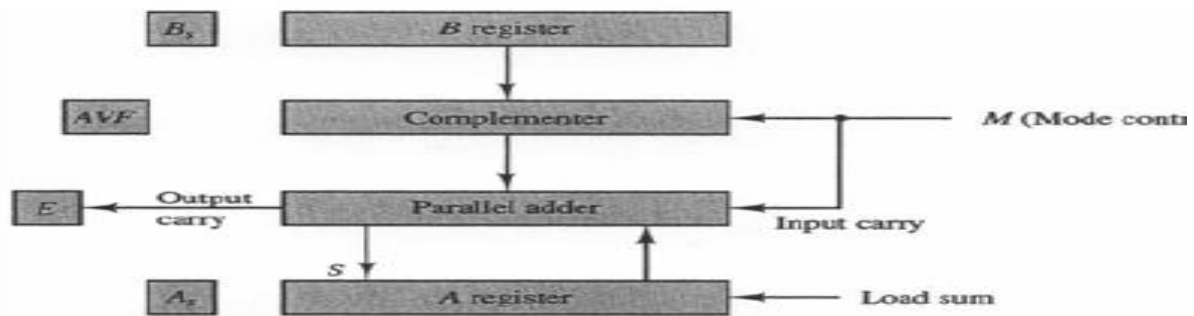


Add / Subtract Signed-Magnitude

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

Forces zero to be positive

Hardware



- A_s Sign of A
- B_s Sign of B
- A_s & A Accumulator
- AVF Overflow bit for $A + B$
- E Output carry for parallel adder

ALGORITHM:

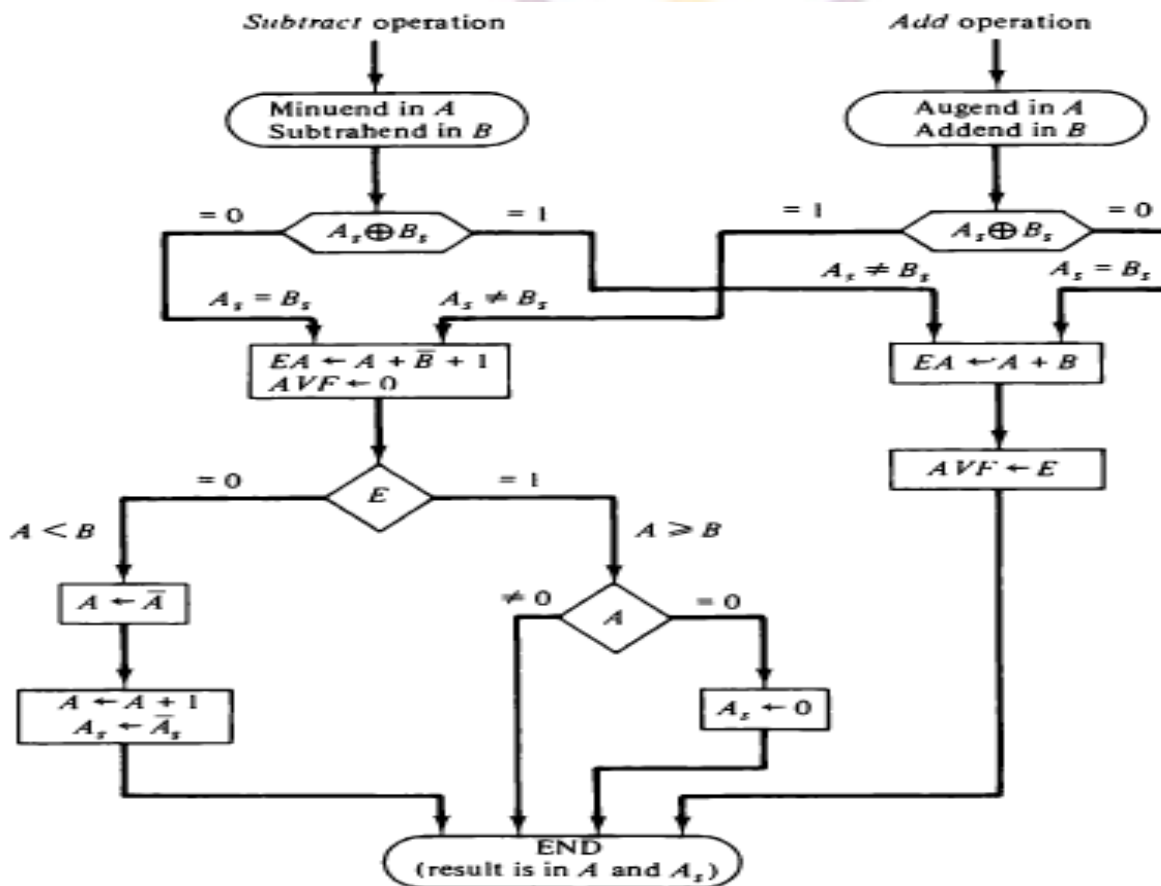
The two signs A_s and B_s are compared by EX-OR them. If result is 0 then $A_s = B_s$ and if result is 1 then $A_s \neq B_s$.

o For add operations if have same sign bits the magnitude must be added. For subtract operations different sign bits means magnitudes be added as well.

o E bit is carry bit after addition and moves to AVF overflow bit only at this state.

o If sign bits are different in add operations or the same in subtract operations the two magnitudes will be subtracted $A - B$. No overflow can occur here.

o After subtract if $E=1$ this means $A > B$ and if $E=0$ then $A < B$. then here it is necessary to get 2's complement of A (by invert A then add 1) and sign of A is inverted only in this case.



SIGNED 2'S COMPLEMENT ADDITION AND SUBTRACTION:

The left most bit in 2's complement represented binary number is the sign bit. If 0 the number is positive and if 1 then number is negative. If sign bit is 1 the entire number is represented in 2's complement.

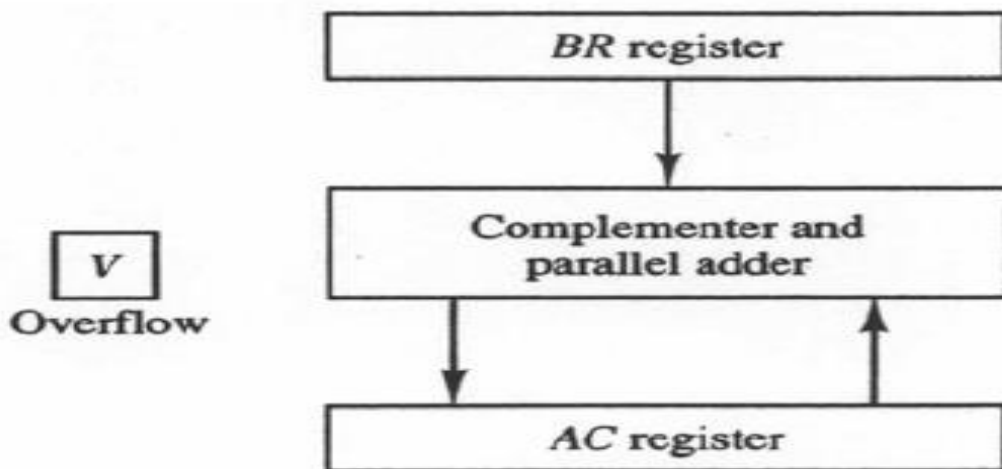
The addition of two numbers represented in 2's complement is carried out by normal binary addition with carry discarded.

The subtraction is carried out by taking 2's complement (B) of subtrahend and adding it to minuend (A).

Overflow can be detected by inspecting last 2 carries out of addition by EX-OR them. If different then overflow is detected.

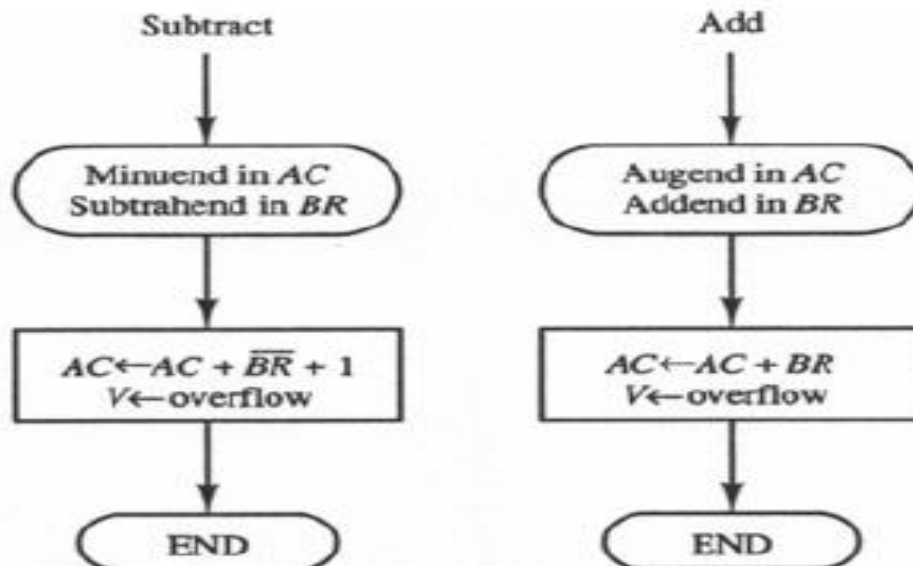
For addition simply implement add then see overflow. For subtract add 2's complement of B to A and watch overflow since the A and -B could be of same sign.

Add / Sub Signed-2's Complement



7

Algorithm

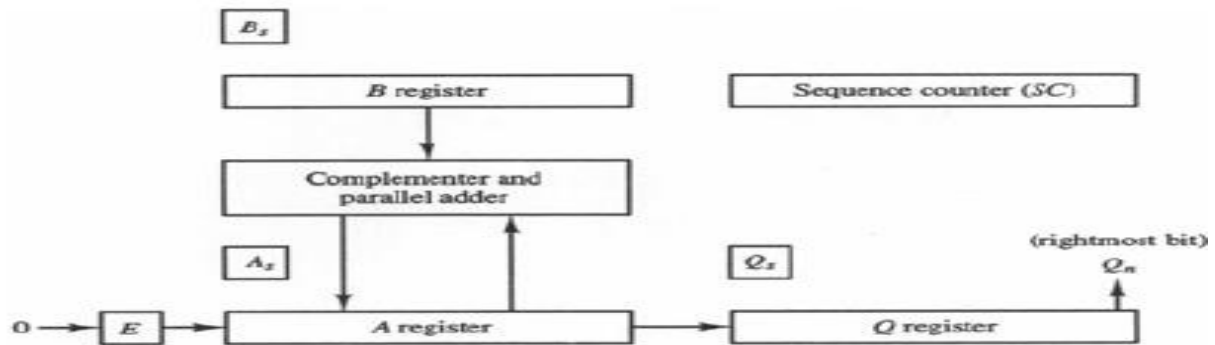


Multiply Signed-Magnitude

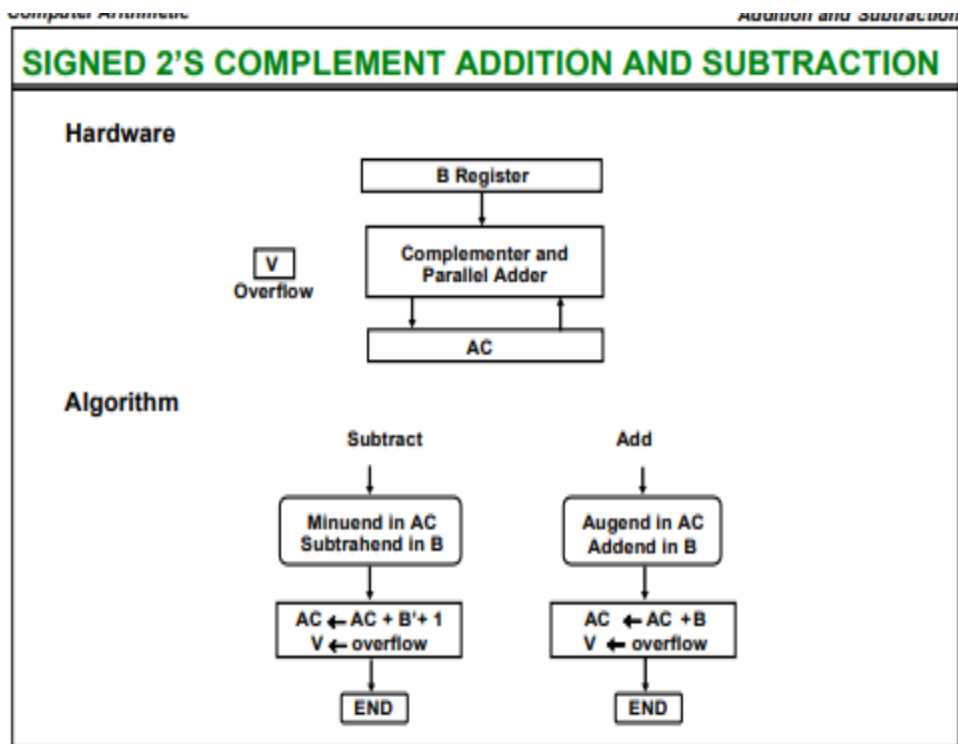
- Series of successive shift and add operations

• 23	10111	Multiplicand
<u>19</u>	<u>x 10011</u>	Multiplier
	10111	
	10111	
	00000	
	00000	
	<u>10111</u>	+
437	110110101	Product

Hardware



- Q multiplier
- B multiplicand
- A 0
- SC number of bits in multiplier
- E overflow bit for A
- Do SC times
 - If low-order bit of Q is 1
 - ◆ $A \leftarrow A + B$
 - Shift right EAQ
- Product is in AQ

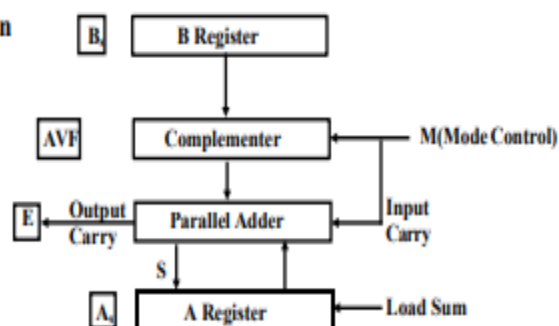


Computer Arithmetic

Addition and Subtraction

SIGNED MAGNITUDE ADDITION AND SUBTRACTIONAddition: $A + B$; A: Augend; B: AddendSubtraction: $A - B$; A: Minuend; B: Subtrahend

Operation	Add Magnitude	Subtract Magnitude		
		When $A > B$	When $A < B$	When $A = B$
$(+A) + (+B)$	$+(A + B)$			
$(+A) + (-B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(-A) + (+B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$
$(-A) + (-B)$	$-(A + B)$			
$(+A) - (+B)$		$+(A - B)$	$-(B - A)$	$+(A - B)$
$(+A) - (-B)$	$+(A + B)$			
$(-A) - (+B)$	$-(A + B)$			
$(-A) - (-B)$		$-(A - B)$	$+(B - A)$	$+(A - B)$

Hardware Implementation

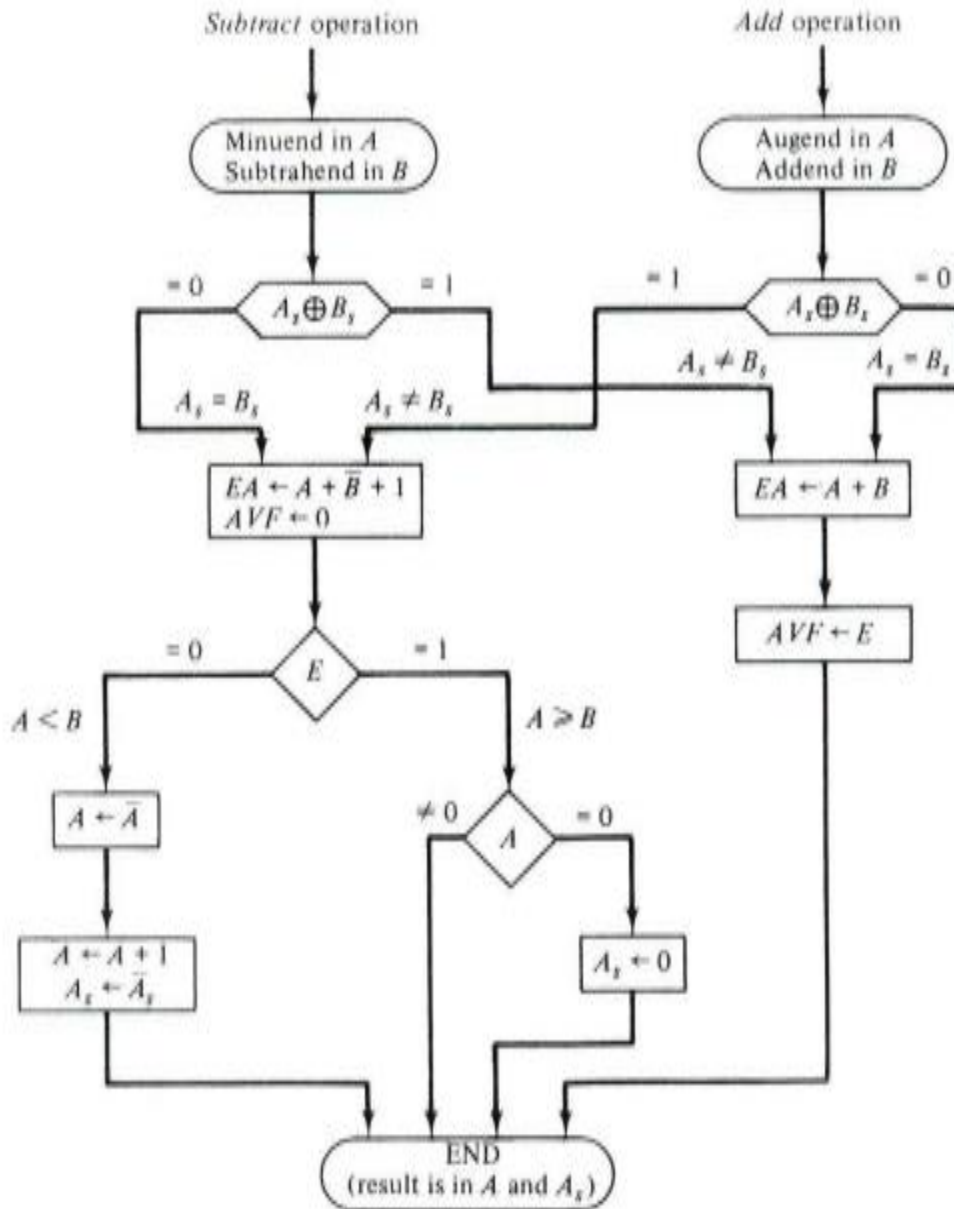
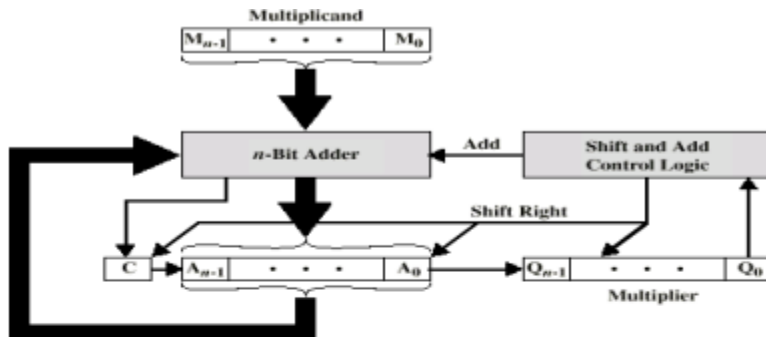


Figure 10-2 Flowchart for add and subtract operations.

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

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



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



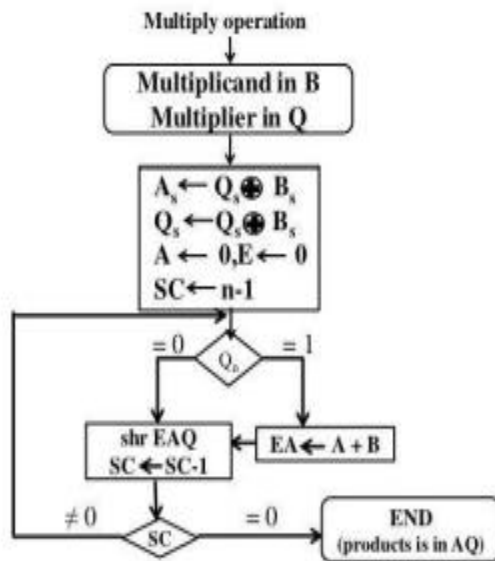


Figure: Flowchart for multiply operation.

Booth's algorithm :

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

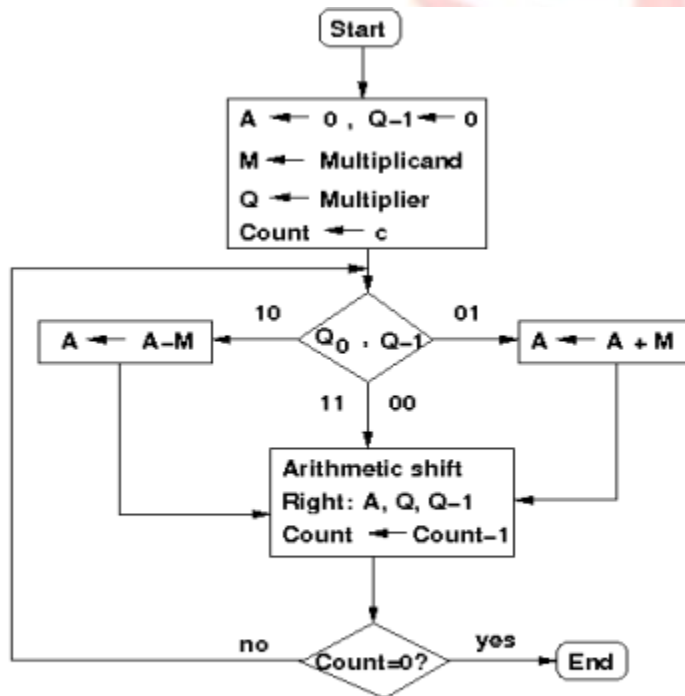
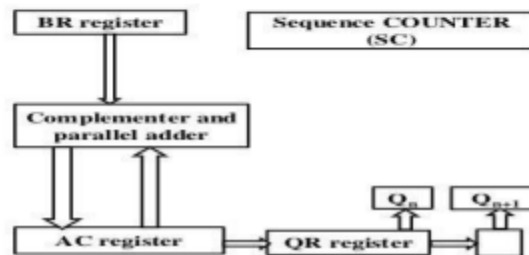


shifting, and a string of 1's in the multiplier from bit weight 2^k to weight 2^m can be treated as $2^{k+1} - 2^m$.

- For example, the binary number 001110 (+14) has a string 1's from 2^3 to 2^1 ($k=3, m=1$). The number can be represented as $2^{k+1} - 2^m = 2^4 - 2^1 = 16 - 2 = 14$. Therefore, the multiplication $M \times 14$, where M is the multiplicand and 14 the multiplier, can be done as $M \times 2^4 - M \times 2^1$.
- Thus the product can be obtained by shifting the binary multiplicand M four times to the left and subtracting M shifted left once.

Hardware for Booth Algorithm

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

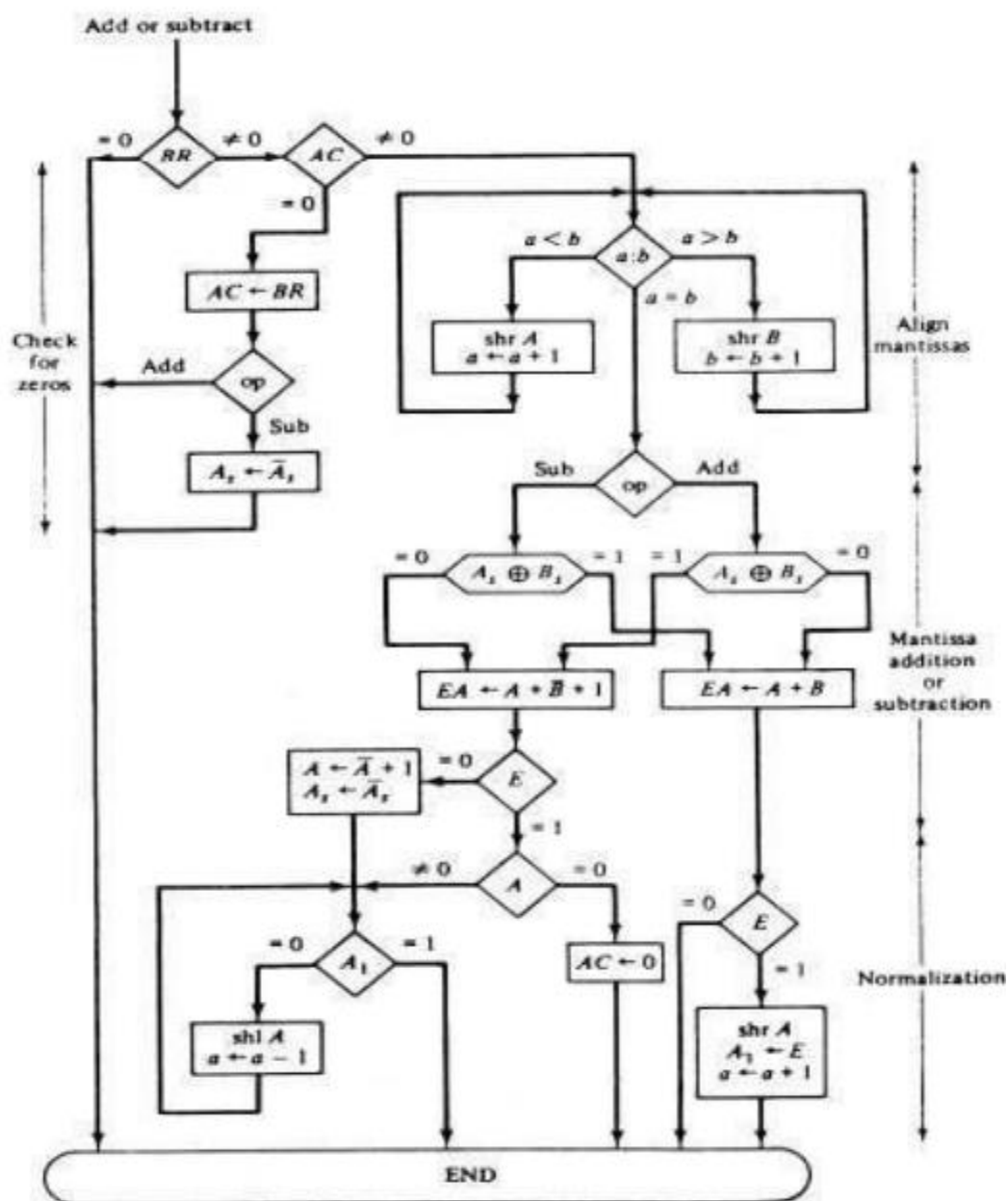


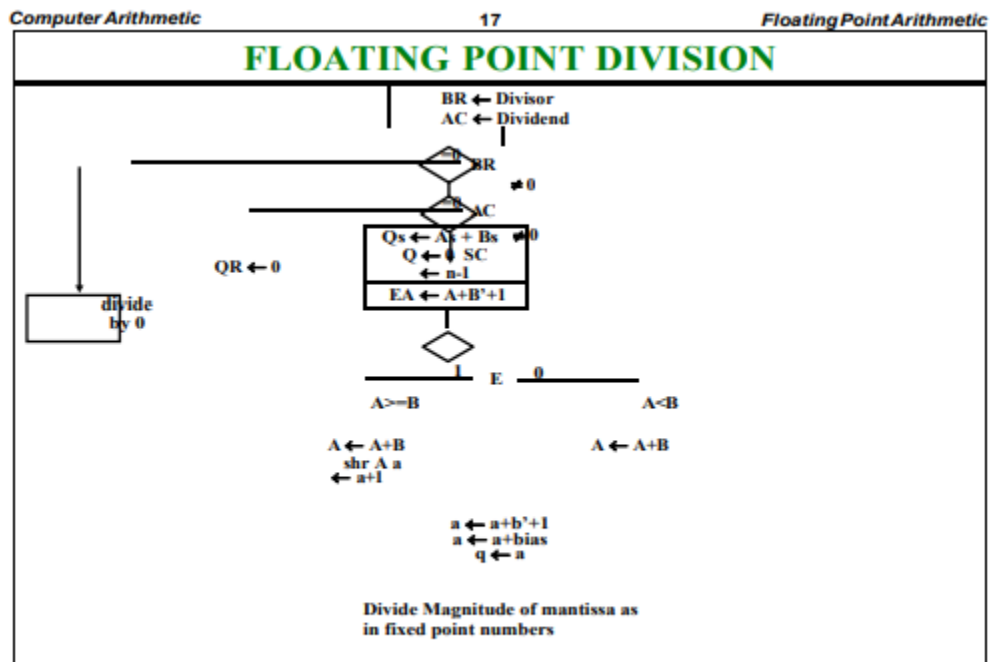
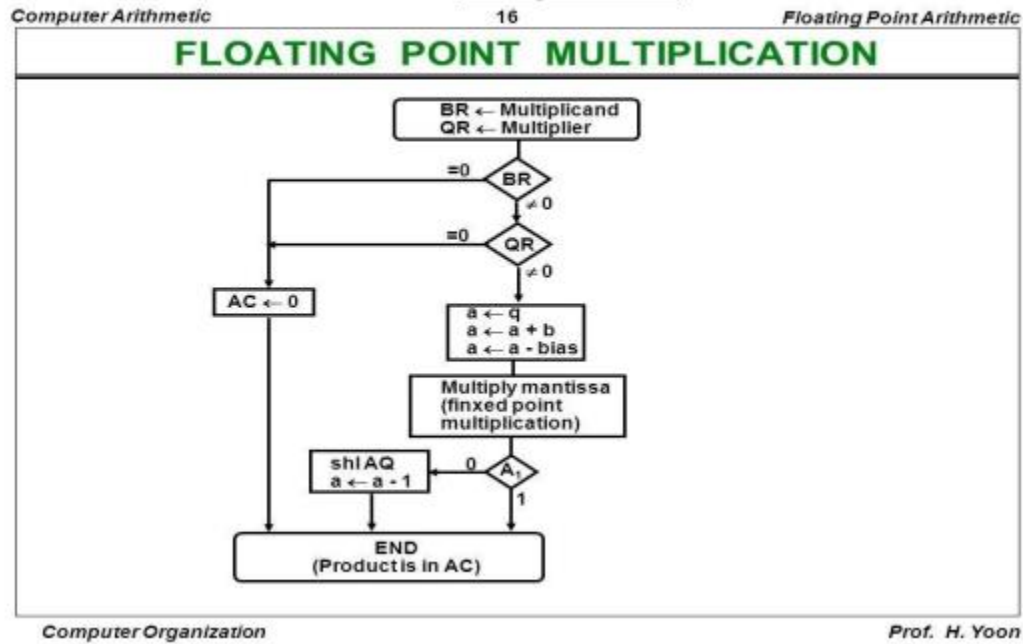
- As in all multiplication schemes, booth algorithm requires examination of the multiplier bits and shifting of partial product.

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

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





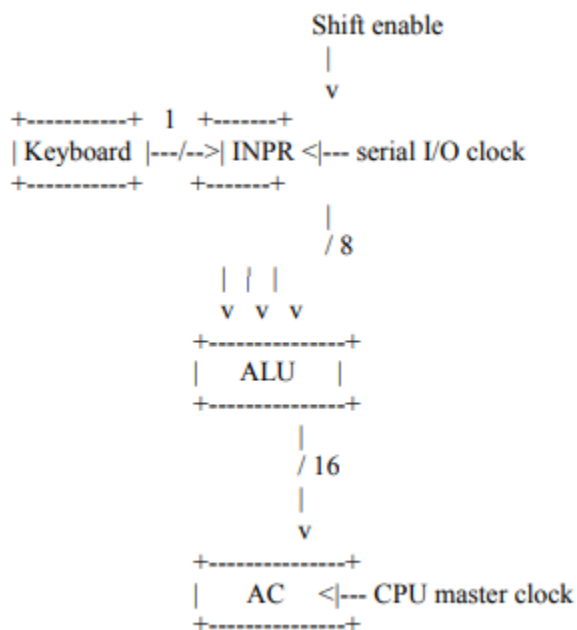
Multiplication:

UNIT 4:

Input-Output Organization: Input-Output Interface, Asynchronous data transfer, Modes of Transfer, Priority Interrupt Direct memory Access.

Memory Organization: Memory Hierarchy, Main Memory, Auxiliary memory, Associate Memory, Cache Memory.

The Basic Computer I/O consists of a simple terminal with a keyboard and a printer/monitor. The keyboard is connected serially (1 data wire) to the INPR register. INPR is a shift register capable of shifting in external data from the keyboard one bit at a time. INPR outputs are connected in parallel to the ALU.



How many CPU clock cycles are needed to transfer a character from the keyboard to the INPR register? (tricky)

Are the clock pulses provided by the CPU master clock?

RS232, USB, Firewire are serial interfaces with their own clock independent of the CPU. (USB speed is independent of processor speed.)

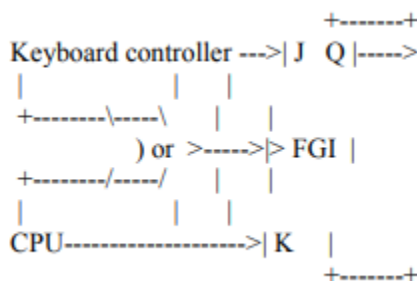
☐ RS232: 115,200 kbps (some faster)

☐ USB: 11 mbps

- USB2: 480 mbps
- FW400: 400 mbps
- FW800: 800 mbps
- USB3: 4.8 gbps

OUTR inputs are connected to the bus in parallel, and the output is connected serially to the terminal. OUTR is another shift register, and the printer/monitor receives an end-bit during each clock pulse.

I/O Operations Since input and output devices are not under the full control of the CPU (I/O events are asynchronous), the CPU must somehow be told when an input device has new input ready to send, and an output device is ready to receive more output. The FGI flip-flop is set to 1 after a new character is shifted into INPR. This is done by the I/O interface, not by the control unit. This is an example of an asynchronous input event (not synchronized with or controlled by the CPU). The FGI flip-flop must be cleared after transferring the INPR to AC. This must be done as a microoperation controlled by the CU, so we must include it in the CU design. The FGO flip-flop is set to 1 by the I/O interface after the terminal has finished displaying the last character sent. It must be cleared by the CPU after transferring a character into OUTR. Since the keyboard controller only sets FGI and the CPU only clears it, a JK flip-flop is convenient:



How do we control the CK input on the FGI flip-flop? (Assume leading-edge triggering.) There are two common methods for detecting when I/O devices are ready, namely software polling and interrupts. These two methods are discussed in the following sections. Table 5-5 outlines the Basic Computer input-output instructions.

Interrupts To alleviate the problems of software polling, a hardware solution is needed. Analogies to software polling in daily life tend to look rather silly. For example, imagine a teacher is analogous to a CPU, and the students are I/O devices. The students are working asynchronously, as the teacher walks around the room constantly asking each individual student "are you done yet?". What would be a better approach? With interrupts, the running program is not responsible for

checking the status of I/O devices. Instead, it simply does its own work, and assumes that I/O will take care of itself! When a device becomes ready, the CPU hardware initiates a branch to an I/O subprogram called an interrupt service routine (ISR), which handles the I/O transaction with the device. An interrupt can occur during any instruction cycle as long as interrupts are enabled. When the current instruction completes, the CPU interrupts the flow of the program, executes the ISR, and then resumes the program. The program itself is not involved and is in fact unaware that it has been interrupted. Figure 5-13 outlines the Basic Computer interrupt process. Interrupts can be globally enabled or disabled via the IEN flag (flip-flop). Some architectures have a separate ISR for each device. The Basic Computer has a single ISR that services both the input and output devices. If interrupts are enabled, then when either FGI or FGO gets set, the R flag also gets set. ($R = FGI \vee FGO$) This allows the system to easily check whether any I/O device needs service. Determining which one needs service can be done by the ISR. If $R = 0$, the CPU goes through a normal instruction cycle. If $R = 1$, the CPU branches to the ISR to process an I/O transaction. How much time does checking for interrupts add to the instruction cycle? Interrupts are usually disabled while the ISR is running, since it is difficult to make an ISR reentrant. (Callable while it is already in progress, such as a recursive function.) Hence, IEN and R are cleared as part of the interrupt cycle. IEN should be re-enabled by the ISR when it is finished. (In many architectures this is done by a special return instruction to ensure that interrupts are not enabled before the return is actually executed.) The Basic Computer interrupt cycle is shown in figure 5-13 (above). The Basic Computer interrupt cycle in detail: T0'T1'T2'(IEN)(FGI \vee FGO): $R \leftarrow 1$ 109 RT0: $AR \leftarrow 0$, $TR \leftarrow PC$ RT1: $M[AR] \leftarrow TR$, $PC \leftarrow 0$ RT2: $PC \leftarrow PC + 1$, $IEN \leftarrow 0$, $R \leftarrow 0$, $SC \leftarrow 0$ To enable the use of interrupts requires several steps: 1. Write an ISR 2. Install the ISR in memory at some arbitrary address X 3. Install the instruction "BUN X" at address 1 4. Enable interrupts with the ION instruction The sequence of events utilizing an interrupt to process keyboard input is as follows: 1. A character is typed 2. $FGI \leftarrow 1$ (same as with polling) 3. $R \leftarrow 1$, $IEN \leftarrow 0$ 4. $M[0] \leftarrow PC$ (store return address) 5. $PC \leftarrow 1$ (branch to interrupt vector) 6. BUN X (branch to ISR) 7. ISR checks FGI (found to be 1) 8. INP ($AC \leftarrow INPR$) 9. Character in AC is placed in a queue 10. ISR checks FGO (found to be 0) 11. ION 12. BUN 0 I Programs then read their input from a queue rather than directly from the input device. The ISR adds input to the queue as soon as it is typed, regardless of what code is running, and then returns to the running program.

Input-Output Interface Peripherals connected to a computer need special communication links for interfacing with CPU. In computer system, there are special hardware components between the CPU and peripherals to control or manage the input-output transfers. These components are called input-output interface units because they provide communication links between processor bus and peripherals. They provide a method for transferring information between internal system and input-output devices. Asynchronous Data Transfer We know that, the internal operations in individual unit of digital system are synchronized by means of clock pulse, means clock pulse is given to all registers within a unit, and all data transfer among internal registers occur simultaneously during

occurrence of clock pulse. Now, suppose any two units of digital system are designed independently such as CPU and I/O interface. And if the registers in the interface (I/O interface) share a common clock with CPU registers, then transfer between the two units is said to be synchronous. But in most cases, the internal timing in each unit is independent from each other in such a way that each uses its own private clock for its internal registers. In that case, the two units are said to be asynchronous to each other, and if data transfer occurs between them this data transfer is said to be Asynchronous Data Transfer. But, the Asynchronous Data Transfer between two independent units requires that control signals be transmitted between the communicating units so that the time can be indicated at which they send data.

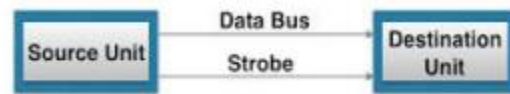
This asynchronous way of data transfer can be achieved by two methods: 1. One way is by means of a strobe pulse which is supplied by one of the units to the other unit. When transfer has to occur, this method is known as “Strobe Control”. 2. Another method commonly used is to accompany each data item being transferred with a control signal that indicates the presence of data in the bus. The unit receiving the data item responds with another signal to acknowledge receipt of the data. This method of data transfer between two independent units is said to be “Handshaking”. The strobe pulse and handshaking method of asynchronous data transfer are not restricted to I/O transfer. In fact, they are used extensively on numerous occasions requiring transfer of data between two independent units. So, here we consider the transmitting unit as source and receiving unit as destination. As an example: The CPU, is the source during an output or write transfer and is the destination unit during input or read transfer. And thus, the sequence of control during an asynchronous transfer depends on whether the transfer is initiated by the source or by the destination. So, while discussing each way of data transfer asynchronously we see the sequence of control in both terms when it is initiated by source or when it is initiated by destination. In this way, each way of data transfer, can be further divided into parts, source initiated and destination initiated. We can also specify, asynchronous transfer between two independent units by means of a timing diagram that shows the timing relationship that exists between the control and the data buses.

Now, we will discuss each method of asynchronous data transfer in detail one by one.

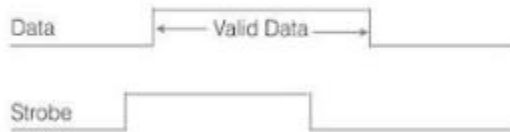
1. Strobe Control:

The Strobe Control method of asynchronous data transfer employs a single control line to time each transfer. This control line is also known as strobe and it may be achieved either by source or destination, depending on which initiates transfer. Source initiated strobe for data transfer:

The block diagram and timing diagram of strobe initiated by source unit is shown in figure below:

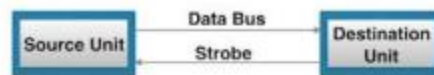


a) Block Diagram

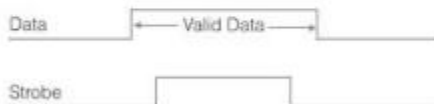


b) Timing Diagram

In block diagram we see that strobe is initiated by source, and as shown in timing diagram, the source unit first places the data on the data bus. After a brief delay to ensure that the data settle to a steady value, the source activates a strobe pulse. The information on data bus and strobe control signal remain in the active state for a sufficient period of time to allow the destination unit to receive the data. Actually, the destination unit, uses a falling edge of strobe control to transfer the contents of data bus to one of its internal registers. The source removes the data from the data bus after it disables its strobe pulse. New valid data will be available only after the strobe is enabled again. Destination-initiated strobe for data transfer: The block diagram and timing diagram of strobe initiated by destination is shown in figure below:



a) Block Diagram

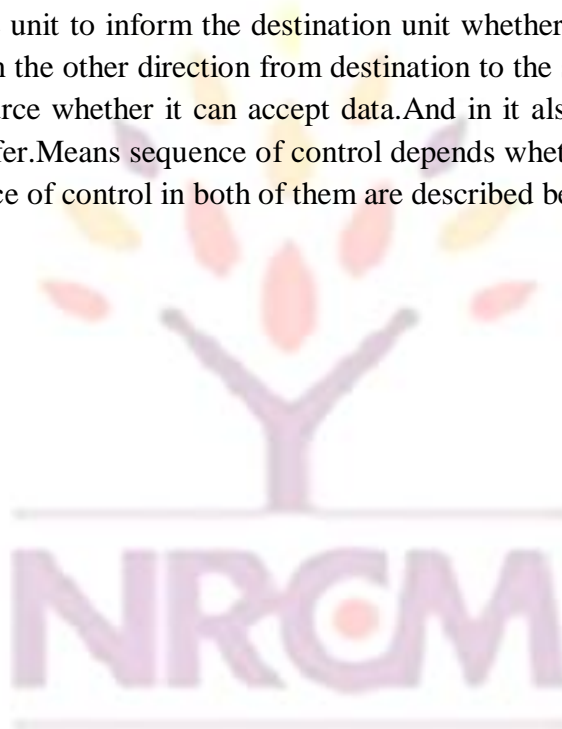


b) Timing Diagram

In block diagram, we see that, the strobe initiated by destination, and as shown in timing diagram, the destination unit first activates the strobe pulse, informing the source to provide the data. The source unit responds by placing the requested binary information on the data bus. The data must be valid and remain in the bus long enough for the destination unit to accept it. The falling edge of strobe pulse can be used again to trigger a destination register. The destination unit then disables the strobe. And source removes the data from data bus after a per determine time interval. Now, actually in computer, in the first case means in strobe initiated by source - the strobe may be a memory-write control signal from the CPU to a memory unit. The source, CPU, places the word on the data bus and

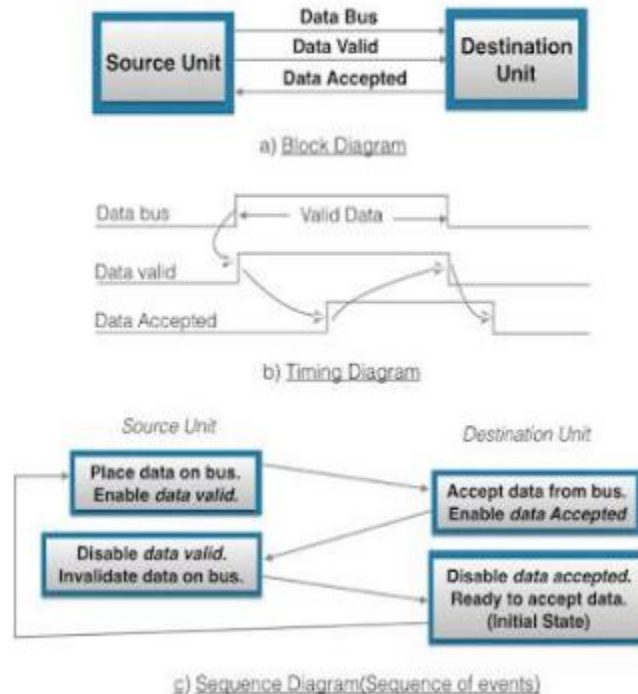
informs the memory unit, which is the destination, that this is a write operation. And in the second case i.e, in the strobe initiated by destination - the strobe may be a memory read control from the CPU to a memory unit. The destination, the CPU, initiates the read operation to inform the memory, which is a source unit, to place selected word into the data bus.

2. Handshaking: The disadvantage of strobe method is that source unit that initiates the transfer has no way of knowing whether the destination has actually received the data that was placed in the bus. Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit, has actually placed data on the bus. This problem can be solved by handshaking method. Hand shaking method introduce a second control signal line that provides a replay to the unit that initiates the transfer. In it, one control line is in the same direction as the data flow in the bus from the source to destination. It is used by source unit to inform the destination unit whether there are valid data in the bus. The other control line is in the other direction from destination to the source. It is used by the destination unit to inform the source whether it can accept data. And in it also, sequence of control depends on unit that initiate transfer. Means sequence of control depends whether transfer is initiated by source and destination. Sequence of control in both of them are described below:



Source initiated Handshaking:

The source initiated transfer using handshaking lines is shown in figure below:



In its block diagram, we see that two handshaking lines are "data valid", which is generated by the source unit, and "data accepted", generated by the destination unit. The timing diagram shows the timing relationship of exchange of signals between the two units. Means as shown in its timing diagram, the source initiates a transfer by placing data on the bus and enabling its data valid signal. The data accepted signal is then activated by destination unit after it accepts the data from the bus. The source unit then disables its data valid signal which invalidates the data on the bus. After this, the destination unit disables its data accepted signal and the system goes into initial state. The source unit does not send the next data item until after the destination unit shows its readiness to accept new data by disabling the data accepted signal. This sequence of events described in its sequence diagram, which shows the above sequence in which the system is present, at any given time.

Modes of I/O Data Transfer

Data transfer between the central unit and I/O devices can be handled in generally three types of modes which are given below:

1. Programmed I/O

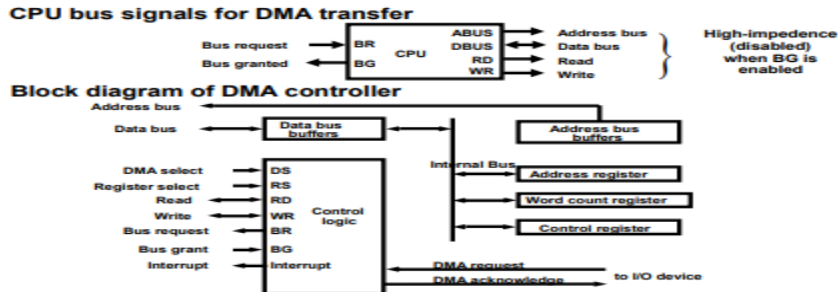
2. Interrupt Initiated I/O

3. Direct Memory Access

Programmed I/O Programmed I/O instructions are the result of I/O instructions written in computer program. Each data item transfer is initiated by the instruction in the program. Usually the program controls data transfer to and from CPU and peripheral. Transferring data under programmed I/O requires constant monitoring of the peripherals by the CPU. Interrupt Initiated I/O In the programmed I/O method the CPU stays in the program loop until the I/O unit indicates that it is ready for data transfer. This is time consuming process because it keeps the processor busy needlessly. This problem can be overcome by using interrupt initiated I/O. In this when the interface determines that the peripheral is ready for data transfer, it generates an interrupt. After receiving the interrupt signal, the CPU stops the task which it is processing and service the I/O transfer and then returns back to its previous processing task. Direct Memory Access Removing the CPU from the path and letting the peripheral device manage the memory buses directly would improve the speed of transfer. This technique is known as DMA. In this, the interface transfer data to and from the memory through memory bus. A DMA controller manages to transfer data between peripherals and memory unit. Many hardware systems use DMA such as disk drive controllers, graphic cards, network cards and sound cards etc. It is also used for intra chip data transfer in multicore processors. In DMA, CPU would initiate the transfer, do other operations while the transfer is in progress and receive an interrupt from the DMA controller when the transfer has been completed. Priority Interrupt A priority interrupt is a system which decides the priority at which various devices, which generates the interrupt signal at the same time, will be serviced by the CPU. The system has authority to decide which conditions are allowed to interrupt the CPU, while some other interrupt is being serviced. Generally, devices with high speed transfer such as magnetic disks are given high priority and slow devices such as keyboards are given low priority. When two or more devices interrupt the computer simultaneously, the computer services the device with the higher priority first.

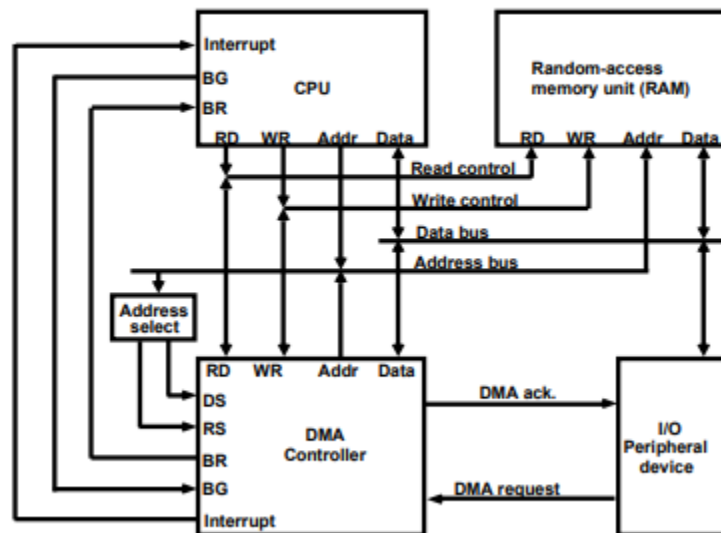
DIRECT MEMORY ACCESS

Block of data transfer from high speed devices, Drum, Disk, Tape



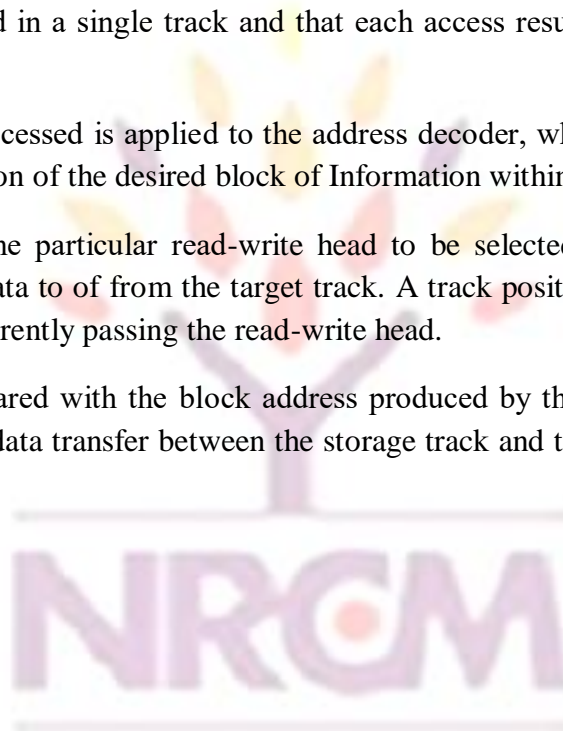
* DMA controller - Interface which allows I/O transfer directly between Memory and Device, freeing CPU for other tasks

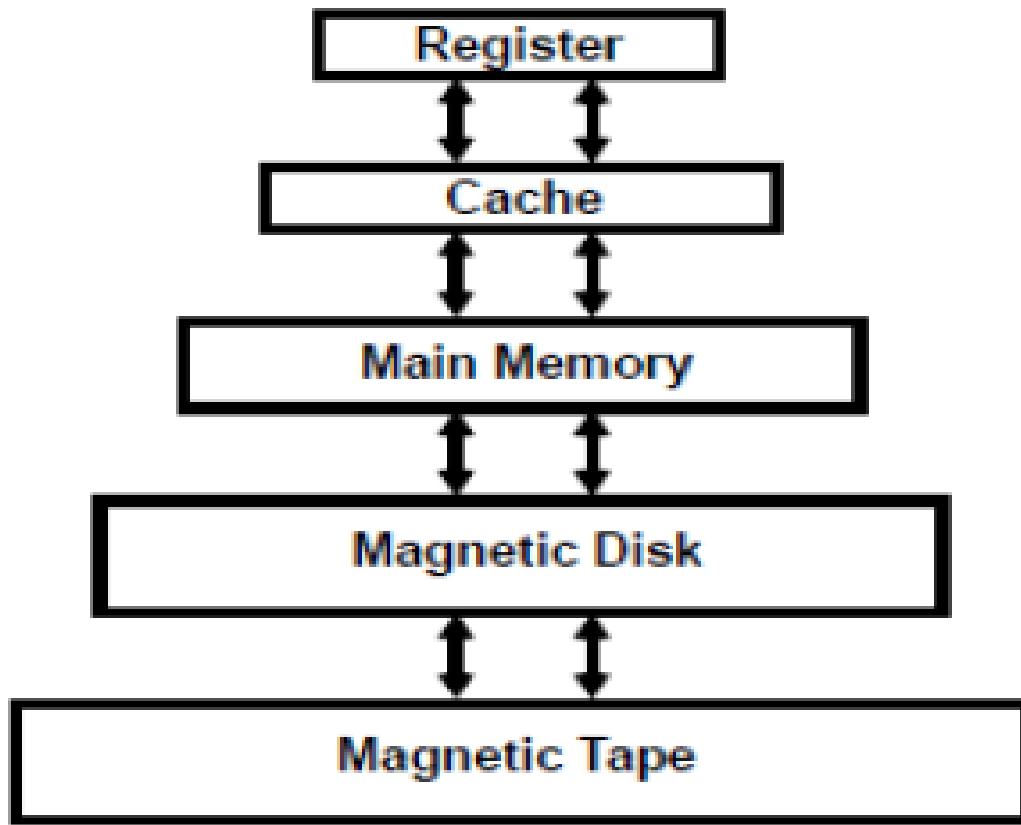
* CPU initializes DMA Controller by sending memory address and the block size(number of words)

DMA TRANSFER**MEMORY ORGANIZATION**

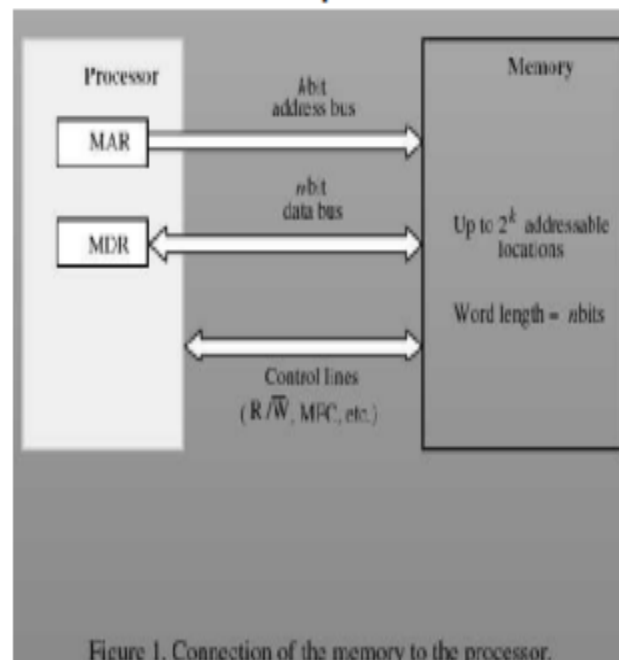
- RAM composed of a large number of (2M) of addressable locations, each of which stores a w-bit word.
- RAM operates as follows: first the address of the target location to be accessed is transferred via the address bus to the RAM's address buffer.

- The address is then processed by the address decoder, which selects the required location in the storage cell unit.
- If a read operation is requested, the contents of the addressed location are transferred from the storage cell unit to the data buffer and from there to the data bus.
- If a write operation is requested, the word to be stored is transferred from the data bus to the selected location in the storage unit. The storage unit is made up of many identical 1-bit memory cells and their interconnections. In each line connected to the storage cell unit, we can expect to
 - find a driver that acts as either an amplifier or a transducer of physical signals.
- assume that each word is stored in a single track and that each access results in the transfer of a block of words.
- The address of the data to be accessed is applied to the address decoder, whose output determines the track to be used and the location of the desired block of information within the track.
- the track address determines the particular read-write head to be selected. The selected head is moved into position to transfer data to or from the target track. A track position indicator generates the address of the block that is currently passing the read-write head.
- The generated address is compared with the block address produced by the address decoder. The selected head is enabled and the data transfer between the storage track and the memory data buffer register begins.





- The read-write head is disabled when a complete block information has been transferred



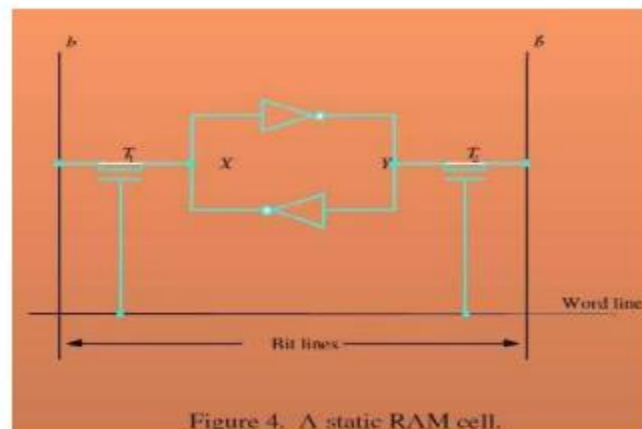


Figure 4. A static RAM cell.

Static memories (RAM)

Circuits capable of retaining their state as long as power is applied

Static RAM(SRAM)

volatile

DRAMs:

Charge on a capacitor

Needs —Refreshing



Synchronous DRAMs

Synchronized with a clock signal

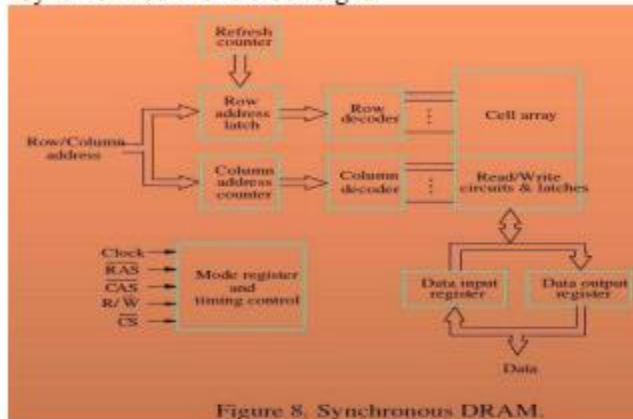


Figure 8. Synchronous DRAM.

Memory system considerations

- Cost
- Speed
- Power dissipation
- Size of chip



Principle of locality:

Temporal locality (locality in time): If an item is referenced, it will tend to be referenced again soon.

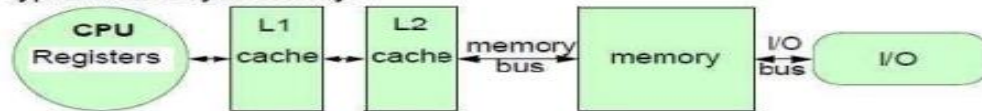
Spatial locality (locality in space): If an item is referenced, items whose addresses are close by will tend to be referenced soon.

Sequentiality (subset of spatial locality).

The principle of locality can be exploited implementing the memory of computer as a *memory hierarchy*, taking advantage of all types of memories.

Method: The level closer to processor (the fastest) is a subset of any level further away, and all the data is stored at the lowest level (the slowest).

Typical memory hierarchy:



Level	1	2	3	4
Named as	Registers	Cache	memory	disk storage
Typical size	<1 KB	< 4 MB	<2 GB	>2GB
Access time (ns)	2 - 5	3 - 10	80 - 400	5'000'000
Bandwidth(MB/sec)	4000 - 32'000	800 - 5000	400 - 2000	4 - 32
Managed by	Compiler	Hardware	Operating system	Operating system / user

Cache Memories

- Speed of the main memory is very low in comparison with the speed of processor – For good performance, the processor cannot spend much time of its time waiting to access instructions and data in main memory. – Important to devise a scheme that reduces the time to access the information – An efficient solution is to use fast cache memory When a cache is full and a memory word 101 that is not in the cache is referenced, the cache control hardware must decide which block should be removed to create space for the new block that contains the referenced word.

The basics of Caches

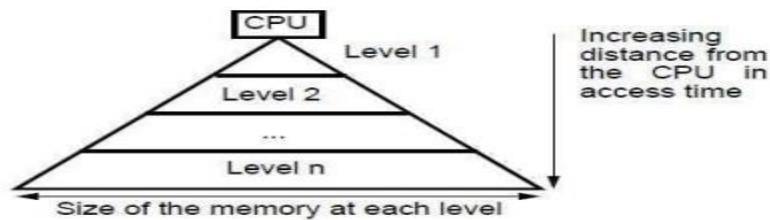
" The caches are organized on basis of *blocks*, the smallest amount of data which can be copied between two adjacent levels at a time.

" If data requested by the processor is present in some block in the upper level, it is called a *hit*.

" If data is not found in the upper level, the request is called a *miss* and the data is retrieved from the lower level in the hierarchy.

" The fraction of memory accesses found in the upper level is called a *hit ratio*.

" The storage, which takes advantage of locality of accesses is called a *cache*



Amdahl's Law about overall speedup:

$$\text{Speedup} = \frac{1}{(1 - \text{fraction of time cache can be used}) + \frac{\text{fraction of time cache can be used}}{\text{Speedup using cache}}}$$

Alternatively, CPU stalls can be considered¹:

CPU execution time = (CPU clock cycles + Memory stall cycles) × Clock cycle

The number of memory stall cycles depends:

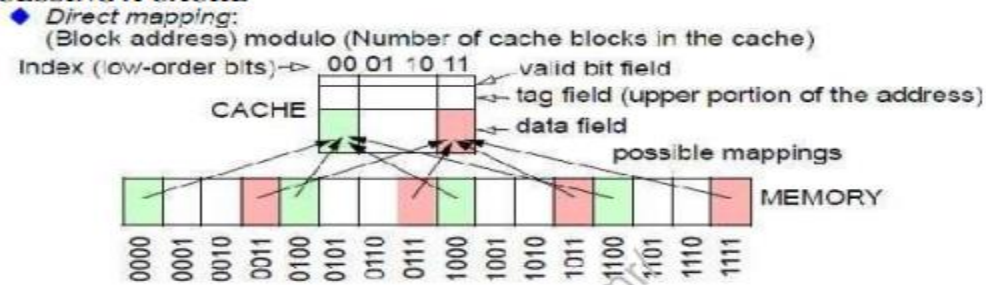
Memory stall cycles = IC × Memory references per instruction × Miss rate × Miss penalty

Size	Instruction cache	Data cache	
1 KB	3.06%	24.61%	13.34%
4 KB	1.76%	15.94%	7.24%
16 KB	0.64%	6.47%	2.87%
64 KB	0.15%	3.77%	1.35%

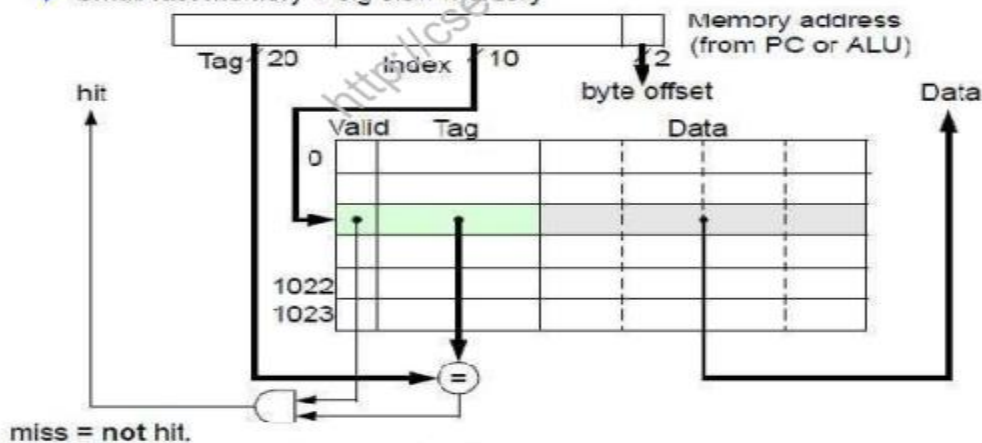
Table: Direct mapped cache, 32-byte blocks, SPEC92, DECstation 5000.

1. Here is assumed that CPU clock cycles include the time to handle a cache hit, and the CPU is stalled during a cache miss.



ACCESSING A CACHE

The *valid bit* indicates whether an entry contains a valid address. Initially, all valid bits are reset ("0" - not valid).

◆ **Small fast memory + big slow memory**

Virtual memory It is a computer system technique which gives an application program the impression that it has contiguous working memory (an address space), while in fact it may be physically fragmented and may even overflow on to disk storage. Virtual memory provides two primary functions: 1. Each process has its own address space, thereby not required to be relocated nor required to use relative addressing mode. 2. Each process sees one contiguous block of free memory upon launch. Fragmentation is hidden.

Auxiliary Memory

Devices that provide backup storage are called auxiliary memory. For example: Magnetic disks and tapes are commonly used auxiliary devices. Other devices used as auxiliary memory are magnetic drums, magnetic bubble memory and optical disks.

It is not directly accessible to the CPU, and is accessed using the Input/Output channels.

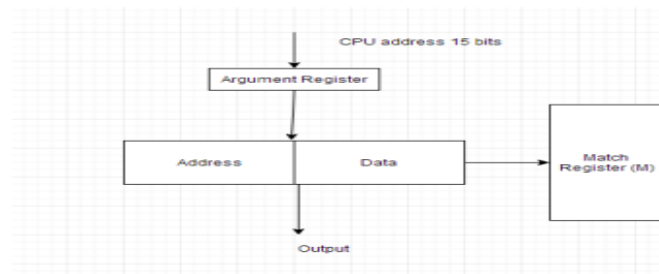
Cache Memory

The data or contents of the main memory that are used again and again by CPU, are stored in the cache memory so that we can easily access that data in shorter time.

Whenever the CPU needs to access memory, it first checks the cache memory. If the data is not found in cache memory then the CPU moves onto the main memory. It also transfers block of recent data into the cache and keeps on deleting the old data in cache to accommodate the new one.

Memory Mapping and Concept of Virtual Memory:

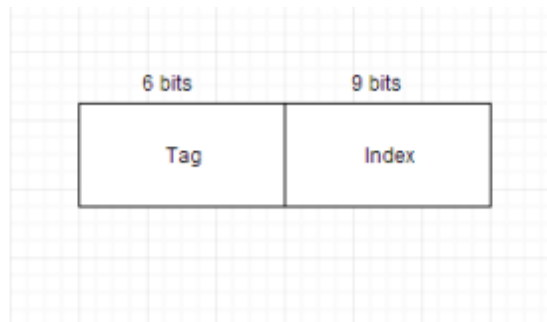
The transformation of data from main memory to cache memory is called mapping. There are 3 main types of mapping: Associative Mapping • Direct Mapping • Set Associative Mapping • Associative Mapping The associative memory stores both address and data. The address value of 15 bits is 5 digit octal numbers and data is of 12 bits word in 4 digit octal number. A CPU address of 15 bits is placed in argument register and the associative memory is searched for matching address.



Direct Mapping

The CPU address of 15 bits is divided into 2 fields. In this the 9 least significant bits constitute the **index** field and the remaining 6 bits constitute the **tag** field. The number of bits in index field is equal to the number of address bits required to access cache memory.

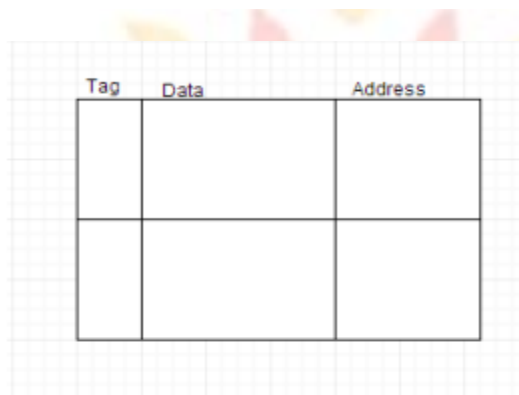




Set Associative Mapping

The disadvantage of direct mapping is that two words with same index address can't reside in cache memory at the same time. This problem can be overcome by set associative mapping.

In this we can store two or more words of memory under the same index address. Each data word is stored together with its tag and this forms a set.



Replacement Algorithms

Data is continuously replaced with new data in the cache memory using replacement algorithms. Following are the 2 replacement algorithms used:

- FIFO - First in First out. Oldest item is replaced with the latest item.
- LRU - Least Recently Used. Item which is least recently used by CPU is removed.

Writing in to cache and cache Initialization:

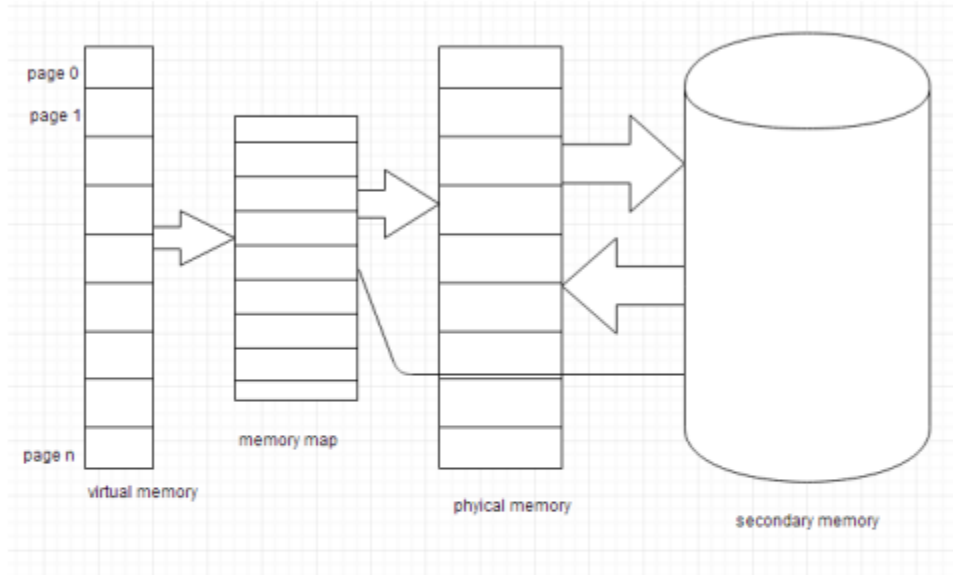
The benefit of write-through to main memory is that it simplifies the design of the computer system. With write-through, the main memory always has an up-to-date copy of the line. So when a read is done, main memory can always reply with the requested data.

If write-back is used, sometimes the up-to-date data is in a processor cache, and sometimes it is in main memory. If the data is in a processor cache, then that processor must stop main memory from replying to the read request, because the main memory might have a stale copy of the data. This is more complicated than write-through.

Virtual Memory:

Virtual memory is the separation of logical memory from physical memory. This separation provides large virtual memory for programmers when only small physical memory is available.

Virtual memory is used to give programmers the illusion that they have a very large memory even though the computer has a small main memory. It makes the task of programming easier because the programmer no longer needs to worry about the amount of physical memory available.

**Address mapping using pages:**

- The table implementation of the address mapping is simplified if the information in the address space. And the memory space is each divided into groups of fixed size.
- Moreover, The physical memory is broken down into groups of equal size called blocks, which may range from 64 to 4096 words each.
- The term page refers to groups of address space of the same size.
- Also, Consider a computer with an address space of 8K and a memory space of 4K.
- If we split each into groups of 1K words we obtain eight pages and four blocks as shown in the figure.

UNIT 5:

Reduced Instruction Set Computer: CISC Characteristics, RISC Characteristics.

Pipeline and Vector Processing: Parallel Processing, Pipelining, Arithmetic Pipeline, Instruction Pipeline, RISC Pipeline, Vector Processing, Array Processor.

MultiProcessors: Characteristics of Multiprocessors, Interconnection Structures, Interprocessor arbitration, Interprocessor communication and synchronization, cache Coherence.

Reduced Instruction Set Computer:

Reduced Instruction Set Architecture (RISC) –

The main idea behind this is to make hardware simpler by using an instruction set composed of a few basic steps for loading, evaluating, and storing operations just like a load command will load data, a store command will store the data.

Complex Instruction Set Architecture (CISC) –

The main idea is that a single instruction will do all loading, evaluating, and storing operations just like a multiplication command will do stuff like loading data, evaluating, and storing it, hence it's complex.

Earlier when programming was done using assembly language, a need was felt to make instruction do more tasks because programming in assembly was tedious and error-prone due to which CISC architecture evolved but with the uprise of high-level language dependency on assembly reduced RISC architecture prevailed.

Characteristic of RISC –

1. Simpler instruction, hence simple instruction decoding.
2. Instruction comes undersize of one word.
3. Instruction takes a single clock cycle to get executed.
4. More general-purpose registers.
5. Simple Addressing Modes.
6. Fewer Data types.
7. A pipeline can be achieved.

Characteristic of CISC –

1. Complex instruction, hence complex instruction decoding.
2. Instructions are larger than one-word size.
3. Instruction may take more than a single clock cycle to get executed.
4. Less number of general-purpose registers as operations get performed in memory itself.
5. Complex Addressing Modes.
6. More Data types.

Example – Suppose we have to add two 8-bit numbers:

- **CISC approach:** There will be a single command or instruction for this like ADD which will perform the task.
- **RISC approach:** Here programmer will write the first load command to load data in registers then it will use a suitable operator and then it will store the result in the desired location.

So, add operation is divided into parts i.e. load, operate, store due to which RISC programs are longer and require more memory to get stored but require fewer transistors due to less complex command.

Difference – RISC and CISC

RISC	CISC
Focus on software	Focus on hardware

RISC	CISC
Uses only Hardwired control unit	Uses both hardwired and microprogrammed control unit
Transistors are used for more registers	Transistors are used for storing complex Instructions
Fixed sized instructions	Variable sized instructions
Can perform only Register to Register Arithmetic operations	Can perform REG to REG or REG to MEM or MEM to MEM
Requires more number of registers	Requires less number of registers
Code size is large	Code size is small
An instruction executed in a single clock cycle	Instruction takes more than one clock cycle
An instruction fit in one word	Instructions are larger than the size of one word

Both approaches try to increase the CPU performance

- **RISC:** Reduce the cycles per instruction at the cost of the number of instructions per program.
- **CISC:** The CISC approach attempts to minimize the number of instructions per program but at the cost of an increase in the number of cycles per instruction.

Parallel processing

Execution of Concurrent Events in the computing process to achieve faster Computational Speed

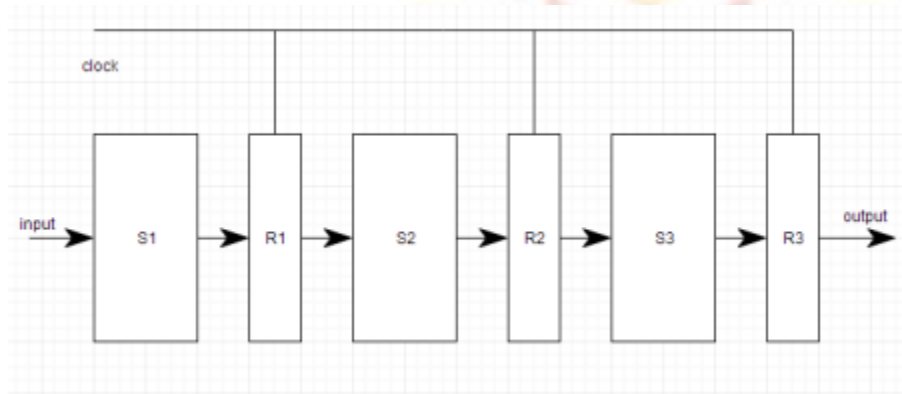
Levels of Parallel Processing

- Job or Program level
- Task or Procedure level
- Inter-Instruction level

- Intra-Instruction level

PARALLEL COMPUTERS Architectural Classification Flynn's classification » Based on the multiplicity of Instruction Streams and Data Streams » Instruction Stream Sequence of Instructions read from memory » Data Stream Operations performed on the data in the processor.

What is Pipelining? Pipelining is the process of accumulating instruction from the processor through a pipeline. It allows storing and executing instructions in an orderly process. It is also known as pipeline processing. Pipelining is a technique where multiple instructions are overlapped during execution. Pipeline is divided into stages and these stages are connected with one another to form a pipe like structure. Instructions enter from one end and exit from another end. Pipelining increases the overall instruction throughput. In pipeline system, each segment consists of an input register followed by a combinational circuit. The register is used to hold data and combinational circuit performs operations on it. The output of combinational circuit is applied to the input register of the next segment



Pipeline system is like the modern day assembly line setup in factories. For example in a car manufacturing industry, huge assembly lines are setup and at each point, there are robotic arms to perform a certain task, and then the car moves on ahead to the next arm. Types of Pipeline It is divided into 2 categories: 1. Arithmetic Pipeline 2. Instruction Pipeline

Arithmetic Pipeline Arithmetic pipelines are usually found in most of the computers. They are used for floating point operations, multiplication of fixed point numbers etc. For example: The input to the Floating Point Adder pipeline is: $X = A \cdot 2^a$ $Y = B \cdot 2^b$ Here A and B are mantissas (significant digit of floating point numbers), while a and b are exponents. The floating point addition and subtraction is done in 4 parts: 1. Compare the exponents. 2. Align the mantissas. 3. Add or subtract mantissas 4. Produce the result. Registers are used for storing the intermediate results between the above operations.

Instruction Pipeline In this a stream of instructions can be executed by overlapping fetch, decode and execute phases of an instruction cycle. This type of technique is used to increase the throughput of the computer system. An instruction pipeline reads instruction from the memory while previous instructions are being executed in other segments of the pipeline. Thus we can execute multiple instructions simultaneously. The pipeline will be more efficient if the instruction cycle is divided into segments of equal duration. Advantages of Pipelining 1. The cycle time of the processor is reduced. 2. It increases the throughput of the system 3. It makes the system reliable. Disadvantages of Pipelining 1. The design of pipelined processor is complex and costly to manufacture. 2. The instruction latency is more.

Vector(Array) Processing There is a class of computational problems that are beyond the capabilities of a conventional computer. These problems require vast number of computations on multiple data items, that will take a conventional computer(with scalar processor) days or even weeks to complete. Such complex instructions, which operates on multiple data at the same time, requires a better way of instruction execution, which was achieved by Vector processors. Scalar CPUs can manipulate one or two data items at a time, which is not very efficient. Also, simple instructions like ADD A to B, and store into C are not practically efficient. Addresses are used to point to the memory location where the data to be operated will be found, which leads to added overhead of data lookup. So until the data is found, the CPU would be sitting ideal, which is a big performance issue. Hence, the concept of Instruction Pipeline comes into picture, in which the instruction passes through several sub-units in turn.

These sub-units perform various independent functions, for example: the first one decodes the instruction, the second sub-unit fetches the data and the third sub-unit performs the math itself. Therefore, while the data is fetched for one instruction, CPU does not sit idle, it rather works on decoding the next instruction set, ending up working like an assembly line. Vector processor, not only use Instruction pipeline, but it also pipelines the data, working on multiple data at the same time. A normal scalar processor instruction would be ADD A, B, which leads to addition of two operands, but what if we can instruct the processor to ADD a group of numbers(from 0 to n memory location) to another group of numbers(lets say, n to k memory location). This can be achieved by vector processors. In vector processor a single instruction, can ask for multiple data operations, which saves time, as instruction is decoded once, and then it keeps on operating on different data items.

Applications of Vector Processors

Computer with vector processing capabilities are in demand in specialized applications. The following are some areas where vector processing is used:

1. Petroleum exploration.
2. Medical diagnosis.

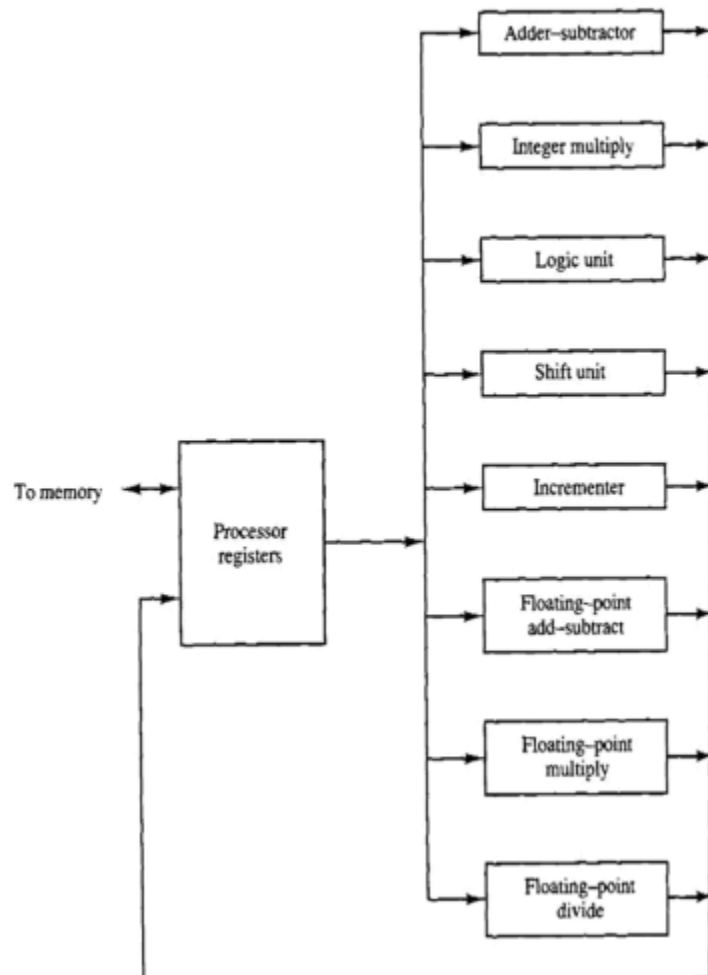
3. Data analysis.
4. Weather forecasting.
5. Aerodynamics and space flight simulations.
6. Image processing.

A parallel processing system is able to perform concurrent data processing to achieve faster execution time

- The system may have two or more ALUs and be able to execute two or more instructions at the same time
- Also, the system may have two or more processors operating concurrently
- Goal is to increase the throughput – the amount of processing that can be accomplished during a given interval of time
- Parallel processing increases the amount of hardware required
- Example: the ALU can be separated into three units and the operands diverted to each unit under the supervision of a control unit
- All units are independent of each other • A multifunctional organization is usually associated with a complex control unit to coordinate all the activities among the various components



Figure 9-1 Processor with multiple functional units.



- Parallel processing can be classified from:
 - o The internal organization of the processors
 - o The interconnection structure between processors
 - o The flow of information through the system
 - o The number of instructions and data items that are manipulated simultaneously
- The sequence of instructions read from memory is the instruction stream
- The operations performed on the data in the processor is the data stream

- Parallel processing may occur in the instruction stream, the data stream, or both Computer classification:

- o Single instruction stream, single data stream – SISD
- o Single instruction stream, multiple data stream – SIMD
- o Multiple instruction stream, single data stream – MISD
- o Multiple instruction stream, multiple data stream – MIMD

- SISD – Instructions are executed sequentially. Parallel processing may be achieved by means of multiple functional units or by pipeline processing

- SIMD – Includes multiple processing units with a single control unit. All processors receive the same instruction, but operate on different data.

- MIMD – A computer system capable of processing several programs at the same time.

- We will consider parallel processing under the following main topics:

PIPELINING

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

- Each segment performs partial processing dictated by the way the task is partitioned

- The result obtained from the computation in each segment is transferred to the next segment in the pipeline

- The final result is obtained after the data have passed through all segments

- Can imagine that each segment consists of an input register followed by an combinational circuit

- A clock is applied to all registers after enough time has elapsed to perform all segment activity

- The information flows through the pipeline one step at a time

- Example: $A_i * B_i + C_i$

for $i = 1, 2, 3, \dots, 7$

The suboperations performed in each segment are:

$$R1 \leftarrow A_i$$

$$R2 \leftarrow B_i$$

$$R3 \leftarrow R1 * R2, R4 \leftarrow C_i$$

$$R5 \leftarrow R3 + R4$$

Figure 9-2 Example of pipeline processing.

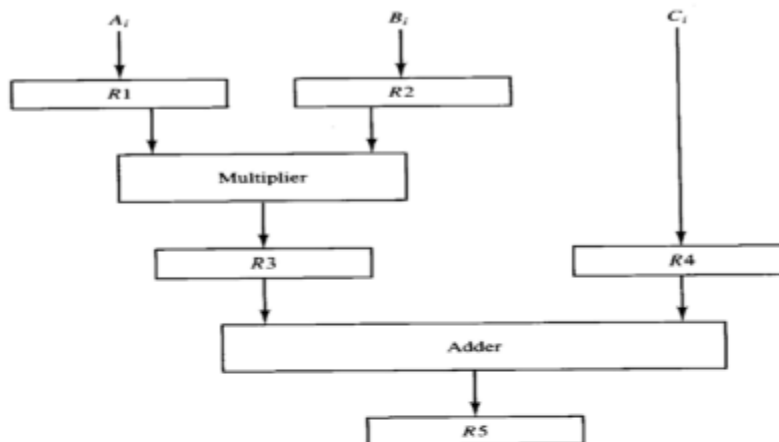


TABLE 9-1 Content of Registers in Pipeline Example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	$R1$	$R2$	$R3$	$R4$	$R5$
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

Any operation that can be decomposed into a sequence of suboperations of about the same complexity can be implemented by a pipeline processor

- The technique is efficient for those applications that need to repeat the same task many time with different sets of data
- A task is the total operation performed going through all segments of a pipeline
- The behavior of a pipeline can be illustrated with a space-time diagram

- This shows the segment utilization as a function of time
- Once the pipeline is full, it takes only one clock period to obtain an output

Figure 9-4 Space-time diagram for pipeline.

	1	2	3	4	5	6	7	8	9	
Segment: 1	T_1	T_2	T_3	T_4	T_5	T_6				→ Clock cycles
2		T_1	T_2	T_3	T_4	T_5	T_6			
3			T_1	T_2	T_3	T_4	T_5	T_6		
4				T_1	T_2	T_3	T_4	T_5	T_6	

Figure 9-2 Example of pipeline processing.

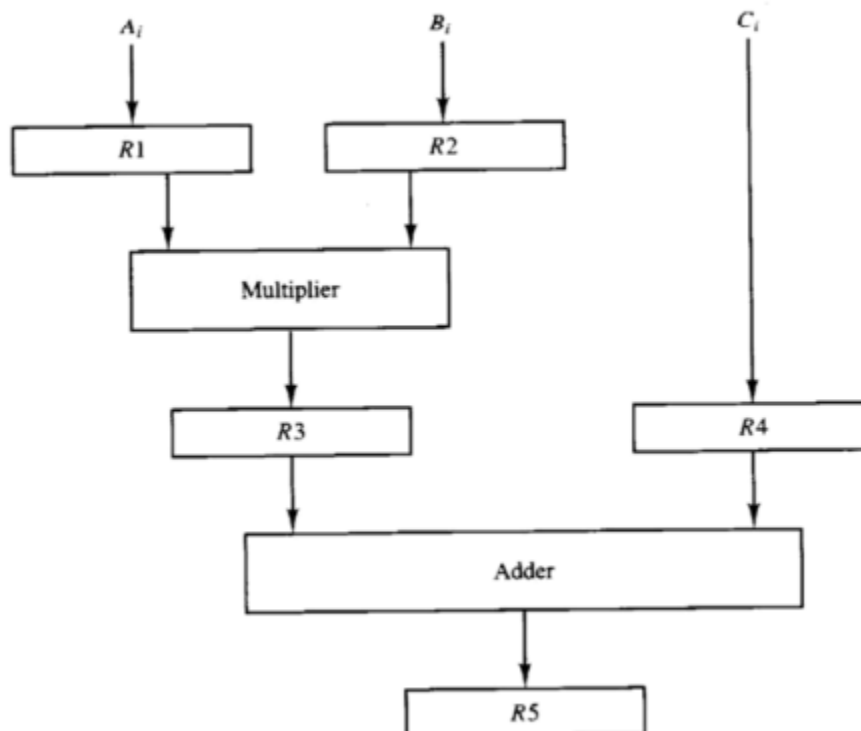


TABLE 9-1 Content of Registers in Pipeline Example

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A_1	B_1	—	—	—
2	A_2	B_2	$A_1 * B_1$	C_1	—
3	A_3	B_3	$A_2 * B_2$	C_2	$A_1 * B_1 + C_1$
4	A_4	B_4	$A_3 * B_3$	C_3	$A_2 * B_2 + C_2$
5	A_5	B_5	$A_4 * B_4$	C_4	$A_3 * B_3 + C_3$
6	A_6	B_6	$A_5 * B_5$	C_5	$A_4 * B_4 + C_4$
7	A_7	B_7	$A_6 * B_6$	C_6	$A_5 * B_5 + C_5$
8	—	—	$A_7 * B_7$	C_7	$A_6 * B_6 + C_6$
9	—	—	—	—	$A_7 * B_7 + C_7$

Arithmetic Pipeline

- Pipeline arithmetic units are usually found in very high speed computers
- They are used to implement floating-point operations, multiplication of fixed-point numbers, and similar computations encountered in scientific problems
- Example for floating-point addition and subtraction
- Inputs are two normalized floating-point binary numbers

$$X = A \times 2^a$$

$$Y = B \times 2^b$$

- A and B are two fractions that represent the mantissas
- a and b are the exponents

Instruction Pipeline

- An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments
- This causes the instruction fetch and execute phases to overlap and perform simultaneous operations

- If a branch out of sequence occurs, the pipeline must be emptied and all the instructions that have been read from memory after the branch instruction must be discarded
- Consider a computer with an instruction fetch unit and an instruction execution unit forming a two segment pipeline
- A FIFO buffer can be used for the fetch segment
- Thus, an instruction stream can be placed in a queue, waiting for decoding and processing by the execution segment
- This reduces the average access time to memory for reading instructions
- Whenever there is space in the buffer, the control unit initiates the next instruction fetch phase
- The following steps are needed to process each instruction:
 - o Fetch the instruction from memory
 - o Decode the instruction
 - o Calculate the effective address
 - o Fetch the operands from memory
 - o Execute the instruction
 - o Store the result in the proper place

Up to four suboperations in the instruction cycle can overlap and up to four different instructions can be in progress of being processed at the same time

- It is assumed that the processor has separate instruction and data memories
- Reasons for the pipeline to deviate from its normal operation are:
 - o Resource conflicts caused by access to memory by two segments at the same time.
 - o Data dependency conflicts arise when an instruction depends on the result of a previous instruction, but its result is not yet available.

Types of Multiprocessors

There are mainly two types of multiprocessors i.e. symmetric and asymmetric multiprocessors. Details about them are as follows –

Symmetric Multiprocessors

In these types of systems, each processor contains a similar copy of the operating system and they all communicate with each other. All the processors are in a peer to peer relationship i.e. no master - slave relationship exists between them.

An example of the symmetric multiprocessing system is the Encore version of Unix for the Multimax Computer.

Asymmetric Multiprocessors

In asymmetric systems, each processor is given a predefined task. There is a master processor that gives instruction to all the other processors. Asymmetric multiprocessor system contains a master slave relationship.

Asymmetric multiprocessor was the only type of multiprocessor available before symmetric multiprocessors were created. Now also, this is the cheaper option.

Advantages of Multiprocessor Systems

There are multiple advantages to multiprocessor systems. Some of these are –

More reliable Systems

In a multiprocessor system, even if one processor fails, the system will not halt. This ability to continue working despite hardware failure is known as graceful degradation. For example: If there are 5 processors in a multiprocessor system and one of them fails, then also 4 processors are still working. So the system only becomes slower and does not ground to a halt.

Enhanced Throughput

If multiple processors are working in tandem, then the throughput of the system increases i.e. number of processes getting executed per unit of time increase. If there are N processors then the throughput increases by an amount just under N.

More Economic Systems

Multiprocessor systems are cheaper than single processor systems in the long run because they share the data storage, peripheral devices, power supplies etc. If there are multiple processes that share data, it is better to schedule them on multiprocessor systems with shared data than have different computer systems with multiple copies of the data.

Disadvantages of Multiprocessor Systems

There are some disadvantages as well to multiprocessor systems. Some of these are:

Increased Expense

Even though multiprocessor systems are cheaper in the long run than using multiple computer systems, still they are quite expensive. It is much cheaper to buy a simple single processor system than a multiprocessor system.

Complicated Operating System Required

There are multiple processors in a multiprocessor system that share peripherals, memory etc

Characteristics of Multiprocessor

There are the major characteristics of multiprocessors are as follows –

- **Parallel Computing** – This involves the simultaneous application of multiple processors. These processors are developed using a single architecture to execute a common task. In general, processors are identical and they work together in such a way that the users are under the impression that they are the only users of the system. In reality, however, many users are accessing the system at a given time.
- **Distributed Computing** – This involves the usage of a network of processors. Each processor in this network can be considered as a computer in its own right and have the capability to solve a problem. These processors are heterogeneous, and generally, one task is allocated to a single processor.
- **Supercomputing** – This involves the usage of the fastest machines to resolve big and computationally complex problems. In the past, supercomputing machines were vector computers but at present, vector or parallel computing is accepted by most people.
- **Pipelining** – This is a method wherein a specific task is divided into several subtasks that must be performed in a sequence. The functional units help in performing each subtask. The units are attached serially and all the units work simultaneously.
- **Vector Computing** – It involves the usage of vector processors, wherein operations such as ‘multiplication’ are divided into many steps and are then applied to a stream of operands (“vectors”).
- **Systolic** – This is similar to pipelining, but units are not arranged in a linear order. The steps in systolic are normally small and more in number and performed in a lockstep manner. This is more frequently applied in special-purpose hardware such as image or signal processors.

Interconnection structures :

The processors must be able to share a set of main memory modules & I/O devices in a multiprocessor system. This sharing capability can be provided through interconnection structures. The interconnection structure that are commonly used can be given as follows –

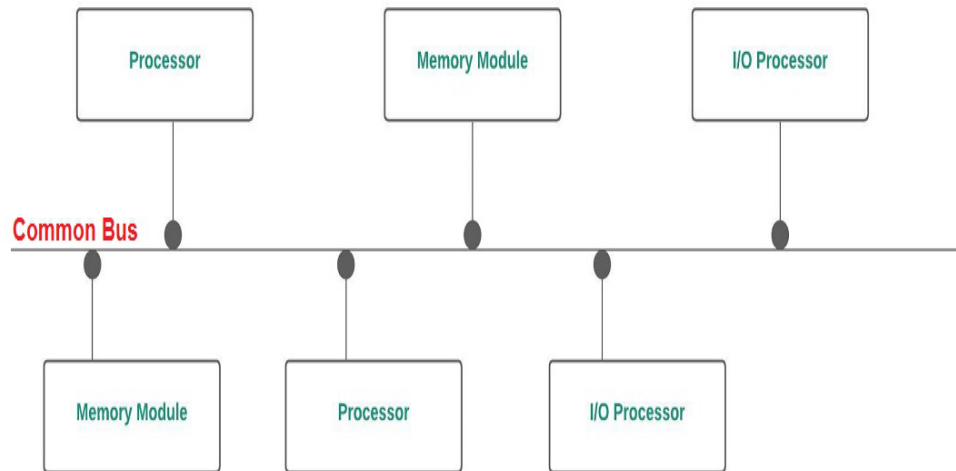
1. Time-shared / Common Bus
2. [Cross bar Switch](#)
3. [Multiport Memory](#)
4. Multistage Switching Network (Covered in 2nd part)
5. [Hypercube System](#)

In this article, we will cover Time shared / Common Bus in detail.

1. Time-shared / Common Bus (Interconnection structure in Multiprocessor System) :

In a multiprocessor system, the time shared bus interconnection provides a common

communication path connecting all the functional units like processor, I/O processor, memory unit etc. The figure below shows the multiple processors with common communication path (single bus).



Single-Bus Multiprocessor Organization

To communicate with any functional unit, processor needs the bus to transfer the data. To do so, the processor first need to see that whether the bus is available / not by checking the status of the bus. If the bus is used by some other functional unit, the status is busy, else free.

A processor can use bus only when the bus is free. The sender processor puts the address of the destination on the bus & the destination unit identifies it. In order to communicate with any functional unit, a command is issued to tell that unit, what work is to be done. The other processors at that time will be either busy in internal operations or will sit free, waiting to get bus. We can use a bus controller to resolve conflicts, if any. (Bus controller can set priority of different functional units)

This Single-Bus Multiprocessor Organization is easiest to reconfigure & is simple. This interconnection structure contains only passive elements. The bus interfaces of sender & receiver units controls the transfer operation here.

To decide the access to common bus without conflicts, methods such as static & fixed priorities, First-In-Out (FIFO) queues & daisy chains can be used.

Advantages –

- Inexpensive as no extra hardware is required such as switch.
- Simple & easy to configure as the functional units are directly connected to the bus .

Disadvantages –

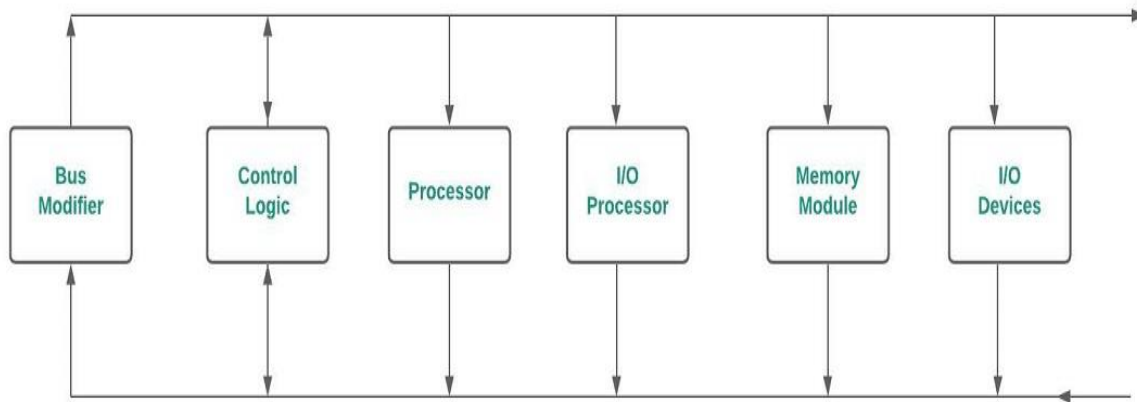
- Major fight with this kind of configuration is that if malfunctioning occurs in any of the bus interface circuits, complete system will fail.

- **Decreased throughput —**

At a time, only one processor can communicate with any other functional unit.

- Increased **arbitration logic** —

As the number of processors & memory unit increases, the bus contention problem increases. To solve the above disadvantages, we can use two uni-directional buses as :

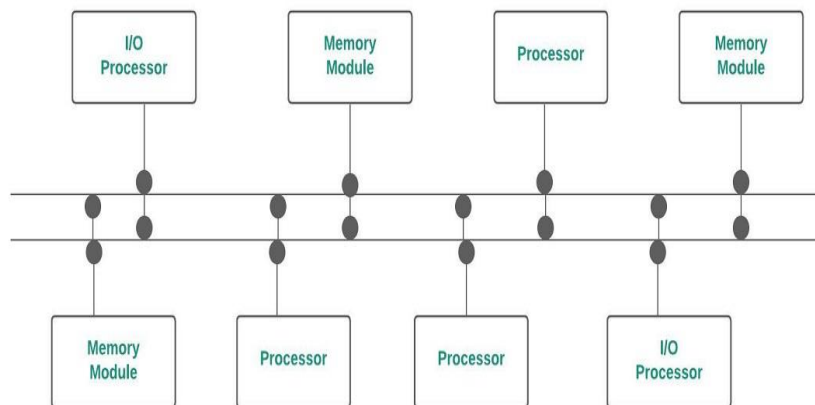


Multiprocessor System with unidirectional buses

Both the buses are required in a single transfer operation. Here, the system complexity is increased & the reliability is decreased, The solution is to use multiple bi-directional buses.

Multiple bi-directional buses :

The multiple bi-directional buses means that in the system there are multiple buses that are bi-directional. It permits simultaneous transfers as many as buses are available. But here also the complexity of the system is increased.



Multiple Bi-Directional Multiprocessor System

Apart from the organization, there are many factors affecting the performance of bus. They are –

- Number of active devices on the bus.
- Data width
- Error Detection method
- Synchronization of data transfer etc.

Advantages of Multiple bi-directional buses –

- Lowest cost for hardware as no extra device is needed such as switch.
- Modifying the hardware system configuration is easy.
- Less complex when compared to other interconnection schemes as there are only 2 buses & all the components are connected via that buses.

Disadvantages of Multiple bi-directional buses –

- System Expansion will degrade the performance because as the number of functional unit increases, more communication is required but at a time only 1 transfer can happen via 1 bus.
- Overall system capacity limits the transfer rate & If bus fails, whole system will fail.
- Suitable for small systems only.

2. Crossbar Switch :

A point is reached at which there is a separate path available for each memory module, if the number of buses in common bus system is increased. Crossbar Switch (for multiprocessors) provides separate path for each module.

3. Multiport Memory :

In Multiport Memory system, the control, switching & priority arbitration logic are distributed throughout the crossbar switch matrix which is distributed at the interfaces to the memory modules.

4. Hypercube Interconnection :

This is a binary n-cube architecture. Here we can connect 2^n processors and each of the processor here forms a node of the cube. A node can be memory module, I/O interface also, not necessarily processor. The processor at a node has communication path that is direct goes to n other nodes (total 2^n nodes). There are total 2^n distinct n-bit binary addresses.

Conclusion :

Interconnection structure can decide overall system's performance in a multi processor environment. Although using common bus system is much easy & simple, but the availability of only 1 path is its major drawback & if the bus fails, whole system fails. To overcome this & improve overall performance, crossbar, multi port, hypercube & then multistage switch network evolved.

Computer systems contain a number of buses at various levels to facilitate the transfer of information between components. The CPU contains a number of internal buses for transferring information between processor registers and ALU.

Inter Processor Arbitration

The processor, main memory and I/O devices can be interconnected by means of a common bus. A bus is set of lines (wires) defined 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 known as system bus. Different types of arbitration: *Serial (Daisy Chain) arbitration, Parallel arbitration, Dynamic arbitration*

Serial (Daisy Chain) arbitration

In this type of arbitration, processors can access bus based on priority. In serial arbitration, bus access priority resolving based on the serial connection of the processors. This technique is obtained from daisy chain (serial) connection of processors. The serial priority resolving technique is obtained from daisy-chain connection similar to the daisy chain priority interrupt logic. The processors connected to the system bus are assigned priority according to their position along the priority control line.

When multiple devices concurrently request the use of the bus, the device with the highest priority is granted access to it. Each processor has its own bus arbiter logic with priority-in and priority-out lines. The priority out (PO) of each arbiter is connected to the priority in (PI) of the next-lower-priority arbiter. The PI of the highest-priority unit is maintained at a logic value 1. The highest-priority unit in the system will always receive access to the system bus when it requests it. The processor whose arbiter has a $PI = 1$ and $PO = 0$. That processor accesses the system bus.

Advantages

Simple and cheaper method

Least number of lines.

Disadvantages

Higher delay

Priority of the processor is fixed

Not reliable

Inter Process Communication (IPC)

A process can be of two types:

- Independent process.
- Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently, in reality, there are many situations when co-operative nature can be utilized for increasing computational speed, convenience, and modularity. Inter-process communication (IPC) is a mechanism that allows processes to

communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other through both:

1. Shared Memory
2. Message passing

Figure 1 below shows a basic structure of communication between processes via the shared memory method and via the message passing method.

An operating system can implement both methods of communication. First, we will discuss the shared memory methods of communication and then message passing. Communication between processes using shared memory requires processes to share some variable, and it completely depends on how the programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously, and they share some resources or use some information from another process. Process1 generates information about certain computations or resources being used and keeps it as a record in shared memory. When process2 needs to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from another process as well as for delivering any specific information to other processes.

Let's discuss an example of communication between processes using the shared memory method.

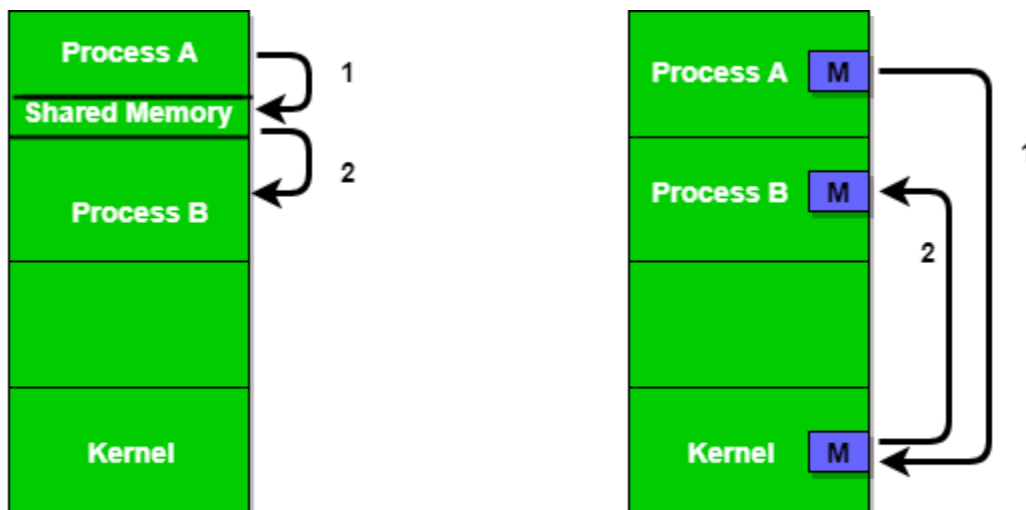


Figure 1 - Shared Memory and Message Passing

i) Shared Memory Method

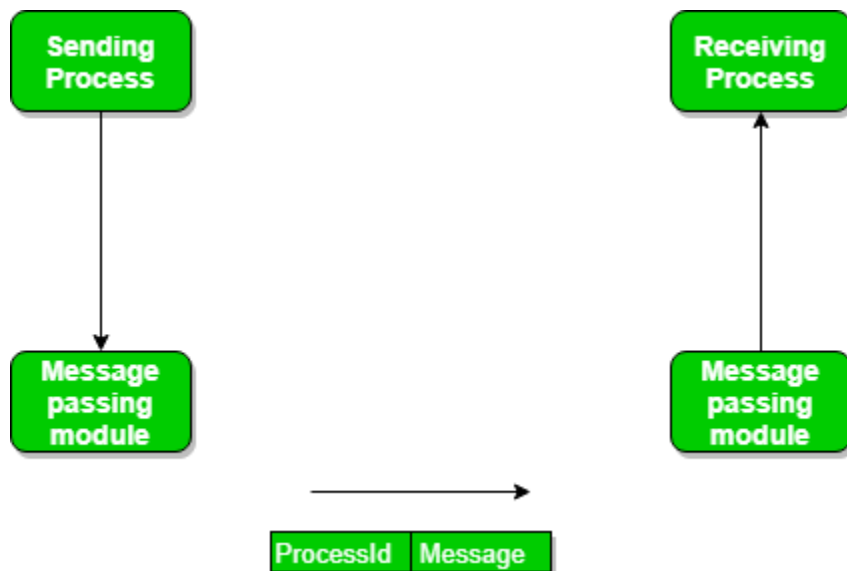
Ex: Producer-Consumer problem

There are two processes: Producer and Consumer. The producer produces some items and the Consumer consumes that item. The two processes share a common space or memory location known as a buffer where the item produced by the Producer is stored and from which the Consumer consumes the item if needed. There are two versions of this problem: the first one is known as the unbounded buffer problem in which the Producer can keep on producing items and there is no limit on the size of the buffer, the second one is known as the bounded buffer problem in which the Producer can produce up to a certain number of items before it starts waiting for Consumer to consume it. We will discuss the bounded buffer problem. First, the Producer and the Consumer will share some common memory, then the producer will start producing items. If the total produced item is equal to the size of the buffer, the producer will wait to get it consumed by the Consumer. Similarly, the consumer will first check for the availability of the item. If no item is available, the Consumer will wait for the Producer to produce it. If there are items available, Consumer will consume them. The pseudo-code to demonstrate is provided below:

Shared Data between the two Processes**ii) Messaging Passing Method**

Now, We will start our discussion of the communication between processes via message passing. In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follows:

- Establish a communication link (if a link already exists, no need to establish it again.)
- Start exchanging messages using basic primitives.
 - **send**(message, destination) or **send**(message)
 - **receive**(message, host) or **receive**(message)



The message size can be of fixed size or of variable size. If it is of fixed size, it is easy for an OS designer but complicated for a programmer and if it is of variable size then it is easy for a programmer but complicated for the OS designer. A standard message can have two parts: **header and body**.

The **header part** is used for storing message type, destination id, source id, message length, and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.

Message Passing through Communication Link.

Direct and Indirect Communication link

Now, We will start our discussion about the methods of implementing communication links. While implementing the link, there are some questions that need to be kept in mind like :

1. How are links established?
2. Can a link be associated with more than two processes?
3. How many links can there be between every pair of communicating processes?
4. What is the capacity of a link? Is the size of a message that the link can accommodate fixed or variable?
5. Is a link unidirectional or bi-directional?

A link has some capacity that determines the number of messages that can reside in it temporarily for which every link has a queue associated with it which can be of zero capacity, bounded capacity, or unbounded capacity. In zero capacity, the sender waits until the receiver informs the sender that it has received the message. In non-zero capacity cases, a process does not know whether a message has been received or not after the send operation. For this, the sender must communicate with the receiver explicitly. Implementation of the link depends on the situation, it can be either a direct communication link or an in-directed communication link.

Direct Communication links are implemented when the processes use a specific process identifier for the communication, but it is hard to identify the sender ahead of time.

For example the print server.

In-direct Communication is done via a shared mailbox (port), which consists of a queue of messages. The sender keeps the message in mailbox and the receiver picks them up.

Message Passing through Exchanging the Messages.

Synchronous and Asynchronous Message Passing:

A process that is blocked is one that is waiting for some event, such as a resource becoming available or the completion of an I/O operation. IPC is possible between the processes on same computer as well as on the processes running on different computer i.e. in networked/distributed system. In both cases, the process may or may not be blocked while sending a message or attempting to receive a message so message passing may be blocking or non-blocking. Blocking is considered **synchronous** and **blocking send** means the sender will be blocked until the message is received by receiver. Similarly, **blocking receive** has the receiver block until a message is available. Non-blocking is considered **asynchronous** and Non-blocking send has the sender sends the message and continue. Similarly, Non-blocking receive has the receiver receive a valid message or null. After a careful analysis, we can come to a conclusion that for a sender it is more

natural to be non-blocking after message passing as there may be a need to send the message to different processes. However, the sender expects acknowledgment from the receiver in case the send fails. Similarly, it is more natural for a receiver to be blocking after issuing the receive as the information from the received message may be used for further execution. At the same time, if the message send keep on failing, the receiver will have to wait indefinitely. That is why we also consider the other possibility of message passing. There are basically three preferred combinations:

- Blocking send and blocking receive
- Non-blocking send and Non-blocking receive
- Non-blocking send and Blocking receive (Mostly used)

In Direct message passing, The process which wants to communicate must explicitly name the recipient or sender of the communication.

e.g. **send(p1, message)** means send the message to p1.

Similarly, **receive(p2, message)** means to receive the message from p2.

In this method of communication, the communication link gets established automatically, which can be either unidirectional or bidirectional, but one link can be used between one pair of the sender and receiver and one pair of sender and receiver should not possess more than one pair of links. Symmetry and asymmetry between sending and receiving can also be implemented i.e. either both processes will name each other for sending and receiving the messages or only the sender will name the receiver for sending the message and there is no need for the receiver for naming the sender for receiving the message. The problem with this method of communication is that if the name of one process changes, this method will not work.

In Indirect message passing, processes use mailboxes (also referred to as ports) for sending and receiving messages. Each mailbox has a unique id and processes can communicate only if they share a mailbox. Link established only if processes share a common mailbox and a single link can be associated with many processes. Each pair of processes can share several communication links and these links may be unidirectional or bi-directional. Suppose two processes want to communicate through Indirect message passing, the required operations are: create a mailbox, use this mailbox for sending and receiving messages, then destroy the mailbox. The standard primitives used are: **send(A, message)** which means send the message to mailbox A. The primitive for the receiving the message also works in the same way e.g. **received (A, message)**. There is a problem with this mailbox implementation. Suppose there are more than two processes sharing the same mailbox and suppose the process p1 sends a message to the mailbox, which process will be the receiver? This can be solved by either enforcing that only two processes can share a single mailbox or enforcing that only one process is allowed to execute the receive at a given time or select any process randomly and notify the sender about the receiver. A mailbox can be made private to a single sender/receiver pair and can also be shared between multiple sender/receiver pairs. Port is an implementation of such mailbox that can have multiple senders and a single receiver. It is used in client/server applications (in this case the server is the receiver). The port is owned by the receiving process and created by OS on the request of the receiver process and can be destroyed either on request of the same receiver processor when the receiver terminates itself. Enforcing that only one process is allowed to execute the receive can be done using the concept of mutual exclusion. **Mutex mailbox** is created which is shared by n process. The sender is non-

blocking and sends the message. The first process which executes the receive will enter in the critical section and all other processes will be blocking and will wait.

Now, let's discuss the Producer-Consumer problem using the message passing concept. The producer places items (inside messages) in the mailbox and the consumer can consume an item when at least one message present in the mailbox. The code is given below:

Examples of IPC systems

1. Posix : uses shared memory method.
2. Mach : uses message passing
3. Windows XP : uses message passing using local procedural calls

Communication in client/server Architecture:

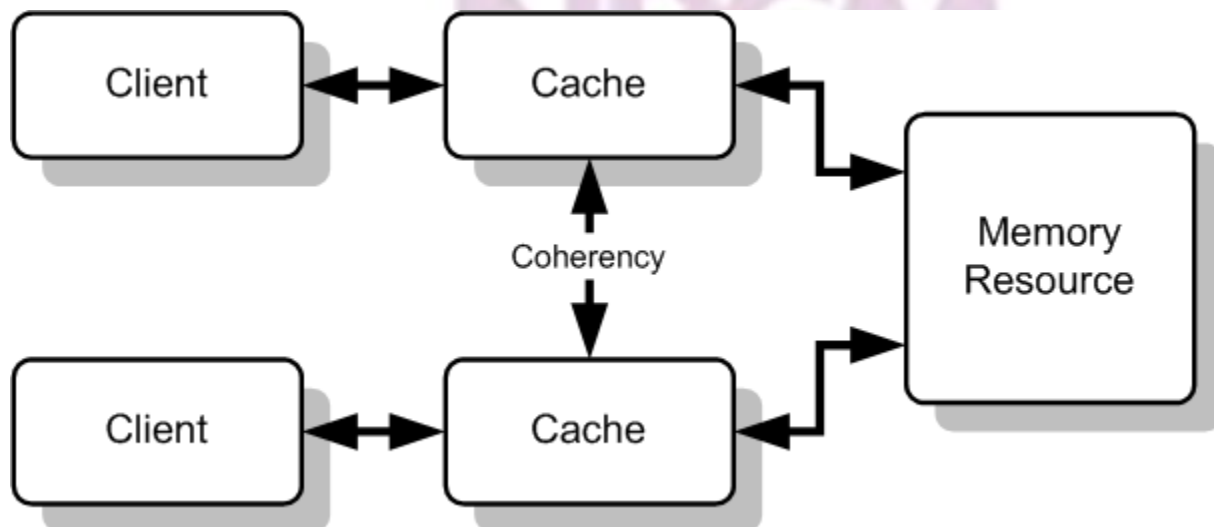
There are various mechanism:

- Pipe
- Socket
- Remote Procedural calls (RPCs)

Cache Coherence.

In [computer architecture](#), **cache coherence** is the uniformity of shared resource data that ends up stored in multiple [local caches](#). When clients in a system maintain [caches](#) of a common memory resource, problems may arise with incoherent data, which is particularly the case with [CPUs](#) in a [multiprocessing](#) system.

In the illustration on the right, consider both the clients have a cached copy of a particular memory block from a previous read. Suppose the client on the bottom updates/changes that memory block, the client on the top could be left with an invalid cache of memory without any notification of the change. Cache coherence is intended to manage such conflicts by maintaining a coherent view of the data values in multiple caches.





COMPUTER SYSTEM ARCHITECTURE

THIRD EDITION



**Dr. Chao Tan,
Carnegie Mellon University**

M. Morris Mano

Chap. 1: Digital Logic Circuits

- **Logic Gates, • Boolean Algebra**
- **Map Simplification, • Combinational Circuits**
- **Filp-Flops, • Sequential Circuits**

Chap. 2: Digital Components

- **Integrated Circuits, • Decoders, • Multiplexers**
- **Registers, • Shift Registers, • Binary Counters**
- **Memory Unit**

Chap. 3: Data Representation

- **Data Types, • Complements**
- **Fixed Point Representation**
- **Floating Point Representation**
- **Other Binary Codes, • Error Detection Codes**

Chap. 4: Register Transfer and Microoperations

- **Register Transfer Language, • Register Transfer**
- **Bus and Memory Transfers**
- **Arithmetic Microoperations**
- **Logic Microoperations, • Shift Microoperations**
- **Arithmetic Logic Shift Unit**

Chap. 5: Basic Computer Organization and Design

- **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 Logic**

Chap. 6: Programming the Basic Computer

- **Machine Language, • Assembly Language**
- **Assembler, • Program Loops**
- **Programming Arithmetic and Logic Operations**
- **Subroutines, • Input-Output Programming**

Chap. 7: Microprogrammed Control

- **Control Memory, • Sequencing Microinstructions**
- **Microprogram Example, • Design of Control Unit**
- **Microinstruction Format**

Chap. 8: Central Processing Unit

- **General Register Organization**
- **Stack Organization, • Instruction Formats**
- **Addressing Modes**
- **Data Transfer and Manipulation**
- **Program Control**
- **Reduced Instruction Set Computer**

Chap. 9: Pipeline and Vector Processing

- **Parallel Processing, • Pipelining**
- **Arithmetic Pipeline, • Instruction Pipeline**
- **RISC Pipeline, • Vector Processing**

Chap. 10: Computer Arithmetic

- **Arithmetic with Signed-2's Complement Numbers**
- **Multiplication and Division Algorithms**
- **Floating-Point Arithmetic Operations**
- **Decimal Arithmetic Unit**
- **Decimal Arithmetic Operations**

Chap. 11: Input-Output Organization

- **Peripheral Devices, • Input-Output Interface**
- **Asynchronous Data Transfer, • Modes of Transfer**
- **Priority Interrupt, • Direct Memory Access**

Chap. 12: Memory Organization

- **Memory Hierarchy, • Main Memory**
- **Auxiliary Memory. • Associative Memory**
- **Cache Memory, • Virtual Memory**

Chap. 13: Multiprocessors

(Δ)

- **Characteristics of Multiprocessors**
- **Interconnection Structures**
- **Interprocessor Arbitration**
- **Interprocessor Communication/Synchronization**
- **Cache Coherence**

SIMPLE DIGITAL SYSTEMS

- **Combinational and sequential circuits (learned in Chapters 1 and 2) can be used to create simple digital systems.**
- **These are the low-level building blocks of a digital computer.**
- **Simple digital systems are frequently characterized in terms of**
 - the registers they contain, and
 - the operations that they perform.
- **Typically,**
 - What operations are performed on the data in the registers
 - What information is passed between registers

REGISTER TRANSFER AND MICROOPERATIONS

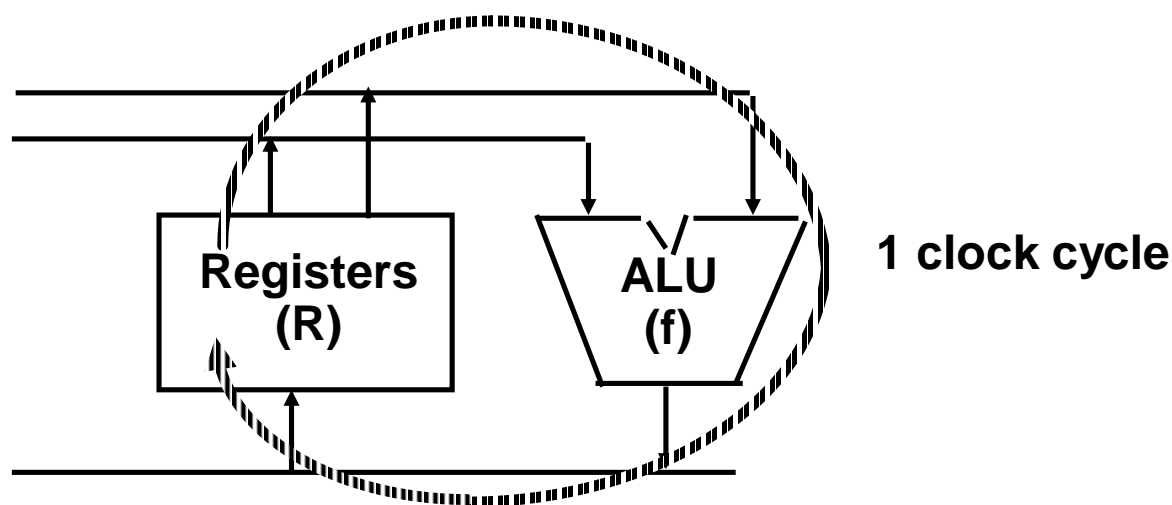
- **Register Transfer Language**
- **Register Transfer**
- **Bus and Memory Transfers**
- **Arithmetic Microoperations**
- **Logic Microoperations**
- **Shift Microoperations**
- **Arithmetic Logic Shift Unit**

MICROOPERATIONS (1)

- The operations on the data in registers are called microoperations.
- The functions built into registers are examples of microoperations
 - Shift
 - Load
 - Clear
 - Increment
 - ...

MICROOPERATION (2)

An elementary operation performed (during one clock pulse), on the information stored in one or more registers



$$R \leftarrow f(R, R)$$

f: shift, load, clear, increment, add, subtract, complement, and, or, xor, ...

ORGANIZATION OF A DIGITAL SYSTEM

- **Definition of the (internal) organization of a computer**
 - **Set of registers and their functions**
 - **Microoperations set**

Set of allowable microoperations provided by the organization of the computer
 - **Control signals that initiate the sequence of microoperations (to perform the functions)**

REGISTER TRANSFER LEVEL

- Viewing a computer, or any digital system, in this way is called the register transfer level
- This is because we're focusing on
 - The system's registers
 - The data transformations in them, and
 - The data transfers between them.

REGISTER TRANSFER LANGUAGE

- Rather than specifying a digital system in words, a specific notation is used, *register transfer language*
- For any function of the computer, the register transfer language can be used to describe the (sequence of) microoperations
- Register transfer language
 - A symbolic language
 - A convenient tool for describing the internal organization of digital computers
 - Can also be used to facilitate the design process of digital systems.

DESIGNATION OF REGISTERS

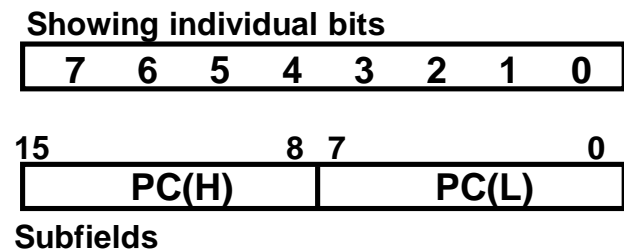
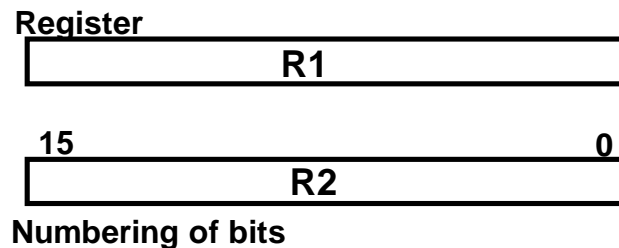
- Registers are designated by capital letters, sometimes followed by numbers (e.g., A, R13, IR)
- Often the names indicate function:
 - MAR - memory address register
 - PC - program counter
 - IR - instruction register
- Registers and their contents can be viewed and represented in *various ways*
 - A register can be viewed as a single entity:



- Registers may also be represented showing the bits of data they contain

DESIGNATION OF REGISTERS

- Designation of a register
 - a register
 - portion of a register
 - a bit of a register
- Common ways of drawing the block diagram of a register



REGISTER TRANSFER

- Copying the contents of one register to another is a register transfer
- A register transfer is indicated as

R2 \leftarrow R1

- In this case the contents of register R2 are copied (loaded) into register R1
- A simultaneous transfer of all bits from the source R1 to the destination register R2, during one clock pulse
- Note that this is a non-destructive; i.e. the contents of R1 are not altered by copying (loading) them to R2

REGISTER TRANSFER

- A register transfer such as

$R3 \leftarrow R5$

Implies that the digital system has

- the data lines from the source register (R5) to the destination register (R3)
- Parallel load in the destination register (R3)
- Control lines to perform the action

CONTROL FUNCTIONS

- Often actions need to only occur if a certain condition is true
- This is similar to an “if” statement in a programming language
- In digital systems, this is often done via a *control signal*, called a *control function*
 - If the signal is 1, the action takes place
- This is represented as:

P: $R2 \leftarrow R1$

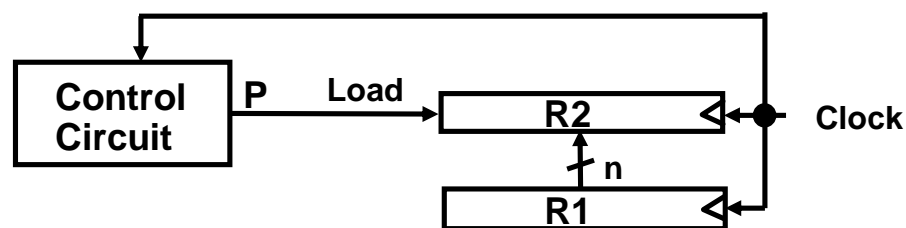
Which means “if $P = 1$, then load the contents of register R1 into register R2”, i.e., if $(P = 1)$ then $(R2 \leftarrow R1)$

HARDWARE IMPLEMENTATION OF CONTROLLED TRANSFERS

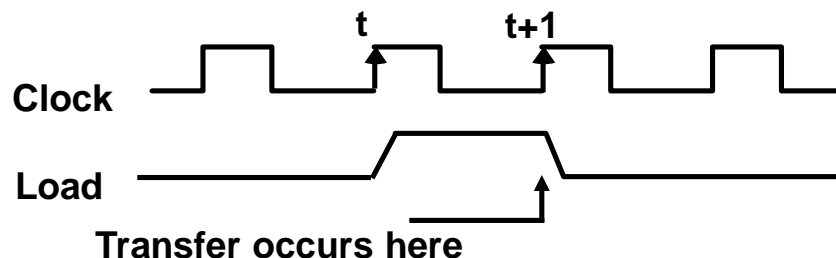
Implementation of controlled transfer

P: $R2 \leftarrow R1$

Block diagram



Timing diagram



- The same clock controls the circuits that generate the control function and the destination register
- Registers are assumed to use *positive-edge-triggered* flip-flops

SIMULTANEOUS OPERATIONS

- If two or more operations are to occur simultaneously, they are separated with commas

P: R3 \leftarrow R5, MAR \leftarrow IR

- Here, if the control function P = 1, load the contents of R5 into R3, and at the same time (clock), load the contents of register IR into register MAR

BASIC SYMBOLS FOR REGISTER TRANSFERS

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

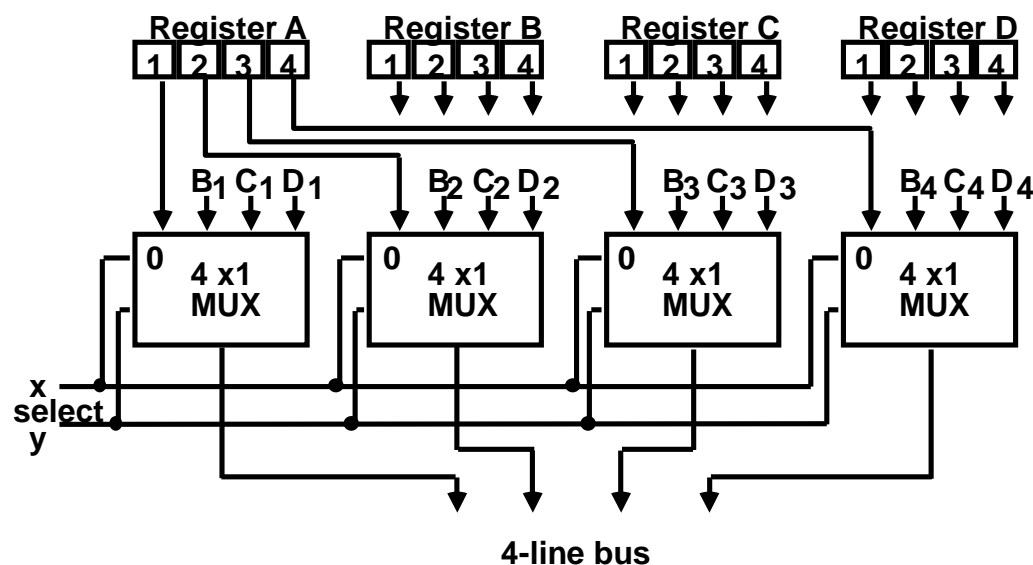
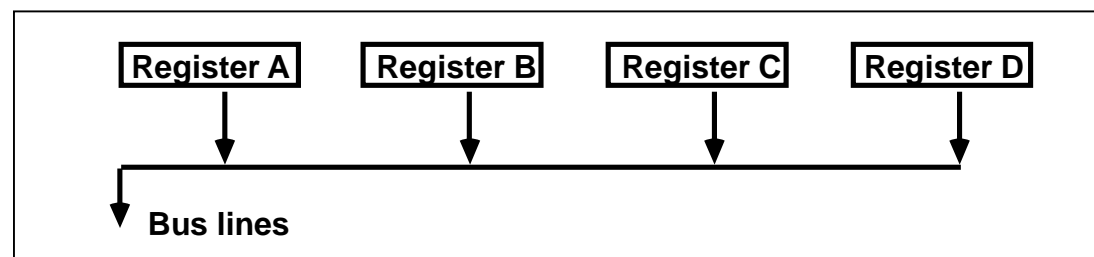
CONNECTING REGISTERS

- In a digital system with many registers, it is impractical to have data and control lines to directly allow each register to be loaded with the contents of every possible other registers
- To completely connect n registers $\rightarrow n(n-1)$ lines
- $O(n^2)$ cost
 - This is not a realistic approach to use in a large digital system
- Instead, take a different approach
- Have one centralized set of circuits for data transfer – the **bus**
- Have control circuits to select which register is the source, and which is the destination

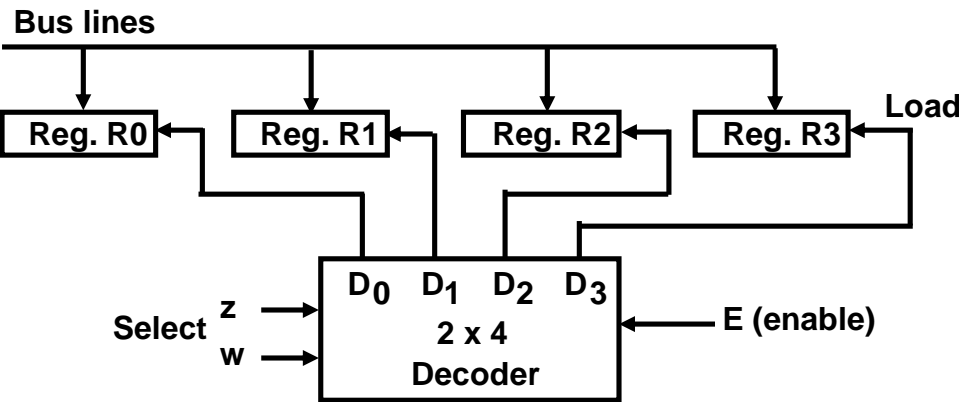
BUS AND BUS TRANSFER

Bus is a path(of a group of wires) over which information is transferred, from any of several sources to any of several destinations.

From a register to bus: $BUS \leftarrow R$



TRANSFER FROM BUS TO A DESTINATION REGISTER

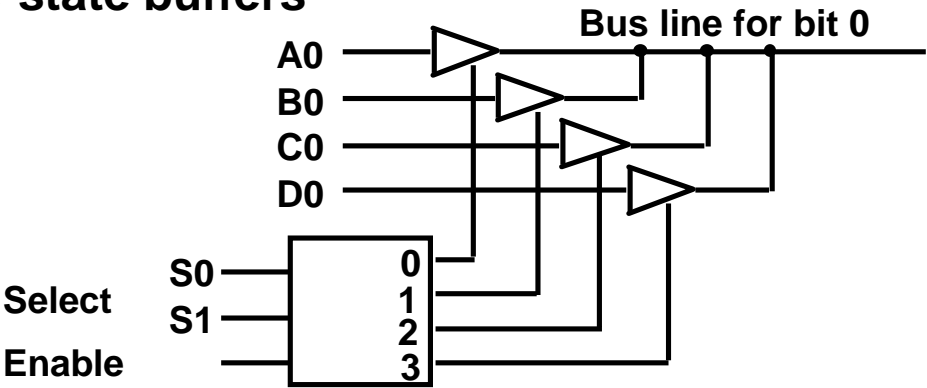


Three-State Bus Buffers

Normal input A
Control input C



Bus line with three-state buffers



BUS TRANSFER IN RTL

- Depending on whether the bus is to be mentioned explicitly or not, register transfer can be indicated as either

$R2 \leftarrow R1$

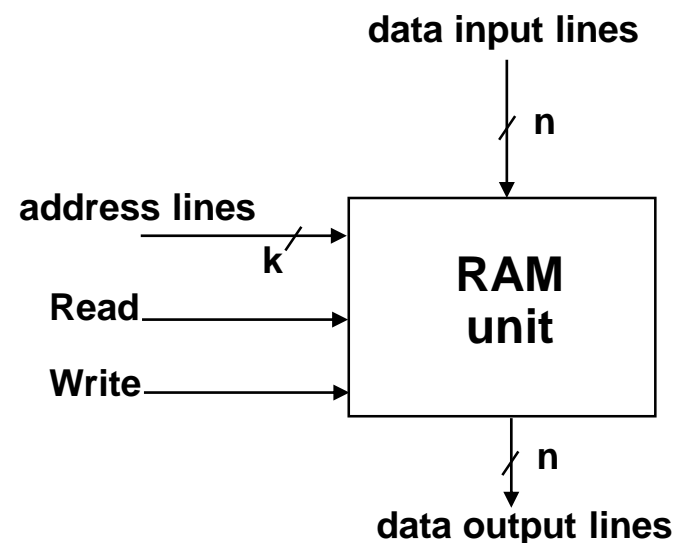
or

$BUS \leftarrow R1, R2 \leftarrow BUS$

- In the former case the bus is implicit, but in the latter, it is explicitly indicated

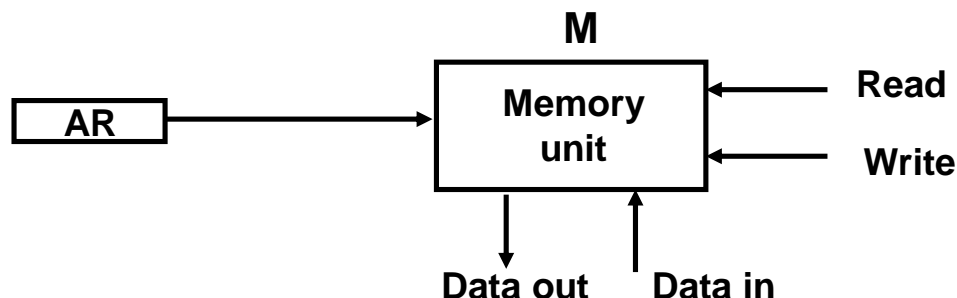
MEMORY (RAM)

- Memory (RAM) can be thought as a sequential circuits containing some number of registers
- These registers hold the *words* of memory
- Each of the r registers is indicated by an *address*
- These addresses range from 0 to $r-1$
- Each register (word) can hold n bits of data
- Assume the RAM contains $r = 2^k$ words. It needs the following
 - n data input lines
 - n data output lines
 - k address lines
 - A Read control line
 - A Write control line



MEMORY TRANSFER

- Collectively, the memory is viewed at the register level as a device, M.
- Since it contains multiple locations, we must specify which address in memory we will be using
- This is done by indexing memory references
- Memory is usually accessed in computer systems by putting the desired address in a special register, the *Memory Address Register (MAR, or AR)*
- When memory is accessed, the contents of the MAR get sent to the memory unit's address lines



MEMORY READ

- To read a value from a location in memory and load it into a register, the register transfer language notation looks like this:

$R1 \leftarrow M[MAR]$

- This causes the following to occur
 - The contents of the MAR get sent to the memory address lines
 - A Read (= 1) gets sent to the memory unit
 - The contents of the specified address are put on the memory's output data lines
 - These get sent over the bus to be loaded into register R1

MEMORY WRITE

- To write a value from a register to a location in memory looks like this in register transfer language:

$M[MAR] \leftarrow R1$

- This causes the following to occur
 - The contents of the MAR get sent to the memory address lines
 - A Write (= 1) gets sent to the memory unit
 - The values in register R1 get sent over the bus to the data input lines of the memory
 - The values get loaded into the specified address in the memory

SUMMARY OF R. TRANSFER MICROOPERATIONS

$A \leftarrow B$

Transfer content of reg. B into reg. A

$AR \leftarrow DR(AD)$

Transfer content of AD portion of reg. DR into reg. AR

$A \leftarrow \text{constant}$

Transfer a binary constant into reg. A

$ABUS \leftarrow R1,$

Transfer content of R1 into bus A and, at the same time,
transfer content of bus A into R2

$R2 \leftarrow ABUS$

AR

Address register

DR

Data register

$M[R]$

Memory word specified by reg. R

M

Equivalent to $M[AR]$

$DR \leftarrow M$

Memory *read* operation: transfers content of
memory word specified by AR into DR

$M \leftarrow DR$

Memory *write* operation: transfers content of
DR into memory word specified by AR

MICROOPERATIONS

- **Computer system microoperations are of four types:**
 - **Register transfer microoperations**
 - **Arithmetic microoperations**
 - **Logic microoperations**
 - **Shift microoperations**

ARITHMETIC MICROOPERATIONS

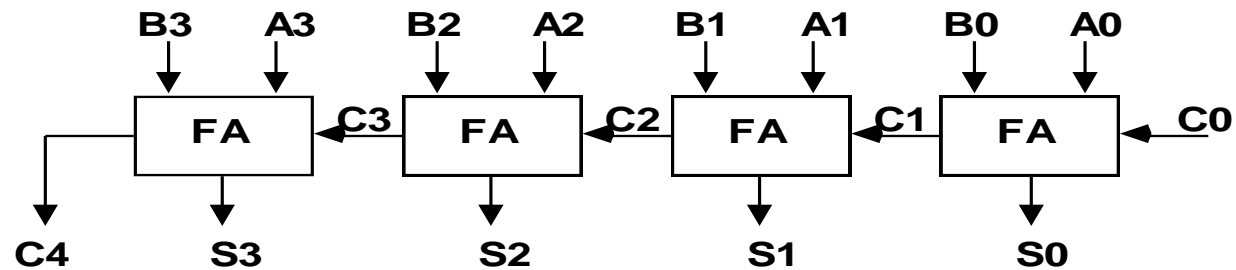
- The basic arithmetic microoperations are
 - Addition
 - Subtraction
 - Increment
 - Decrement
- The additional arithmetic microoperations are
 - Add with carry
 - Subtract with borrow
 - Transfer/Load
 - etc. ...

Summary of Typical Arithmetic Micro-Operations

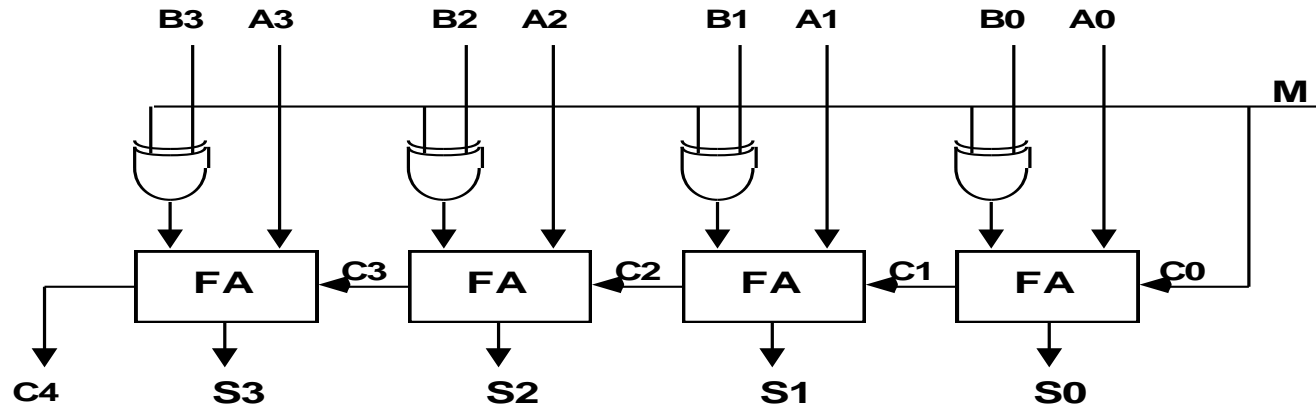
$R3 \leftarrow R1 + R2$	Contents of R1 plus R2 transferred to R3
$R3 \leftarrow R1 - R2$	Contents of R1 minus R2 transferred to R3
$R2 \leftarrow R2'$	Complement the contents of R2
$R2 \leftarrow R2' + 1$	2's complement the contents of R2 (negate)
$R3 \leftarrow R1 + R2' + 1$	subtraction
$R1 \leftarrow R1 + 1$	Increment
$R1 \leftarrow R1 - 1$	Decrement

BINARY ADDER / SUBTRACTOR / INCREMENTER

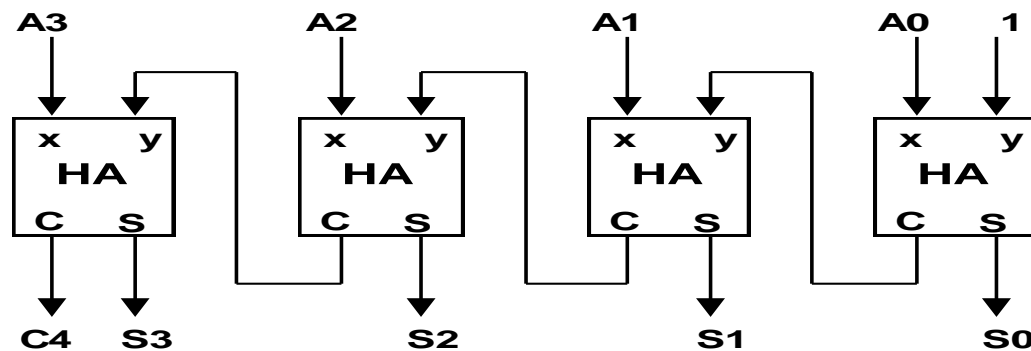
Binary Adder



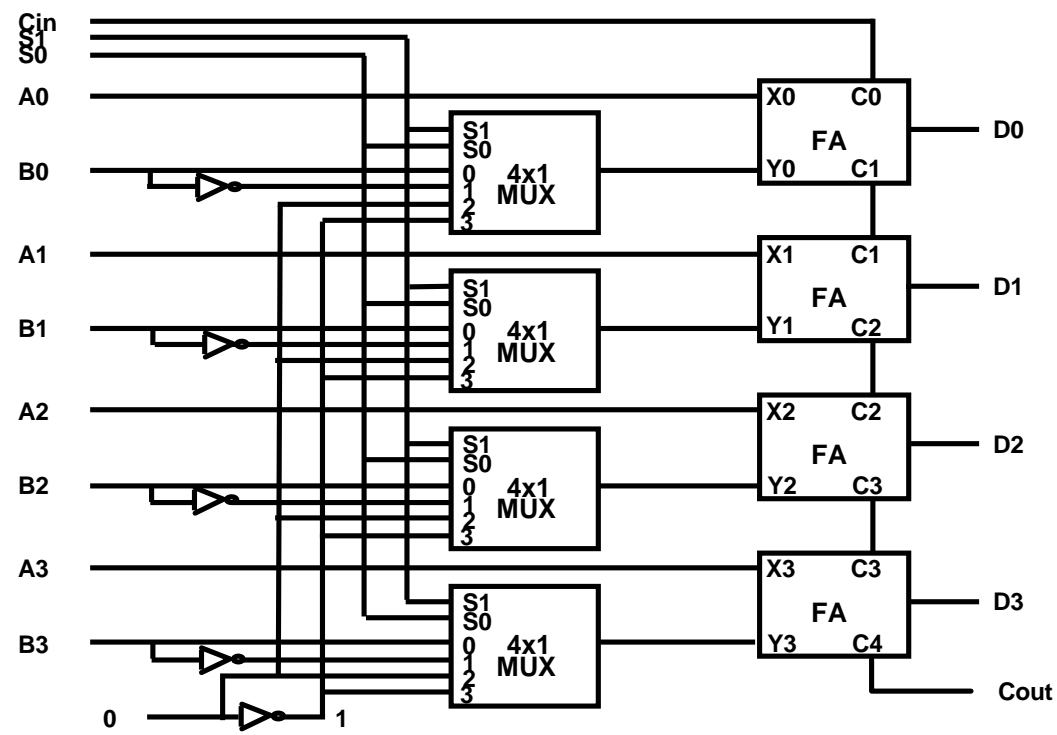
Binary Adder-Subtractor



Binary Incrementer



ARITHMETIC CIRCUIT



S1	S0	Cin	Y	Output	Microoperation
0	0	0	B	$D = A + B$	Add
0	0	1	B	$D = A + B + 1$	Add with carry
0	1	0	B'	$D = A + B'$	Subtract with borrow
0	1	1	B'	$D = A + B' + 1$	Subtract
1	0	0	0	$D = A$	Transfer A
1	0	1	0	$D = A + 1$	Increment A
1	1	0	1	$D = A - 1$	Decrement A
1	1	1	1	$D = A$	Transfer A

LOGIC MICROOPERATIONS

- **Specify binary operations on the strings of bits in registers**
 - Logic microoperations are bit-wise operations, i.e., they work on the individual bits of data
 - useful for bit manipulations on binary data
 - useful for making logical decisions based on the bit value
- **There are, in principle, 16 different logic functions that can be defined over two binary input variables**

A	B	F ₀	F ₁	F ₂ ... F ₁₃	F ₁₄	F ₁₅
0	0	0	0	0 ... 1	1	1
0	1	0	0	0 ... 1	1	1
1	0	0	0	1 ... 0	1	1
1	1	0	1	0 ... 1	0	1

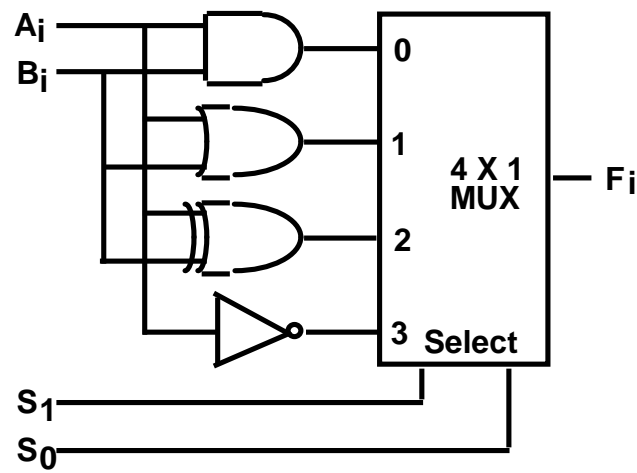
- **However, most systems only implement four of these**
 - AND (\wedge), OR (\vee), XOR (\oplus), Complement/NOT
- **The others can be created from combination of these**

LIST OF LOGIC MICROOPERATIONS

- List of Logic Microoperations
 - 16 different logic operations with 2 binary vars.
 - n binary vars $\rightarrow 2^{2^n}$ functions
- Truth tables for 16 functions of 2 variables and the corresponding 16 logic micro-operations

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

HARDWARE IMPLEMENTATION OF LOGIC MICROOPERATIONS



Function table

S_1	S_0	Output	μ -operation
0	0	$F = A \wedge B$	AND
0	1	$F = A \vee B$	OR
1	0	$F = A \oplus B$	XOR
1	1	$F = A'$	Complement

APPLICATIONS OF LOGIC MICROOPERATIONS

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

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

SELECTIVE SET

- In a selective set operation, the bit pattern in B is used to set certain bits in A

1 1 0 0	A_t	
1 0 1 0	B	
<hr/>		
1 1 1 0	A_{t+1}	$(A \leftarrow A + B)$

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

SELECTIVE COMPLEMENT

- In a selective complement operation, the bit pattern in B is used to *complement* certain bits in A

1 1 0 0	A_t	
1 0 1 0	B	
<hr/>		
0 1 1 0	A_{t+1}	$(A \leftarrow A \oplus B)$

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

SELECTIVE CLEAR

- In a selective clear operation, the bit pattern in B is used to *clear* certain bits in A

1 1 0 0	A_t	
1 0 1 0	B	
<hr/>		
0 1 0 0	A_{t+1}	$(A \leftarrow A \cdot B')$

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

MASK OPERATION

- In a mask operation, the bit pattern in B is used to *clear* certain bits in A

1 1 0 0	A_t	
1 0 1 0	B	
<hr/>		
1 0 0 0	A_{t+1}	$(A \leftarrow A \cdot B)$

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

CLEAR OPERATION

- In a clear operation, if the bits in the same position in A and B are the same, they are cleared in A, otherwise they are set in A

1 1 0 0	A_t	
1 0 1 0	B	
<hr/>		
0 1 1 0	A_{t+1}	$(A \leftarrow A \oplus B)$

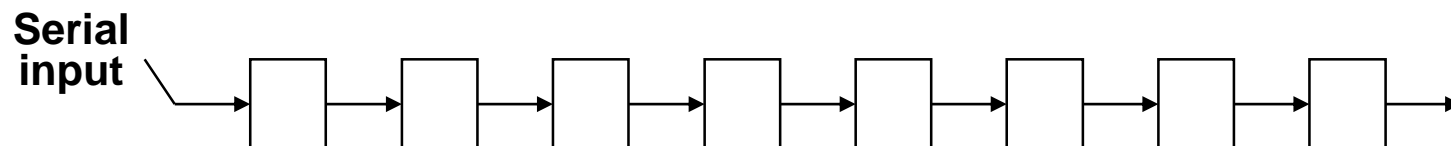
INSERT OPERATION

- An insert operation is used to introduce a specific bit pattern into A register, leaving the other bit positions unchanged
- This is done as
 - A mask operation to clear the desired bit positions, followed by
 - An OR operation to introduce the new bits into the desired positions
 - Example
 - » Suppose you wanted to introduce 1010 into the low order four bits of A:
1101 1000 1011 0001 A (Original)
1101 1000 1011 1010 A (Desired)

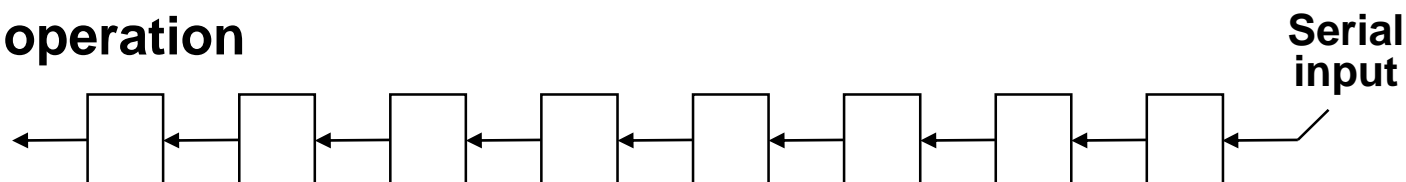
» 1101 1000 1011 0001	A (Original)
1111 1111 1111 0000	Mask
<hr/>	
1101 1000 1011 0000	A (Intermediate)
0000 0000 0000 1010	Added bits
<hr/>	
1101 1000 1011 1010	A (Desired)

SHIFT MICROOPERATIONS

- There are three types of shifts
 - *Logical shift*
 - *Circular shift*
 - *Arithmetic shift*
- What differentiates them is the information that goes into the serial input
- A right shift operation

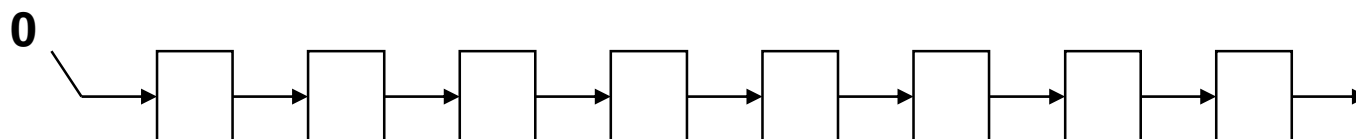


- A left shift operation

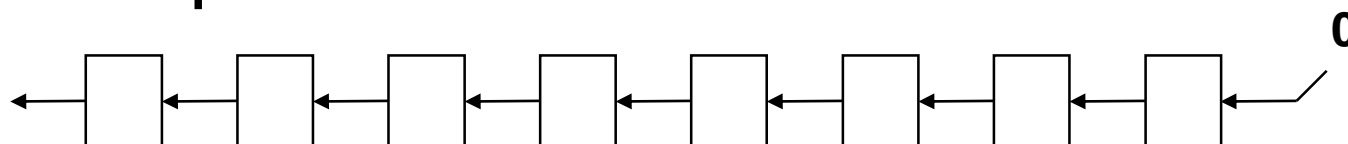


LOGICAL SHIFT

- In a logical shift the serial input to the shift is a 0.
- A right logical shift operation:



- A left logical shift operation:

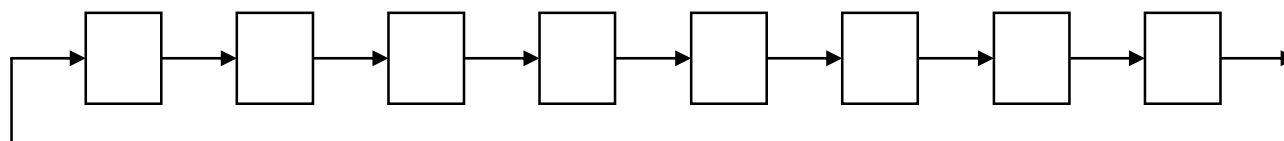


- In a Register Transfer Language, the following notation is used
 - *shl* for a logical shift left
 - *shr* for a logical shift right
 - Examples:
 - » $R2 \leftarrow shr\ R2$
 - » $R3 \leftarrow shl\ R3$

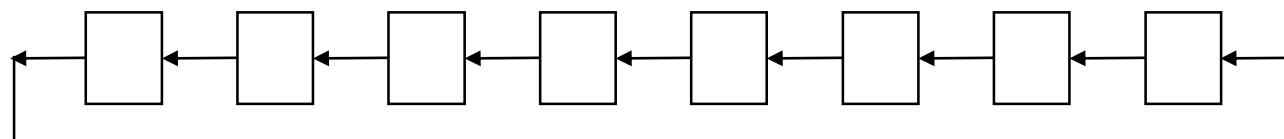
CIRCULAR SHIFT

- In a circular shift the serial input is the bit that is shifted out of the other end of the register.

- A right circular shift operation:



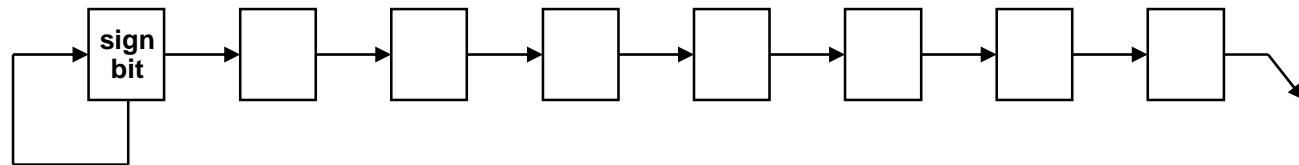
- A left circular shift operation:



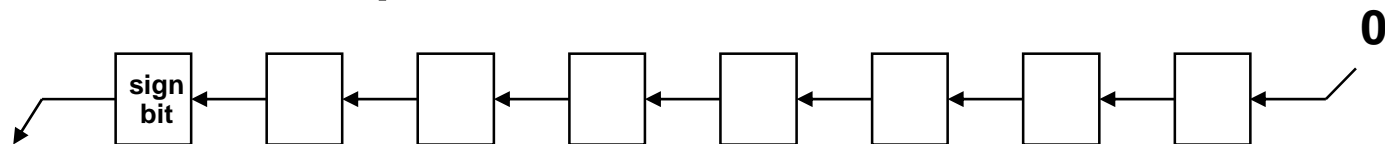
- In a RTL, the following notation is used
 - *cil* for a circular shift left
 - *cir* for a circular shift right
 - Examples:
 - » $R2 \leftarrow cir\ R2$
 - » $R3 \leftarrow cil\ R3$

ARITHMETIC SHIFT

- An arithmetic shift is meant for signed binary numbers (integer)
- An arithmetic left shift **multiplies** a signed number **by two**
- An arithmetic right shift **divides** a signed number **by two**
- The main distinction of an arithmetic shift is that it must keep the sign of the number the same as it performs the multiplication or division
- A right arithmetic shift operation:

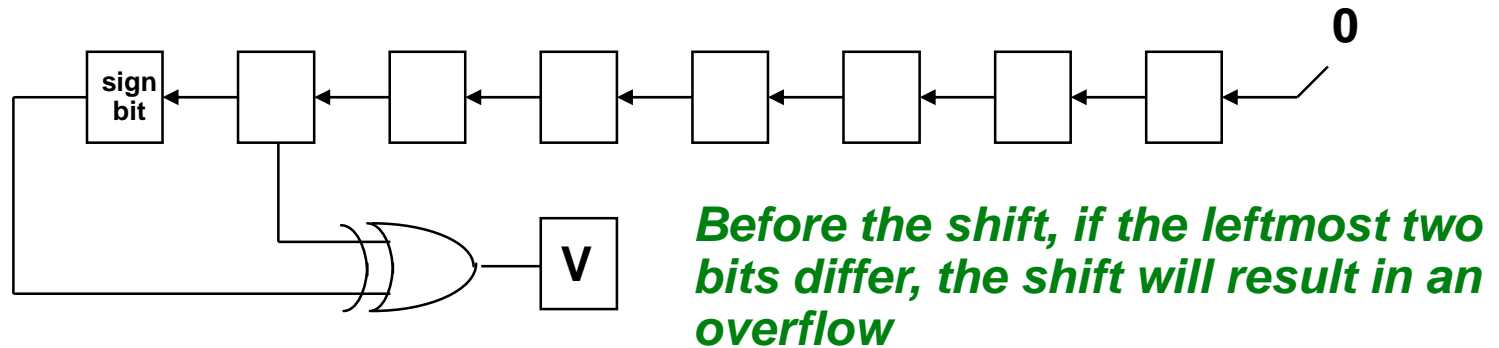


- A left arithmetic shift operation:



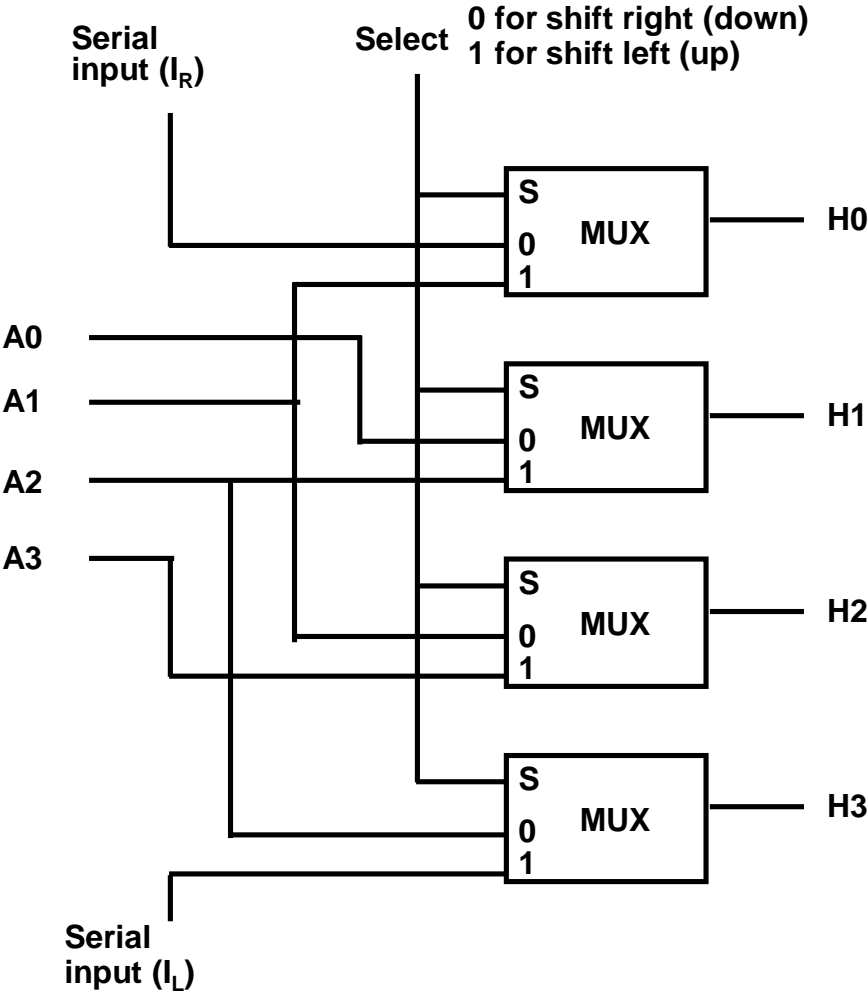
ARITHMETIC SHIFT

- An left arithmetic shift operation must be checked for the **overflow**

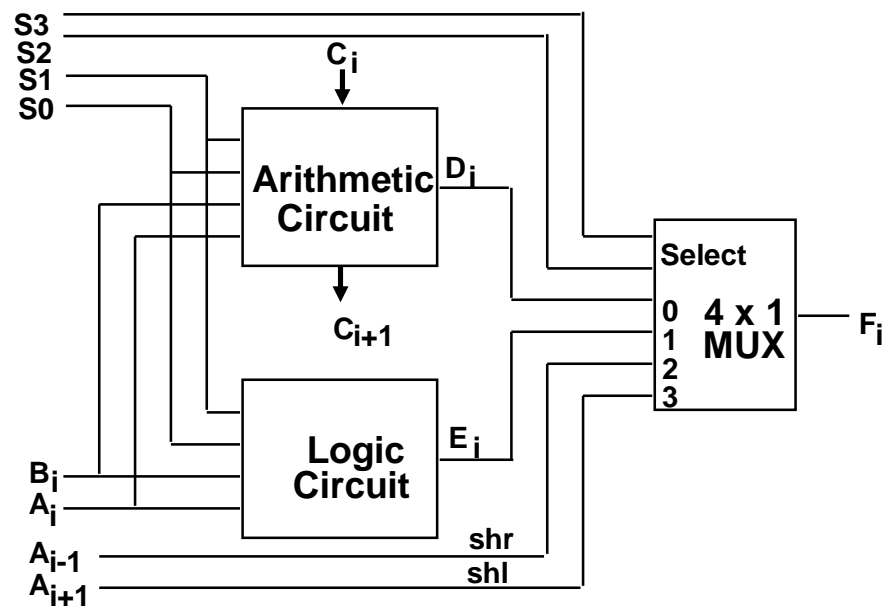


- In a RTL, the following notation is used
 - *ashl* for an arithmetic shift left
 - *ashr* for an arithmetic shift right
 - Examples:
 - » $R2 \leftarrow ashr R2$
 - » $R3 \leftarrow ashl R3$

HARDWARE IMPLEMENTATION OF SHIFT MICROOPERATIONS



ARITHMETIC LOGIC SHIFT UNIT



S3	S2	S1	S0	Cin	Operation	Function
0	0	0	0	0	$F = A$	Transfer A
0	0	0	0	1	$F = A + 1$	Increment A
0	0	0	1	0	$F = A + B$	Addition
0	0	0	1	1	$F = A + B + 1$	Add with carry
0	0	1	0	0	$F = A + B'$	Subtract with borrow
0	0	1	0	1	$F = A + B' + 1$	Subtraction
0	0	1	1	0	$F = A - 1$	Decrement A
0	0	1	1	1	$F = A$	Transfer A
0	1	0	0	X	$F = A \wedge B$	AND
0	1	0	1	X	$F = A \vee B$	OR
0	1	1	0	X	$F = A \oplus B$	XOR
0	1	1	1	X	$F = A'$	Complement A
1	0	X	X	X	$F = \text{shr } A$	Shift right A into F
1	1	X	X	X	$F = \text{shl } A$	Shift left A into F

BASIC COMPUTER ORGANIZATION AND DESIGN

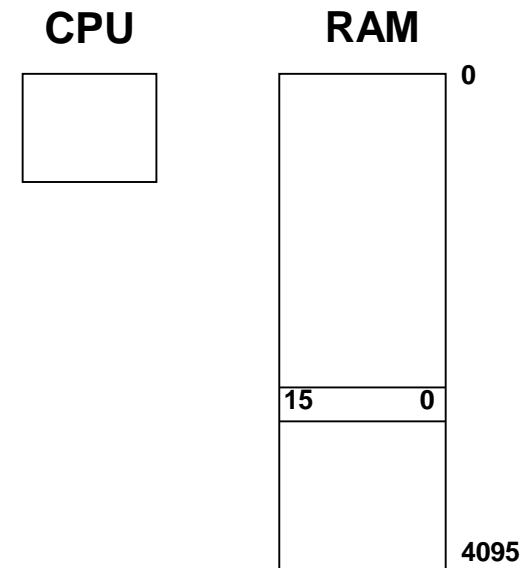
- **Instruction Codes**
- **Computer Registers**
- **Computer Instructions**
- **Timing and Control**
- **Instruction Cycle**
- **Memory Reference Instructions**
- **Input-Output and Interrupt**
- **Complete Computer Description**
- **Design of Basic Computer**
- **Design of Accumulator Logic**

INTRODUCTION

- Every different processor type has its own design (different registers, buses, microoperations, machine instructions, etc)
- Modern processor is a very complex device
- It contains
 - Many registers
 - Multiple arithmetic units, for both integer and floating point calculations
 - The ability to pipeline several consecutive instructions to speed execution
 - Etc.
- However, to understand how processors work, we will start with a simplified processor model
- This is similar to what real processors were like ~25 years ago
- M. Morris Mano introduces a simple processor model he calls the *Basic Computer*
- We will use this to introduce processor organization and the relationship of the RTL model to the higher level computer processor

THE BASIC COMPUTER

- The Basic Computer has two components, a processor and memory
- The memory has 4096 words in it
 - $4096 = 2^{12}$, so it takes 12 bits to select a word in memory
- Each word is 16 bits long



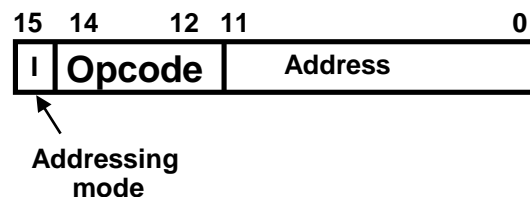
INSTRUCTIONS

- **Program**
 - A sequence of (machine) instructions
- **(Machine) Instruction**
 - A group of bits that tell the computer to *perform a specific operation* (a sequence of micro-operation)
- The instructions of a program, along with any needed data are stored in memory
- The CPU reads the next instruction from memory
- It is placed in an *Instruction Register* (IR)
- Control circuitry in control unit then translates the instruction into the sequence of microoperations necessary to implement it

INSTRUCTION FORMAT

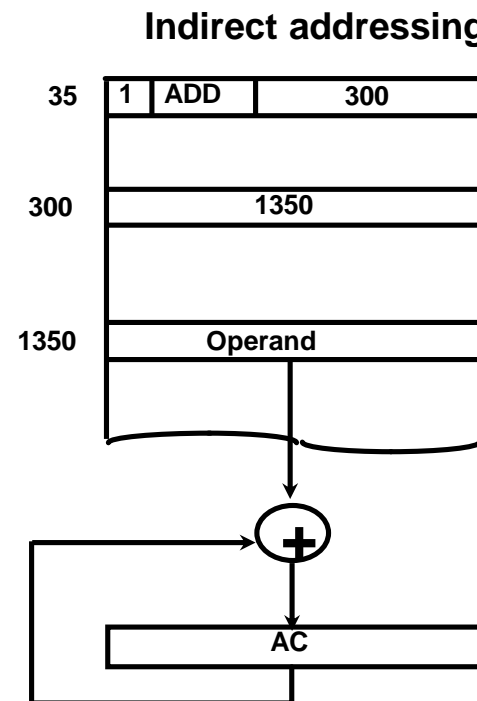
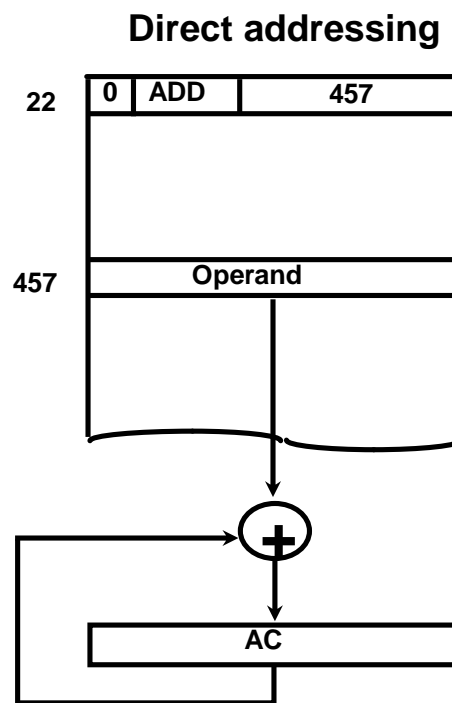
- A computer instruction is often divided into two parts
 - An *opcode* (Operation Code) that specifies the operation for that instruction
 - An *address* that specifies the registers and/or locations in memory to use for that operation
- In the Basic Computer, since the memory contains 4096 ($= 2^{12}$) words, we need 12 bits to specify which memory address this instruction will use
- In the Basic Computer, bit 15 of the instruction specifies the *addressing mode* (0: direct addressing, 1: indirect addressing)
- Since the memory words, and hence the instructions, are 16 bits long, that leaves 3 bits for the instruction's opcode

Instruction Format



ADDRESSING MODES

- The address field of an instruction can represent either
 - Direct address: the address in memory of the data to use (the address of the operand), or
 - Indirect address: the address in memory of the address in memory of the data to use



- **Effective Address (EA)**
 - The address, that can be directly used without modification to access an operand for a computation-type instruction, or as the target address for a branch-type instruction

PROCESSOR REGISTERS

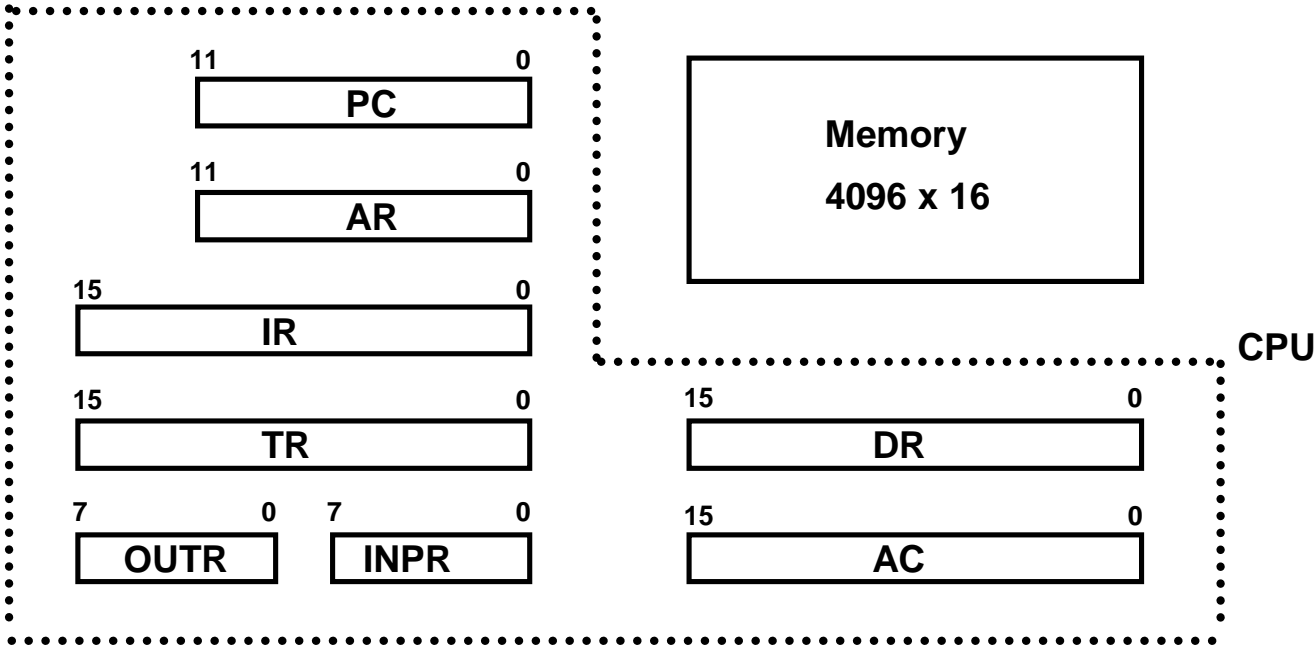
- A processor has many registers to hold instructions, addresses, data, etc
- The processor has a register, the *Program Counter* (**PC**) that holds the memory address of the next instruction to get
 - Since the memory in the Basic Computer only has 4096 locations, the PC only needs 12 bits
- In a direct or indirect addressing, the processor needs to keep track of what locations in memory it is addressing: The *Address Register* (**AR**) is used for this
 - The AR is a 12 bit register in the Basic Computer
- When an operand is found, using either direct or indirect addressing, it is placed in the *Data Register* (**DR**). The processor then uses this value as data for its operation
- The Basic Computer has a single *general purpose register* – the *Accumulator* (**AC**)

PROCESSOR REGISTERS

- The significance of a general purpose register is that it can be referred to in instructions
 - e.g. load AC with the contents of a specific memory location; store the contents of AC into a specified memory location
- Often a processor will need a scratch register to store intermediate results or other temporary data; in the Basic Computer this is the *Temporary Register* (TR)
- The Basic Computer uses a very simple model of input/output (I/O) operations
 - Input devices are considered to send 8 bits of character data to the processor
 - The processor can send 8 bits of character data to output devices
- The *Input Register* (INPR) holds an 8 bit character gotten from an input device
- The *Output Register* (OUTR) holds an 8 bit character to be send to an output device

BASIC COMPUTER REGISTERS

Registers in the Basic Computer



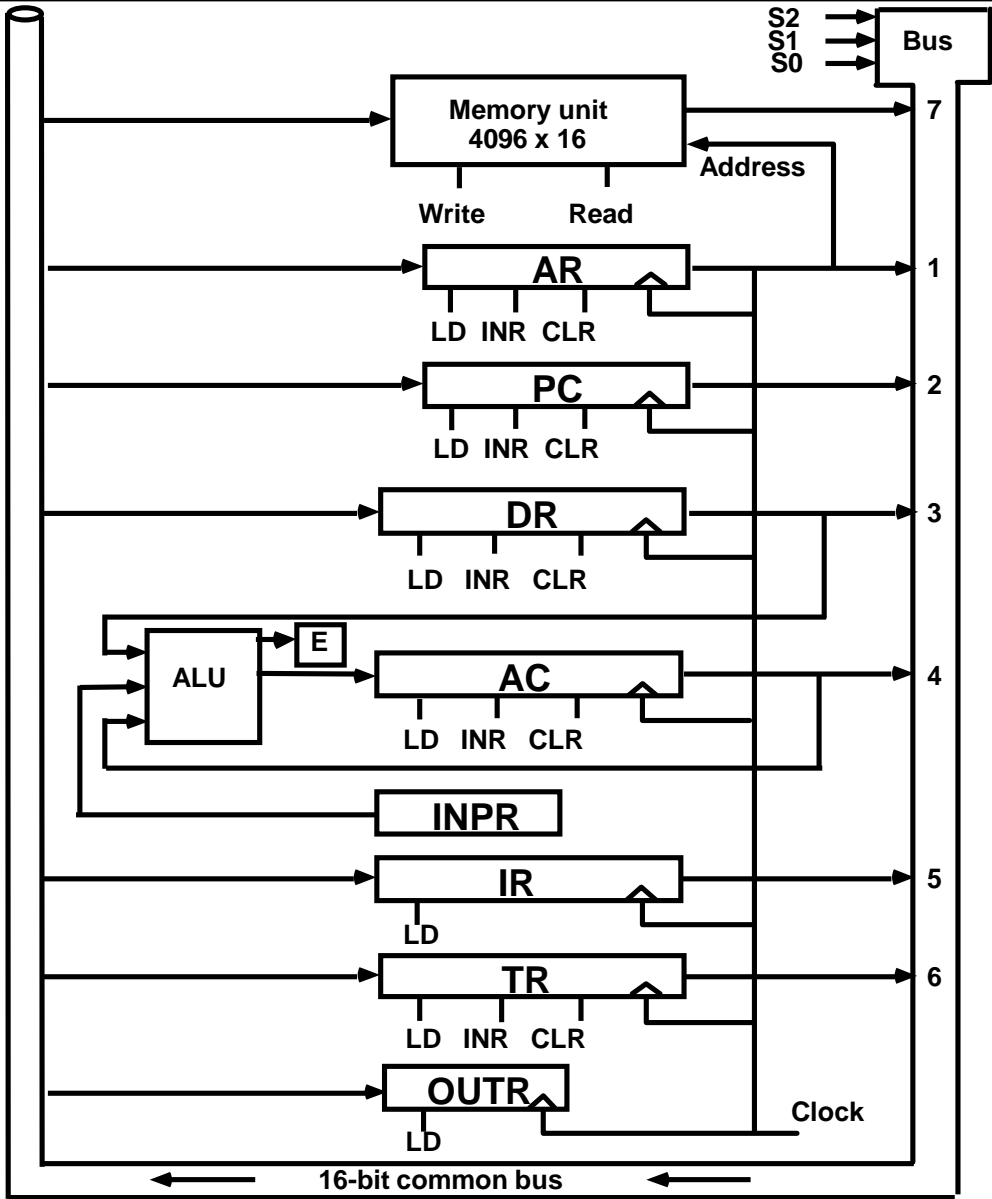
List of BC Registers

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

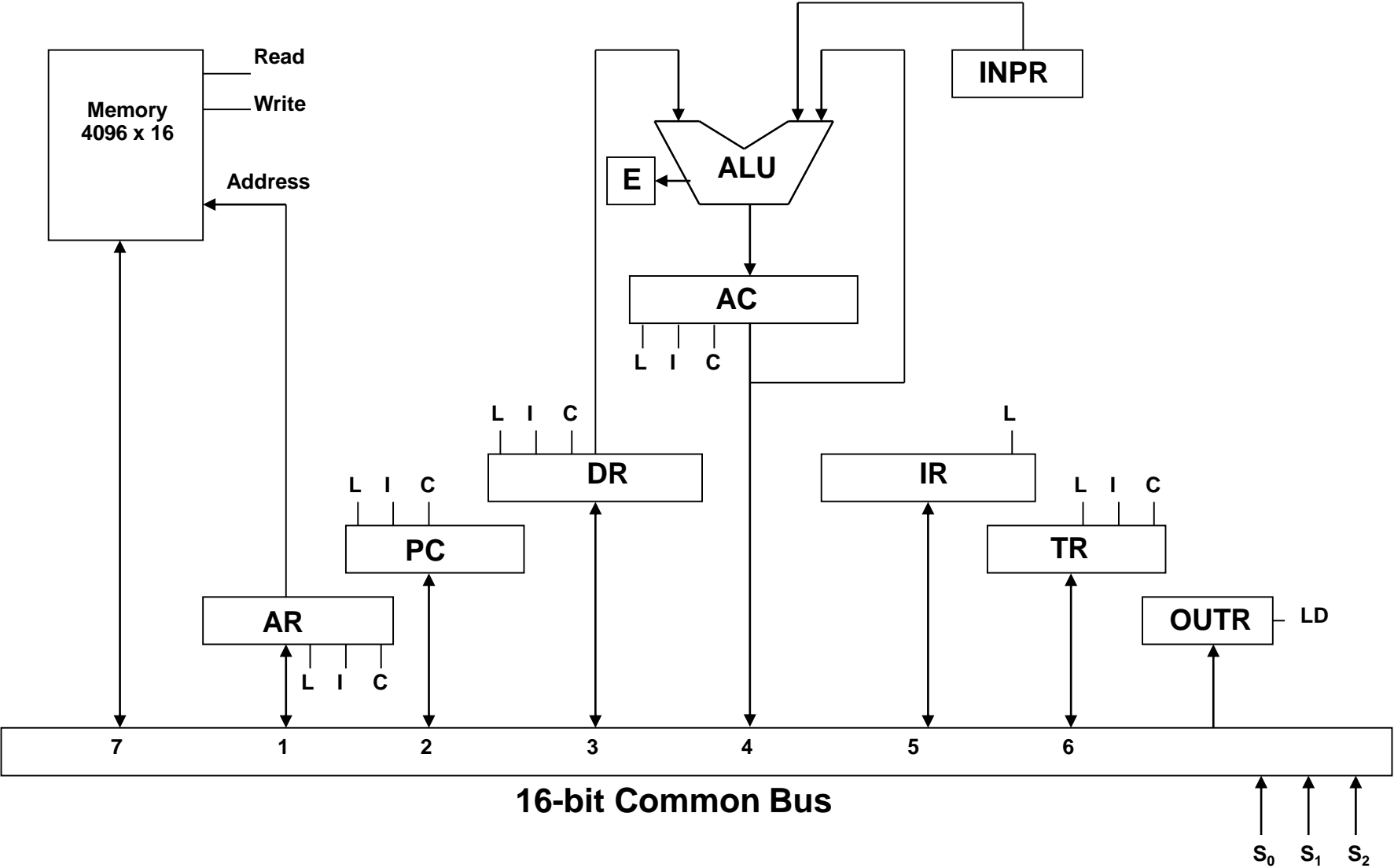
COMMON BUS SYSTEM

- The registers in the Basic Computer are connected using a bus
- This gives a savings in circuitry over complete connections between registers

COMMON BUS SYSTEM



COMMON BUS SYSTEM



COMMON BUS SYSTEM

- Three control lines, S_2 , S_1 , and S_0 control which register the bus selects as its input

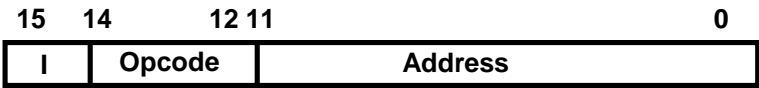
S_2	S_1	S_0	Register
0	0	0	X
0	0	1	AR
0	1	0	PC
0	1	1	DR
1	0	0	AC
1	0	1	IR
1	1	0	TR
1	1	1	Memory

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

BASIC COMPUTER INSTRUCTIONS

- Basic Computer Instruction Format

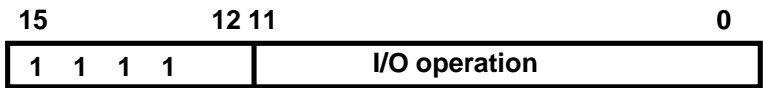
Memory-Reference Instructions (OP-code = 000 ~ 110)



Register-Reference Instructions (OP-code = 111, I = 0)



Input-Output Instructions (OP-code =111, I = 1)



BASIC COMPUTER INSTRUCTIONS

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

INSTRUCTION SET COMPLETENESS

A computer should have a set of instructions so that the user can construct machine language programs to evaluate any function that is known to be computable.

- **Instruction Types**

- Functional Instructions**

- Arithmetic, logic, and shift instructions
 - ADD, CMA, INC, CIR, CIL, AND, CLA

- Transfer Instructions**

- Data transfers between the main memory and the processor registers
 - LDA, STA

- Control Instructions**

- Program sequencing and control
 - BUN, BSA, ISZ

- Input/Output Instructions**

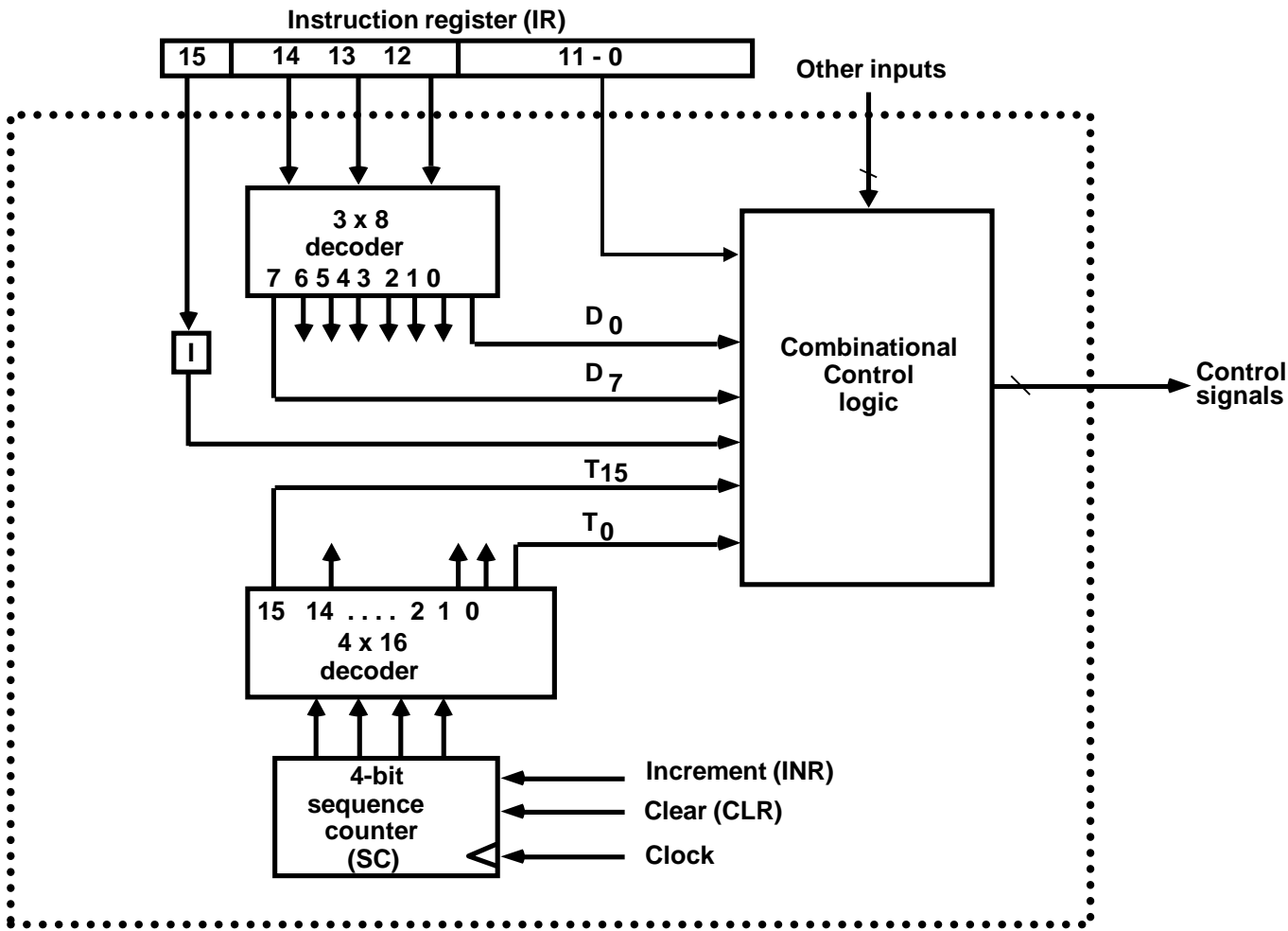
- Input and output
 - INP, OUT

CONTROL UNIT

- Control unit (CU) of a processor translates from machine instructions to the control signals for the microoperations that implement them
- Control units are implemented in one of two ways
- **Hardwired Control**
 - CU is made up of sequential and combinational circuits to generate the control signals
- **Microprogrammed Control**
 - A control memory on the processor contains microprograms that activate the necessary control signals
- We will consider a hardwired implementation of the control unit for the Basic Computer

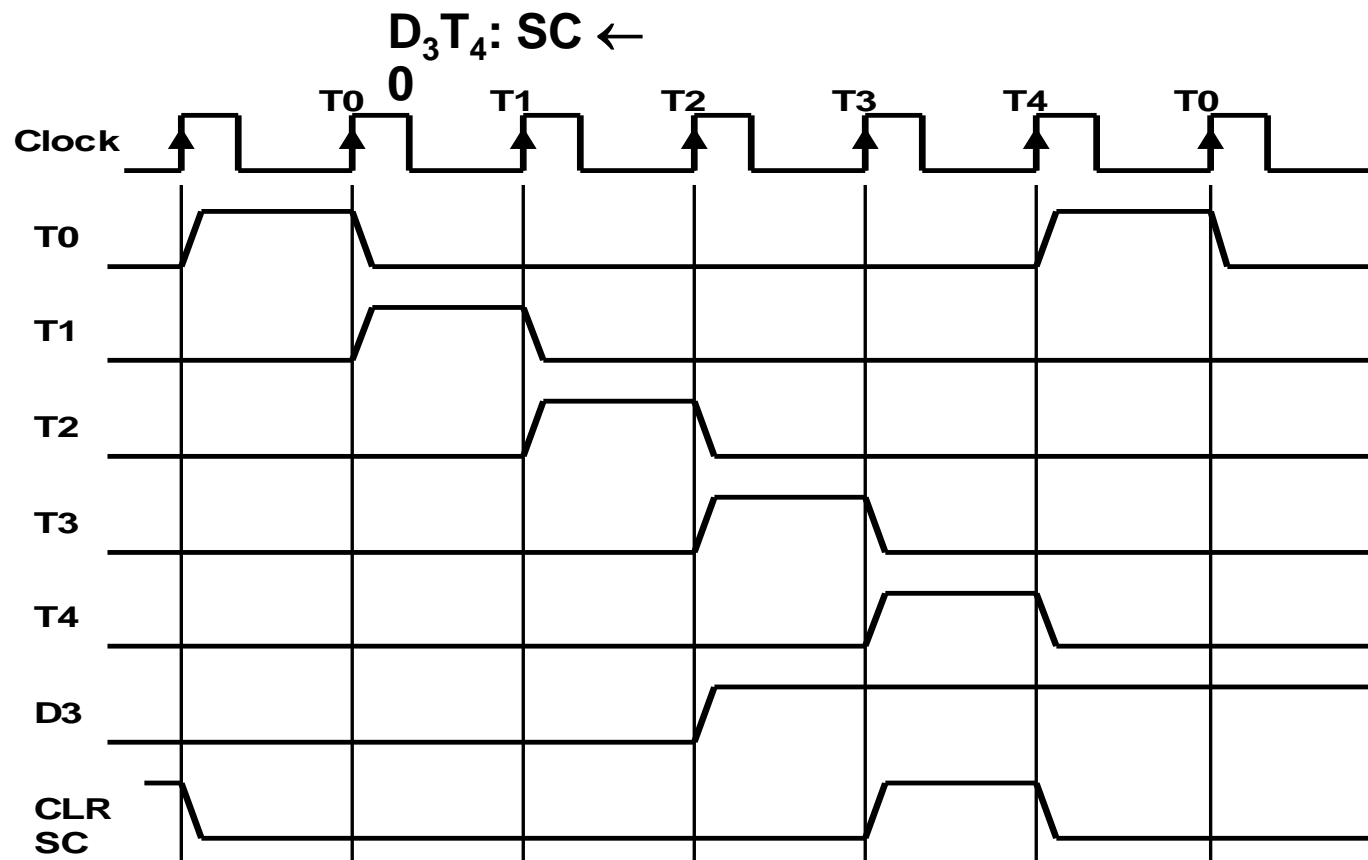
TIMING AND CONTROL

Control unit of Basic Computer



TIMING SIGNALS

- Generated by 4-bit sequence counter and 4×16 decoder
- The SC can be incremented or cleared.
- Example: $T_0, T_1, T_2, T_3, T_4, T_0, T_1, \dots$
Assume: At time T_4 , SC is cleared to 0 if decoder output D3 is active.



INSTRUCTION CYCLE

- **In Basic Computer, a machine instruction is executed in the following cycle:**
 1. **Fetch an instruction from memory**
 2. **Decode the instruction**
 3. **Read the effective address from memory if the instruction has an indirect address**
 4. **Execute the instruction**
- **After an instruction is executed, the cycle starts again at step 1, for the next instruction**
- **Note: Every different processor has its own (different) instruction cycle**

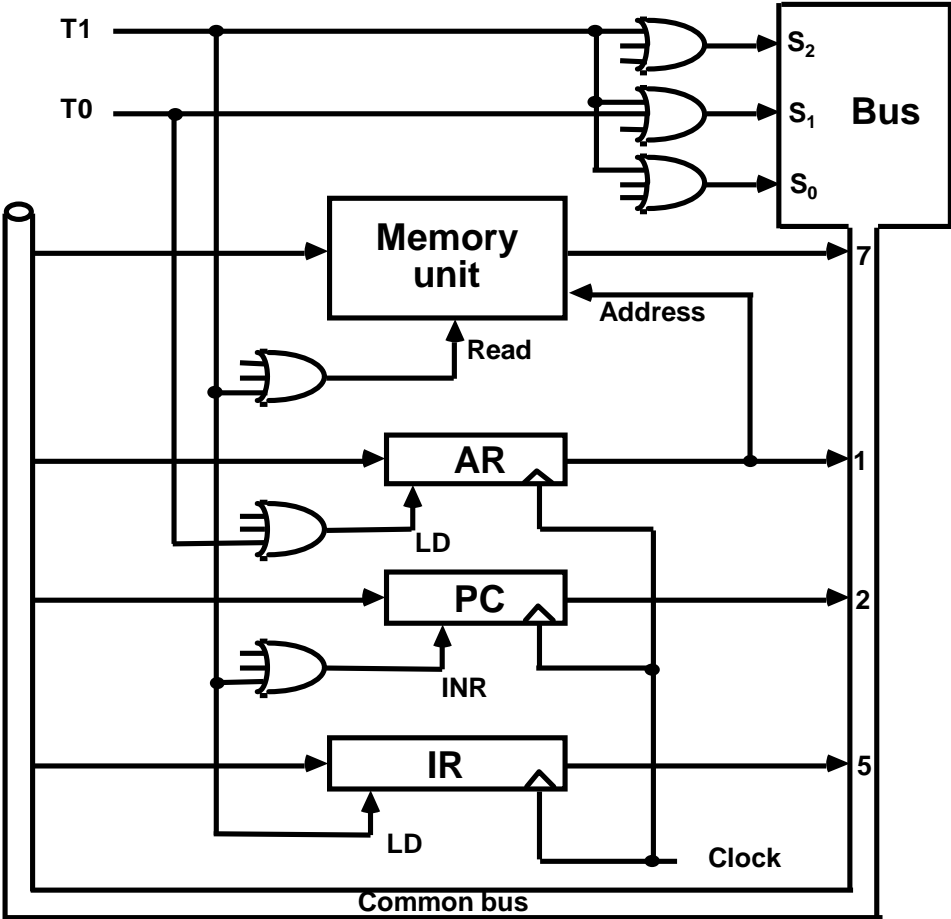
FETCH and DECODE

- Fetch and Decode

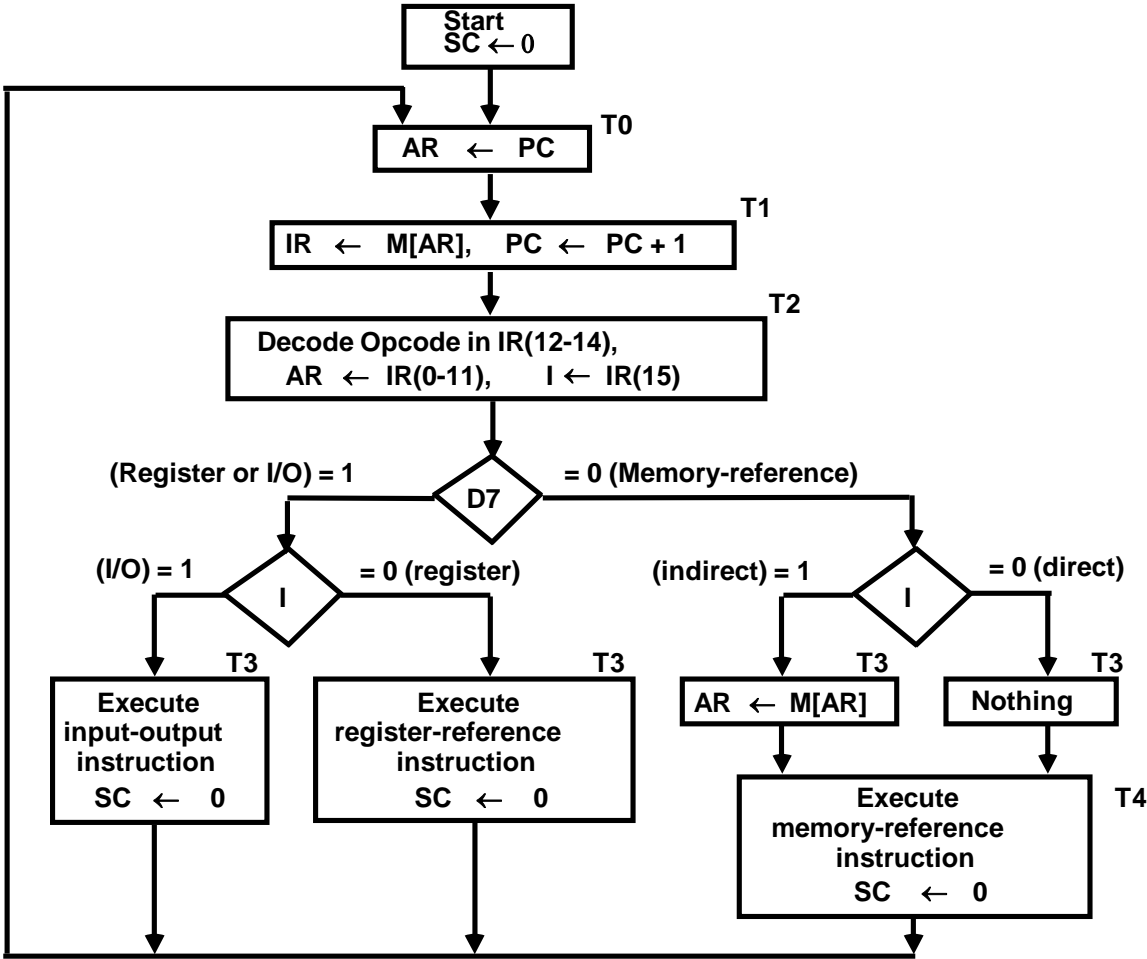
T0: $AR \leftarrow PC$ ($S_0S_1S_2=010$, $T_0=1$)

T1: $IR \leftarrow M[AR]$, $PC \leftarrow PC + 1$ ($S_0S_1S_2=111$, $T_1=1$)

T2: $D_0, \dots, D_7 \leftarrow \text{Decode } IR(12-14)$, $AR \leftarrow IR(0-11)$, $I \leftarrow IR(15)$



DETERMINE THE TYPE OF INSTRUCTION



D₇I T₃: AR ← M[AR]

D₇I' T₃: Nothing

D₇I' T₃: Execute a register-reference instr.

D₇I T₃: Execute an input-output instr.

REGISTER REFERENCE INSTRUCTIONS

Register Reference Instructions are identified when

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

$r = D_7 I' T_3 \Rightarrow$ Register Reference Instruction
 $B_i = IR(i), i=0,1,2,...,11$

	$r:$	$SC \leftarrow 0$
CLA	$rB_{11}:$	$AC \leftarrow 0$
CLE	$rB_{10}:$	$E \leftarrow 0$
CMA	$rB_9:$	$AC \leftarrow AC'$
CME	$rB_8:$	$E \leftarrow E'$
CIR	$rB_7:$	$AC \leftarrow shr\ AC, AC(15) \leftarrow E, E \leftarrow AC(0)$
CIL	$rB_6:$	$AC \leftarrow shl\ AC, AC(0) \leftarrow E, E \leftarrow AC(15)$
INC	$rB_5:$	$AC \leftarrow AC + 1$
SPA	$rB_4:$	if $(AC(15) = 0)$ then $(PC \leftarrow PC+1)$
SNA	$rB_3:$	if $(AC(15) = 1)$ then $(PC \leftarrow PC+1)$
SZA	$rB_2:$	if $(AC = 0)$ then $(PC \leftarrow PC+1)$
SZE	$rB_1:$	if $(E = 0)$ then $(PC \leftarrow PC+1)$
HLT	$rB_0:$	$S \leftarrow 0$ (S is a start-stop flip-flop)

MEMORY REFERENCE INSTRUCTIONS

Symbol	Operation Decoder	Symbolic Description
AND	D ₀	AC ← AC ∧ M[AR]
ADD	D ₁	AC ← AC + M[AR], E ← C _{out}
LDA	D ₂	AC ← M[AR]
STA	D ₃	M[AR] ← AC
BUN	D ₄	PC ← AR
BSA	D ₅	M[AR] ← PC, PC ← AR + 1
ISZ	D ₆	M[AR] ← M[AR] + 1, if M[AR] + 1 = 0 then PC ← PC+1

- The effective address of the instruction is in AR and was placed there during timing signal T₂ when I = 0, or during timing signal T₃ when I = 1
- Memory cycle is assumed to be short enough to complete in a CPU cycle
- The execution of MR instruction starts with T₄

AND to AC

D₀T₄: DR ← M[AR]Read operand

D₀T₅: AC ← AC ∧ DR, SC ← 0AND with AC

ADD to AC

D₁T₄: DR ← M[AR]Read operand

D₁T₅: AC ← AC + DR, E ← C_{out}, SC ← 0Add to AC and store carry in E

MEMORY REFERENCE INSTRUCTIONS

LDA: Load to AC

$D_2T_4:$ $DR \leftarrow M[AR]$
 $D_2T_5:$ $AC \leftarrow DR, SC \leftarrow 0$

STA: Store AC

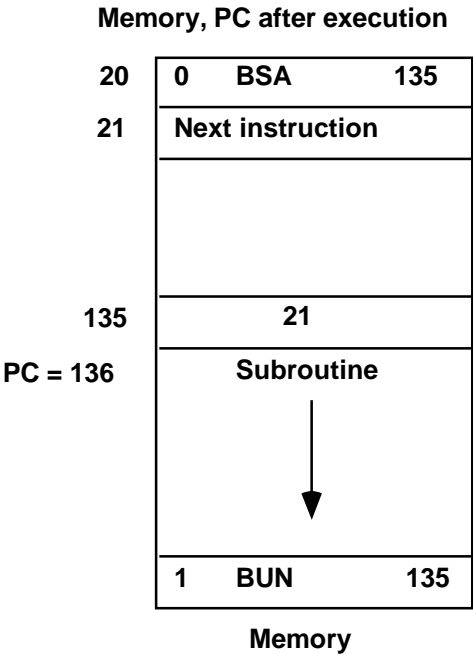
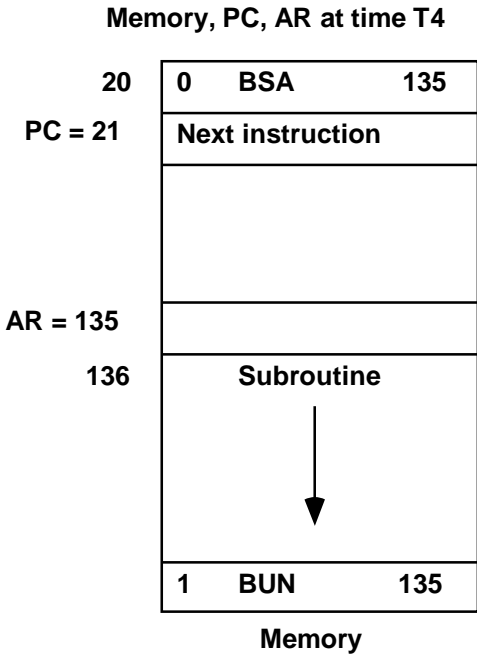
$D_3T_4:$ $M[AR] \leftarrow AC, SC \leftarrow 0$

BUN: Branch Unconditionally

$D_4T_4:$ $PC \leftarrow AR, SC \leftarrow 0$

BSA: Branch and Save Return Address

$M[AR] \leftarrow PC, PC \leftarrow AR + 1$



MEMORY REFERENCE INSTRUCTIONS

BSA:

$D_5T_4: M[AR] \leftarrow PC, AR \leftarrow AR + 1$

$D_5T_5: PC \leftarrow AR, SC \leftarrow 0$

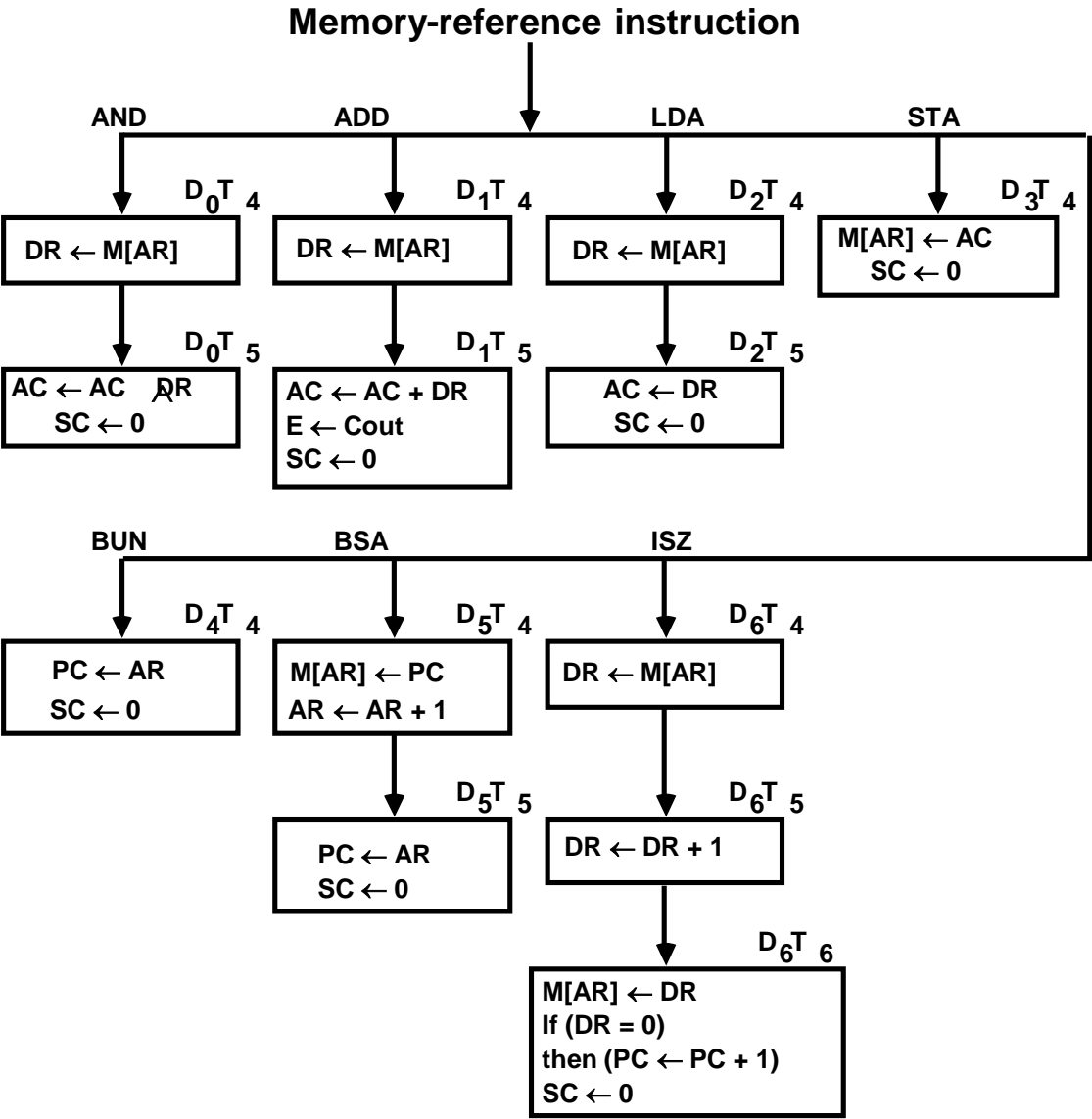
ISZ: Increment and Skip-if-Zero

$D_6T_4: DR \leftarrow M[AR]$

$D_6T_5: DR \leftarrow DR + 1$

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

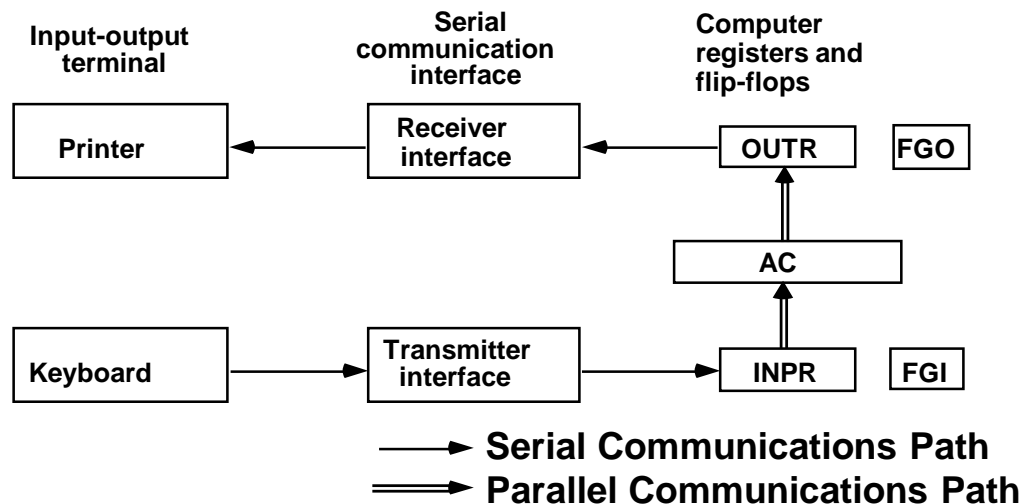
FLOWCHART FOR MEMORY REFERENCE INSTRUCTIONS



INPUT-OUTPUT AND INTERRUPT

A Terminal with a keyboard and a Printer

• Input-Output Configuration



INPR Input register - 8 bits
OUTR Output register - 8 bits
FGI Input flag - 1 bit
FGO Output flag - 1 bit
IEN Interrupt enable - 1 bit

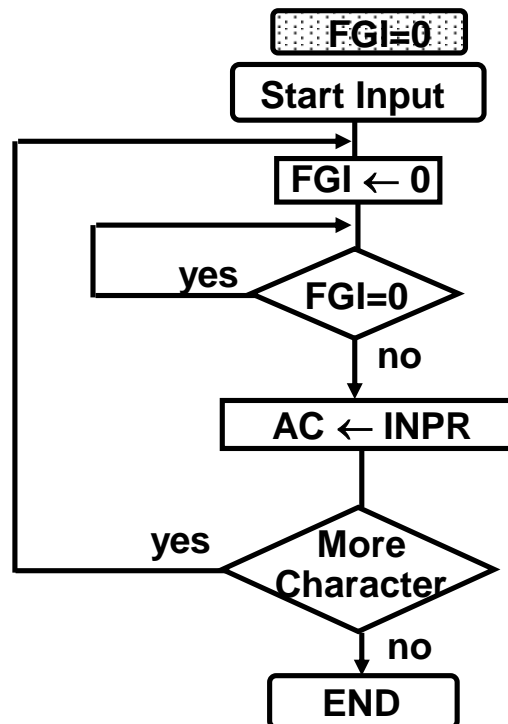
- The terminal sends and receives serial information
- The serial info. from the keyboard is shifted into INPR
- The serial info. for the printer is stored in the OUTR
- INPR and OUTR communicate with the terminal serially and with the AC in parallel.
- The flags are needed to **synchronize** the timing difference between I/O device and the computer

PROGRAM CONTROLLED DATA TRANSFER

-- CPU --

```
/* Input */      /* Initially FGI = 0 */
loop: If FGI = 0 goto loop
      AC ← INPR, FGI ← 0
```

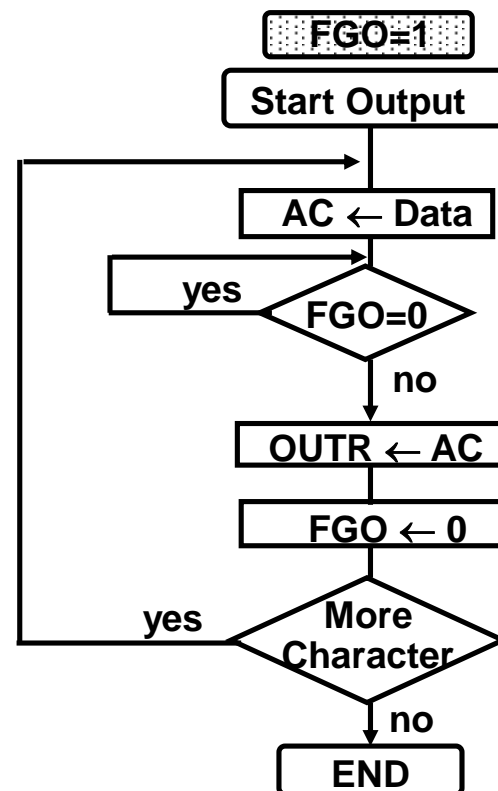
```
/* Output */     /* Initially FGO = 1 */
loop: If FGO = 0 goto loop
      OUTR ← AC, FGO ← 0
```



-- I/O Device --

```
loop: If FGI = 1 goto loop
      INPR ← new data, FGI ← 1
```

```
loop: If FGO = 1 goto loop
      consume OUTR, FGO ← 1
```



INPUT-OUTPUT INSTRUCTIONS

$D_7IT_3 = p$
 $IR(i) = B_i, i = 6, \dots, 11$

	<p>p: SC \leftarrow 0</p>	Clear SC
INP	pB₁₁: AC(0-7) \leftarrow INPR, FGI \leftarrow 0	Input char. to AC
OUT	pB₁₀: OUTR \leftarrow AC(0-7), FGO \leftarrow 0	Output char. from AC
SKI	pB₉: if(FGI = 1) then (PC \leftarrow PC + 1)	Skip on input flag
SKO	pB₈: if(FGO = 1) then (PC \leftarrow PC + 1)	Skip on output flag
ION	pB₇: IEN \leftarrow 1	Interrupt enable on
IOF	pB₆: IEN \leftarrow 0	Interrupt enable off

PROGRAM-CONTROLLED INPUT/OUTPUT

- Program-controlled I/O
 - Continuous CPU involvement
 - I/O takes valuable CPU time
 - CPU slowed down to I/O speed
 - Simple
 - Least hardware

Input

LOOP,	SKI	DEV
	BUN	LOOP
	INP	DEV

Output

LOOP,	LDA	DATA
LOP,	SKO	DEV
	BUN	LOP
	OUT	DEV

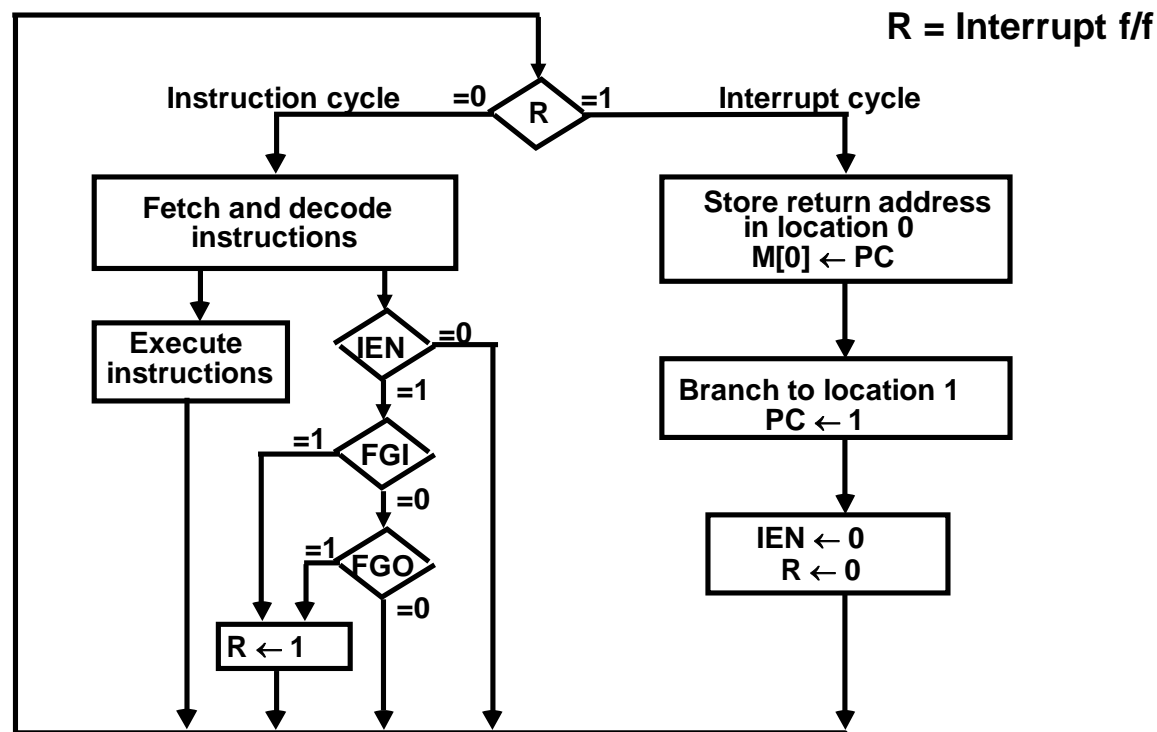
INTERRUPT INITIATED INPUT/OUTPUT

- Open communication only when some data has to be passed --> *interrupt*.
- The I/O interface, instead of the CPU, monitors the I/O device.
- When the interface finds that the I/O device is ready for data transfer, it generates an interrupt request to the CPU
- Upon detecting an interrupt, the CPU stops momentarily the task it is doing, branches to the service routine to process the data transfer, and then returns to the task it was performing.

* IEN (Interrupt-enable flip-flop)

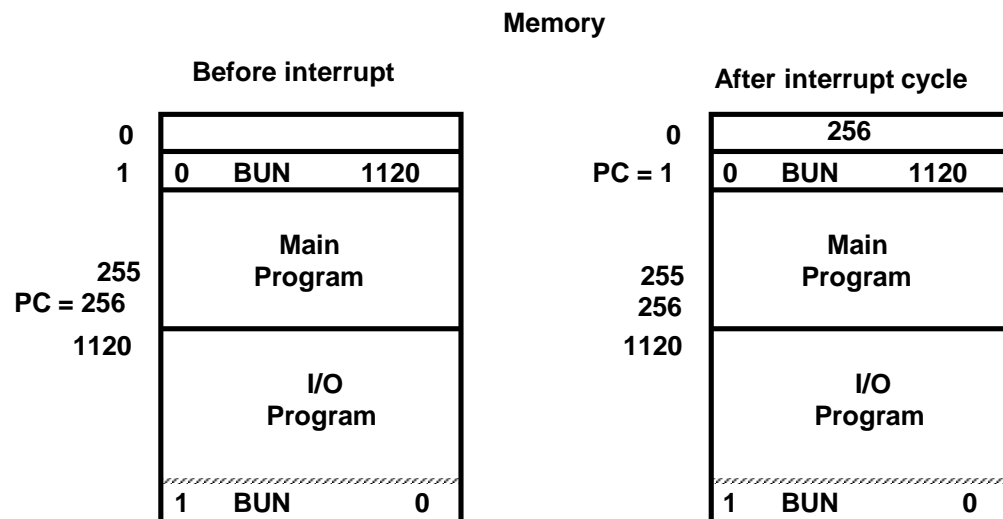
- can be set and cleared by instructions
- when cleared, the computer cannot be interrupted

FLOWCHART FOR INTERRUPT CYCLE



- The interrupt cycle is a HW implementation of a branch and save return address operation.
- At the beginning of the next instruction cycle, the instruction that is read from memory is in address 1.
- At memory address 1, the programmer must store a branch instruction that sends the control to an interrupt service routine
- The instruction that returns the control to the original program is "indirect BUN 0"

REGISTER TRANSFER OPERATIONS IN INTERRUPT CYCLE



Register Transfer Statements for Interrupt Cycle

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

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

$RT_0: AR \leftarrow 0, TR \leftarrow PC$

$RT_1: M[AR] \leftarrow TR, PC \leftarrow 0$

$RT_2: PC \leftarrow PC + 1, IEN \leftarrow 0, R \leftarrow 0, SC \leftarrow 0$

FURTHER QUESTIONS ON INTERRUPT

How can the CPU recognize the device requesting an interrupt ?

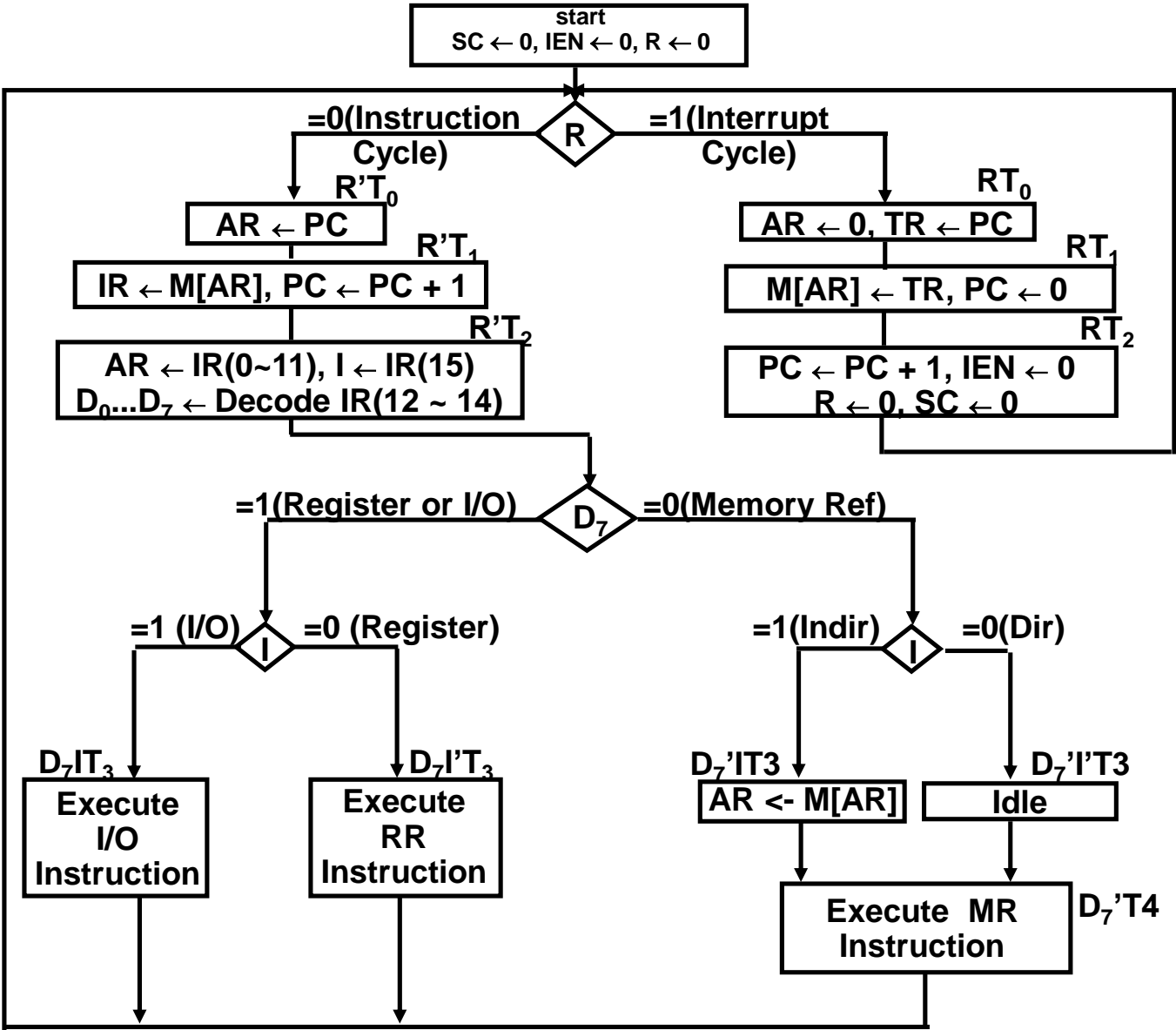
Since different devices are likely to require different interrupt service routines, how can the CPU obtain the starting address of the appropriate routine in each case ?

Should any device be allowed to interrupt the CPU while another interrupt is being serviced ?

How can the situation be handled when two or more interrupt requests occur simultaneously ?

COMPLETE COMPUTER DESCRIPTION

Flowchart of Operations



COMPLETE COMPUTER DESCRIPTION

Microoperations

Fetch	R'T ₀ :	AR ← PC
	R'T ₁ :	IR ← M[AR], PC ← PC + 1
Decode	R'T ₂ :	D0, ..., D7 ← Decode IR(12 ~ 14), AR ← IR(0 ~ 11), I ← IR(15)
Indirect Interrupt	D ₇ 'IT ₃ :	AR ← M[AR]
	T ₀ 'T ₁ 'T ₂ '(IEN)(FGI + FGO):	R ← 1
	RT ₀ :	AR ← 0, TR ← PC
	RT ₁ :	M[AR] ← TR, PC ← 0
	RT ₂ :	PC ← PC + 1, IEN ← 0, R ← 0, SC ← 0
Memory-Reference		
AND	D ₀ T ₄ :	DR ← M[AR]
	D ₀ T ₅ :	AC ← AC ∧ DR, SC ← 0
ADD	D ₁ T ₄ :	DR ← M[AR]
	D ₁ T ₅ :	AC ← AC + DR, E ← C _{out} , SC ← 0
LDA	D ₂ T ₄ :	DR ← M[AR]
	D ₂ T ₅ :	AC ← DR, SC ← 0
STA	D ₃ T ₄ :	M[AR] ← AC, SC ← 0
BUN	D ₄ T ₄ :	PC ← AR, SC ← 0
BSA	D ₅ T ₄ :	M[AR] ← PC, AR ← AR + 1
	D ₅ T ₅ :	PC ← AR, SC ← 0
ISZ	D ₆ T ₄ :	DR ← M[AR]
	D ₆ T ₅ :	DR ← DR + 1
	D ₆ T ₆ :	M[AR] ← DR, if(DR=0) then (PC ← PC + 1), SC ← 0

COMPLETE COMPUTER DESCRIPTION

Microoperations

Register-Reference

	$D_7I'T_3 = r$	(Common to all register-reference instr)
	$IR(i) = B_i$	($i = 0,1,2, \dots, 11$)
	$r:$	$SC \leftarrow 0$
CLA	$rB_{11}:$	$AC \leftarrow 0$
CLE	$rB_{10}:$	$E \leftarrow 0$
CMA	$rB_9:$	$AC \leftarrow AC'$
CME	$rB_8:$	$E \leftarrow E'$
CIR	$rB_7:$	$AC \leftarrow shr\ AC, AC(15) \leftarrow E, E \leftarrow AC(0)$
CIL	$rB_6:$	$AC \leftarrow shl\ AC, AC(0) \leftarrow E, E \leftarrow AC(15)$
INC	$rB_5:$	$AC \leftarrow AC + 1$
SPA	$rB_4:$	If($AC(15) = 0$) then ($PC \leftarrow PC + 1$)
SNA	$rB_3:$	If($AC(15) = 1$) then ($PC \leftarrow PC + 1$)
SZA	$rB_2:$	If($AC = 0$) then ($PC \leftarrow PC + 1$)
SZE	$rB_1:$	If($E = 0$) then ($PC \leftarrow PC + 1$)
HLT	$rB_0:$	$S \leftarrow 0$

Input-Output

	$D_7IT_3 = p$	(Common to all input-output instructions)
	$IR(i) = B_i$	($i = 6,7,8,9,10,11$)
	$p:$	$SC \leftarrow 0$
INP	$pB_{11}:$	$AC(0-7) \leftarrow INPR, FGI \leftarrow 0$
OUT	$pB_{10}:$	$OUTR \leftarrow AC(0-7), FGO \leftarrow 0$
SKI	$pB_9:$	If($FGI = 1$) then ($PC \leftarrow PC + 1$)
SKO	$pB_8:$	If($FGO = 1$) then ($PC \leftarrow PC + 1$)
ION	$pB_7:$	$IEN \leftarrow 1$
IOF	$pB_6:$	$IEN \leftarrow 0$

DESIGN OF BASIC COMPUTER(BC)

Hardware Components of BC

A memory unit: 4096 x 16.

Registers:

AR, PC, DR, AC, IR, TR, OUTR, INPR, and SC

Flip-Flops(Status):

I, S, E, R, IEN, FGI, and FGO

Decoders: a 3x8 Opcode decoder
a 4x16 timing decoder

Common bus: 16 bits

Control logic gates:

Adder and Logic circuit: Connected to AC

Control Logic Gates

- Input Controls of the nine registers
- Read and Write Controls of memory
- Set, Clear, or Complement Controls of the flip-flops
- S_2, S_1, S_0 Controls to select a register for the bus
- AC, and Adder and Logic circuit

CONTROL OF REGISTERS AND MEMORY

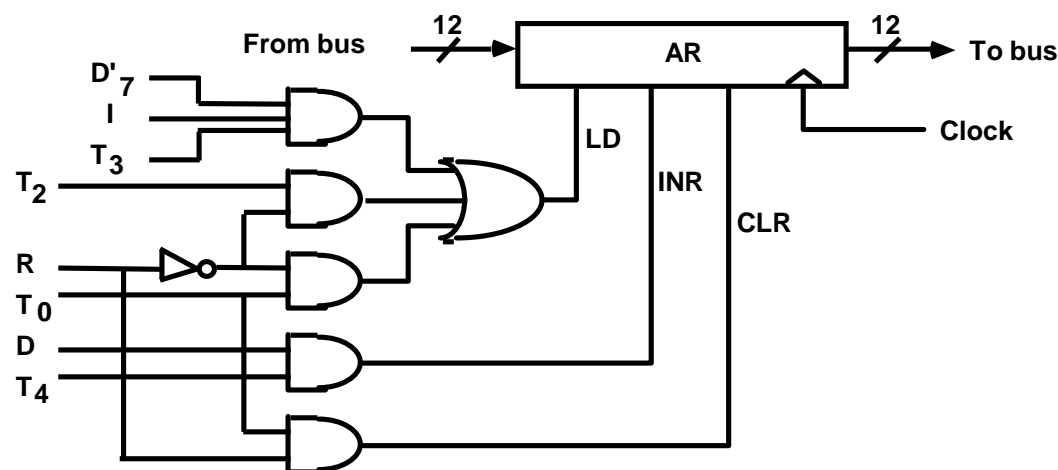
Address Register; AR

Scan all of the register transfer statements that change the content of AR:

$R'T_0$:	$AR \leftarrow PC$	$LD(AR)$
$R'T_2$:	$AR \leftarrow IR(0-11)$	$LD(AR)$
D'_7IT_3 :	$AR \leftarrow M[AR]$	$LD(AR)$
RT_0 :	$AR \leftarrow 0$	$CLR(AR)$
D_5T_4 :	$AR \leftarrow AR + 1$	$INR(AR)$



$LD(AR) = R'T_0 + R'T_2 + D'_7IT_3$
$CLR(AR) = RT_0$
$INR(AR) = D_5T_4$



CONTROL OF FLAGS

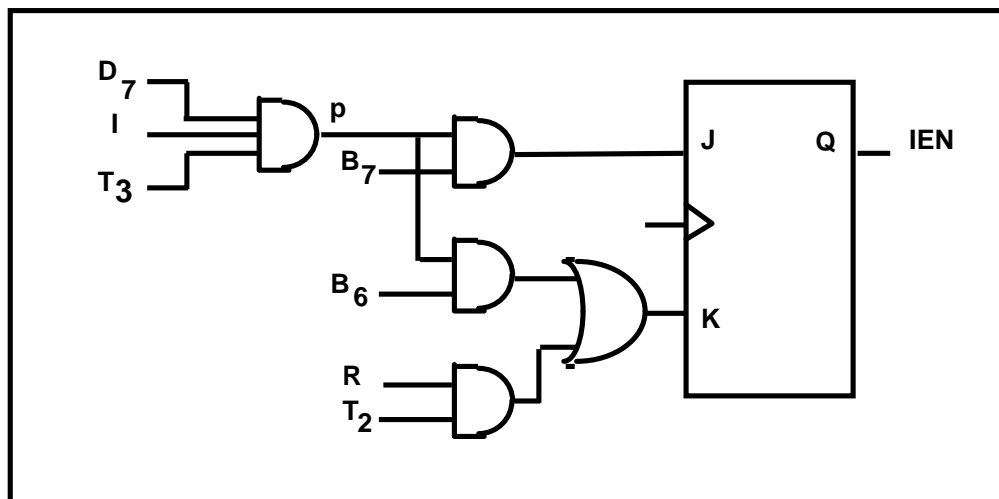
IEN: Interrupt Enable Flag

pB_7 : $IEN \leftarrow 1$ (I/O Instruction)

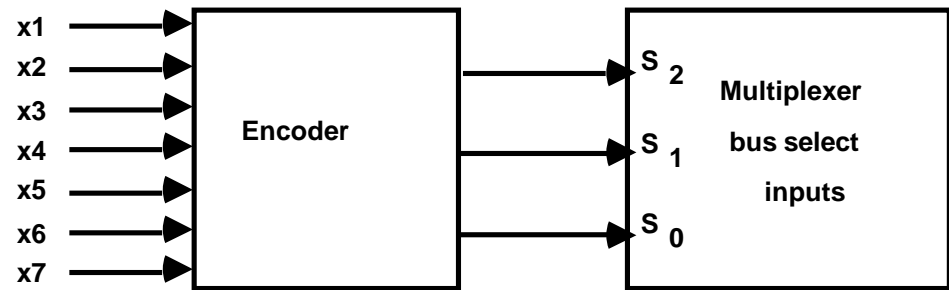
pB_6 : $IEN \leftarrow 0$ (I/O Instruction)

RT_2 : $IEN \leftarrow 0$ (Interrupt)

$p = D_7IT_3$ (Input/Output Instruction)

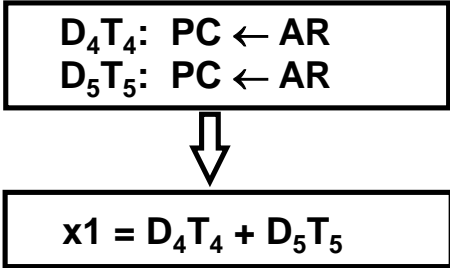


CONTROL OF COMMON BUS



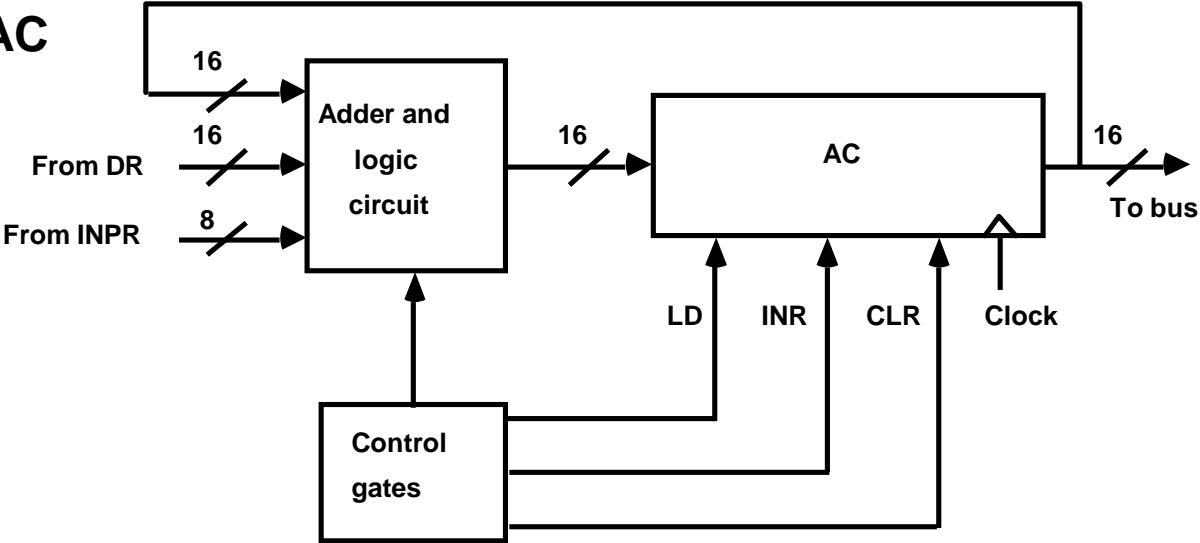
x1	x2	x3	x4	x5	x6	x7	S2	S1	S0	selected register
0	0	0	0	0	0	0	0	0	none	
1	0	0	0	0	0	0	0	1	AR	
0	1	0	0	0	0	0	1	0	PC	
0	0	1	0	0	0	0	1	1	DR	
0	0	0	1	0	0	0	0	0	AC	
0	0	0	0	1	0	0	0	1	IR	
0	0	0	0	0	1	0	1	0	TR	
0	0	0	0	0	0	1	1	1	Memory	

For AR



DESIGN OF ACCUMULATOR LOGIC

Circuits associated with AC

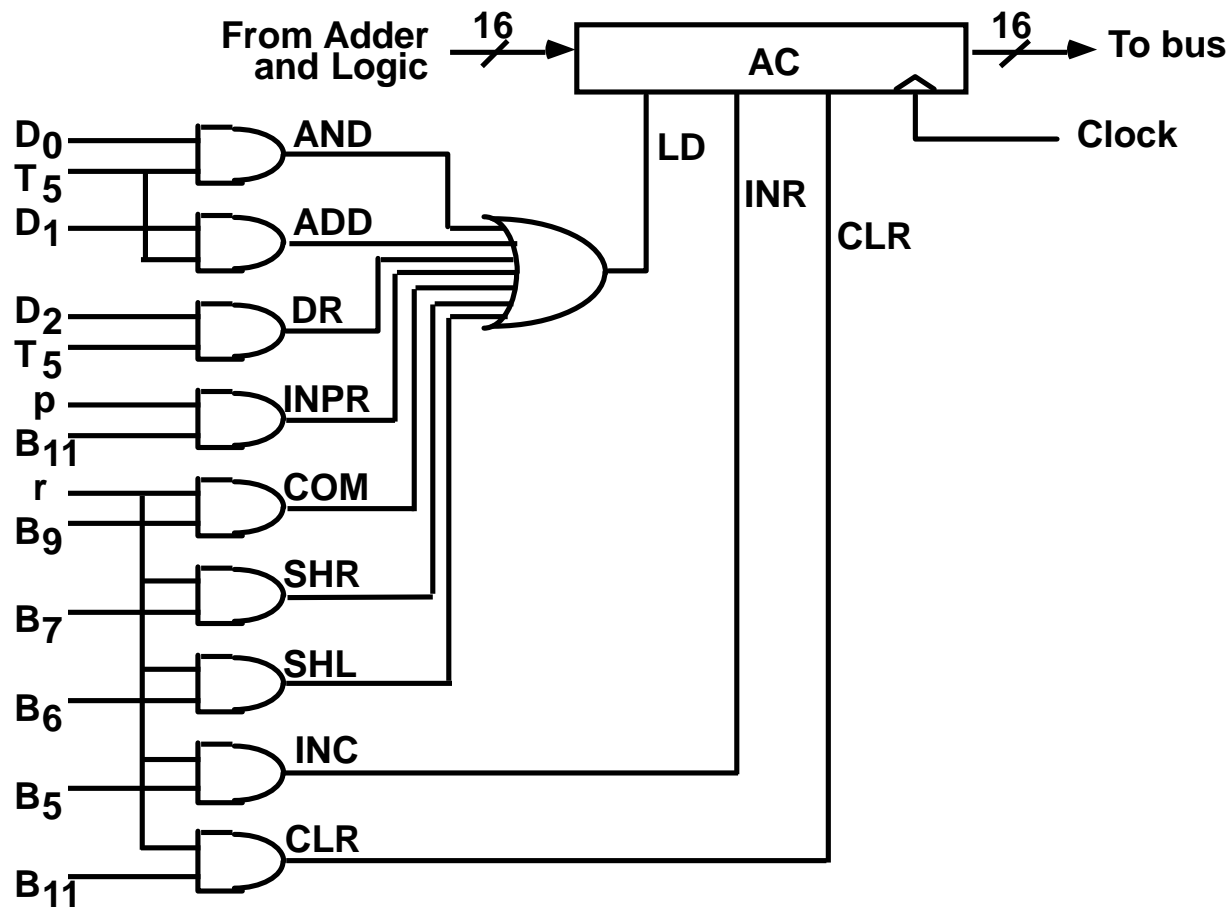


All the statements that change the content of AC

$D_0T_5:$	$AC \leftarrow AC \wedge DR$	AND with DR
$D_1T_5:$	$AC \leftarrow AC + DR$	Add with DR
$D_2T_5:$	$AC \leftarrow DR$	Transfer from DR
$pB_{11}:$	$AC(0-7) \leftarrow INPR$	Transfer from INPR
$rB_9:$	$AC \leftarrow AC'$	Complement
$rB_7:$	$AC \leftarrow shr\ AC, AC(15) \leftarrow E$	Shift right
$rB_6:$	$AC \leftarrow shl\ AC, AC(0) \leftarrow E$	Shift left
$rB_{11}:$	$AC \leftarrow 0$	Clear
$rB_5:$	$AC \leftarrow AC + 1$	Increment

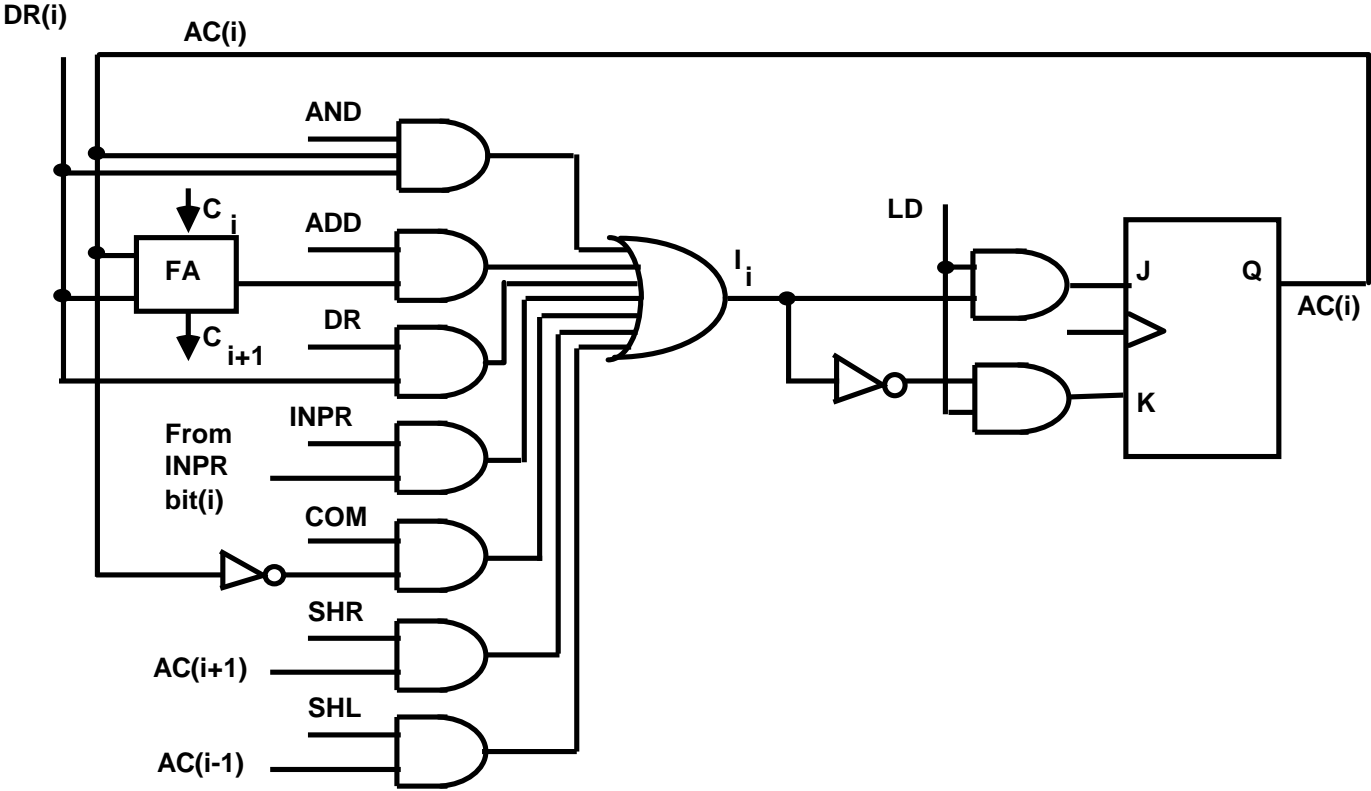
CONTROL OF AC REGISTER

Gate structures for controlling the LD, INR, and CLR of AC



ALU (ADDER AND LOGIC CIRCUIT)

One stage of Adder and Logic circuit



PROGRAMMING THE BASIC COMPUTER

Introduction

Machine Language

Assembly Language

Assembler

Program Loops

Programming Arithmetic and Logic Operations

Subroutines

Input-Output Programming

INTRODUCTION

Those concerned with computer architecture should have a knowledge of both hardware and software because the two branches influence each other.

Instruction Set of the *Basic Computer*

Symbol	Hexa code	Description
AND	0 or 8	AND M to AC
ADD	1 or 9	Add M to AC, carry to E
LDA	2 or A	Load AC from M
STA	3 or B	Store AC in M
BUN	4 or C	Branch unconditionally to m
BSA	5 or D	Save return address in m and branch to m+1
ISZ	6 or E	Increment M and skip if zero
CLA	7800	Clear AC
CLE	7400	Clear E
CMA	7200	Complement AC
CME	7100	Complement E
CIR	7080	Circulate right E and AC
CIL	7040	Circulate left E and AC
INC	7020	Increment AC, carry to E
SPA	7010	Skip if AC is positive
SNA	7008	Skip if AC is negative
SZA	7004	Skip if AC is zero
SZE	7002	Skip if E is zero
HLT	7001	Halt computer
INP	F800	Input information and clear flag
OUT	F400	Output information and clear flag
SKI	F200	Skip if input flag is on
SKO	F100	Skip if output flag is on
ION	F080	Turn interrupt on
IOF	F040	Turn interrupt off

m: effective address
M: memory word (operand)
found at m

MACHINE LANGUAGE

- **Program**
A list of instructions or statements for directing the computer to perform a required data processing task
- **Various types of programming languages**
 - **Hierarchy of programming languages**
 - **Machine-language**
 - **Binary code**
 - **Octal or hexadecimal code**
 - **Assembly-language** (Assembler)
 - **Symbolic code**
 - **High-level language** (Compiler)

COMPARISON OF PROGRAMMING LANGUAGES

• Binary Program to Add Two Numbers

Location	Instruction Code
0	0010 0000 0000 0100
1	0001 0000 0000 0101
10	0011 0000 0000 0110
11	0111 0000 0000 0001
100	0000 0000 0101 0011
101	1111 1111 1110 1001
110	0000 0000 0000 0000

• Hexa program

Location	Instruction
000	2004
001	1005
002	3006
003	7001
004	0053
005	FFE9
006	0000

• Program with Symbolic OP-Code

Location	Instruction	Comments
000	LDA 004	Load 1st operand into AC
001	ADD 005	Add 2nd operand to AC
002	STA 006	Store sum in location 006
003	HLT	Halt computer
004	0053	1st operand
005	FFE9	2nd operand (negative)
006	0000	Store sum here

• Assembly-Language Program

	ORG	0	/Origin of program is location 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

• Fortran Program

INTEGER A, B, C
DATA A,83 / B,-23
C = A + B
END

ASSEMBLY LANGUAGE

Syntax of the BC assembly language

Each line is arranged in three columns called fields

Label field

- May be empty or may specify a symbolic address consists of up to 3 characters
- Terminated by a comma

Instruction field

- Specifies a machine or a pseudo instruction
- May specify one of
 - * Memory reference instr. (MRI)
 - MRI consists of two or three symbols separated by spaces.
 - ADD OPR (direct address MRI)
 - ADD PTR I (indirect address MRI)
 - * Register reference or input-output instr.
 - Non-MRI does not have an address part
 - * Pseudo instr. with or without an operand
 - Symbolic address used in the instruction field must be defined somewhere as a label

Comment field

- May be empty or may include a comment

PSEUDO-INSTRUCTIONS

ORG N

Hexadecimal number N is the memory loc.
for the instruction or operand listed in the following line

END

Denotes the end of symbolic program

DEC N

Signed decimal number N to be converted to the binary

HEX N

Hexadecimal number N to be converted to the binary

Example: Assembly language program to subtract two numbers

	ORG 100	/ Origin of program is location 100
	LDA SUB	/ Load subtrahend to AC
	CMA	/ Complement AC
	INC	/ Increment AC
	ADD MIN	/ Add minuend to AC
	STA DIF	/ Store difference
	HLT	/ Halt computer
MIN,	DEC 83	/ Minuend
SUB,	DEC -23	/ Subtrahend
DIF,	HEX 0	/ Difference stored here
	END	/ End of symbolic program

TRANSLATION TO BINARY

Hexadecimal Code		Symbolic Program
Location	Content	
100	2107	ORG 100
101	7200	LDA SUB
102	7020	CMA
103	1106	INC
104	3108	ADD MIN
105	7001	STA DIF
106	0053	HLT
107	FFE9	MIN, DEC 83
108	0000	SUB, DEC -23
		DIF, HEX 0
		END

ASSEMBLER - FIRST PASS -

Assembler

Source Program - Symbolic Assembly Language Program

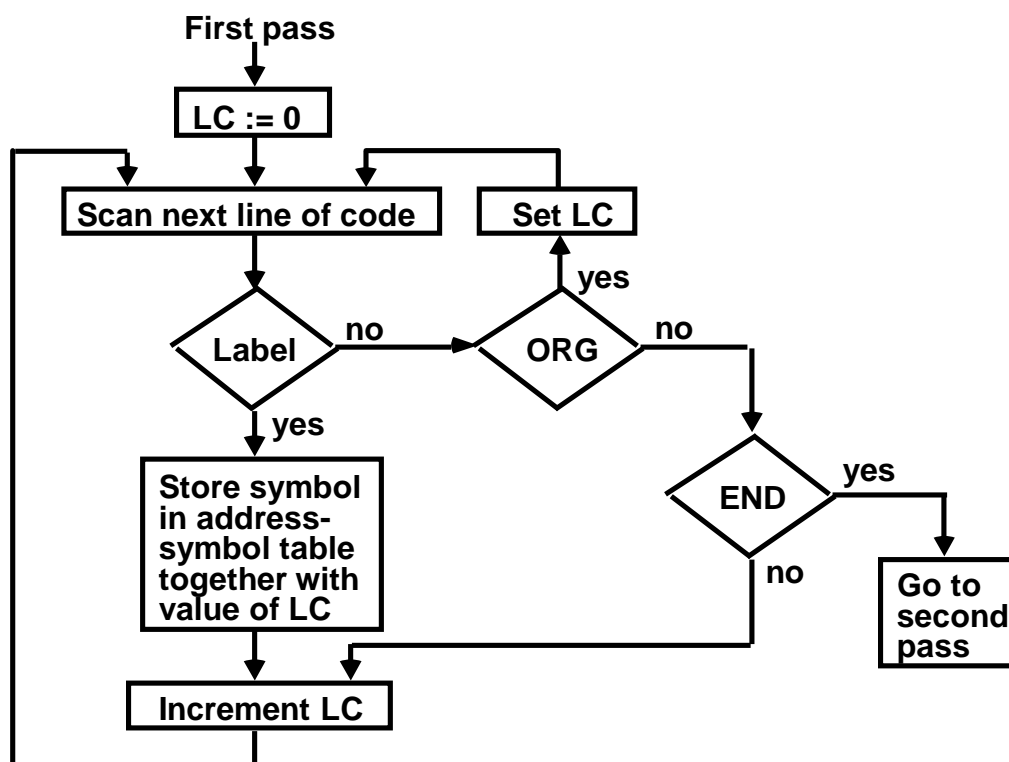
Object Program - Binary Machine Language Program

Two pass assembler

1st pass: generates a table that correlates all user defined (address) symbols with their binary equivalent value

2nd pass: binary translation

First pass

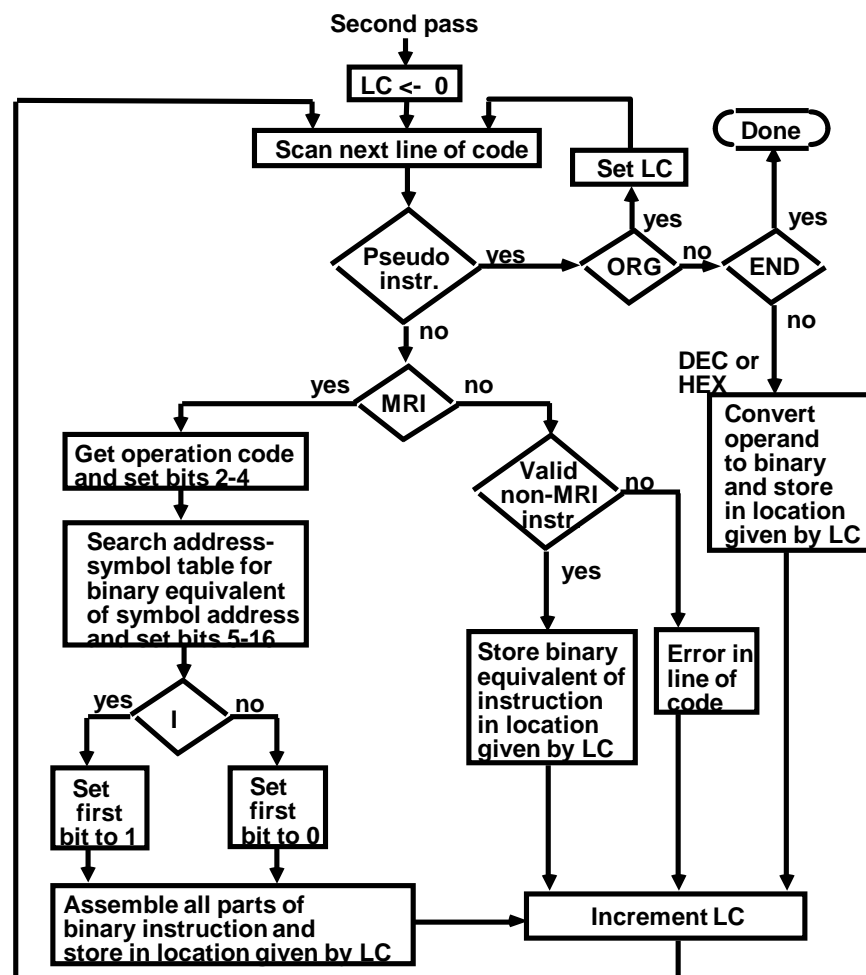


ASSEMBLER - SECOND PASS -

Second Pass

Machine instructions are translated by means of table-lookup procedures;

- (1. Pseudo-Instruction Table, 2. MRI Table, 3. Non-MRI Table
4. Address Symbol Table)



PROGRAM LOOPS

Loop: A sequence of instructions that are executed many times,
each with a different set of data
Fortran program to add 100 numbers:

```
DIMENSION A(100)
INTEGER SUM, A
SUM = 0
DO 3 J = 1, 100
3 SUM = SUM + A(J)
```

Assembly-language program to add 100 numbers:

	ORG 100	/ Origin of program is HEX 100
	LDA ADS	/ Load first address of operand
	STA PTR	/ Store in pointer
	LDA NBR	/ Load -100
	STA CTR	/ Store in counter
	CLA	/ Clear AC
LOP,	ADD PTR I	/ Add an operand to AC
	ISZ PTR	/ Increment pointer
	ISZ CTR	/ Increment counter
	BUN LOP	/ Repeat loop again
	STA SUM	/ Store sum
	HLT	/ Halt
ADS,	HEX 150	/ First address of operands
PTR,	HEX 0	/ Reserved for a pointer
NBR,	DEC -100	/ Initial value for a counter
CTR,	HEX 0	/ Reserved for a counter
SUM,	HEX 0	/ Sum is stored here
	ORG 150	/ Origin of operands is HEX 150
	DEC 75	/ First operand
	:	
	:	
	DEC 23	/ Last operand
	END	/ End of symbolic program

PROGRAMMING ARITHMETIC AND LOGIC OPERATIONS

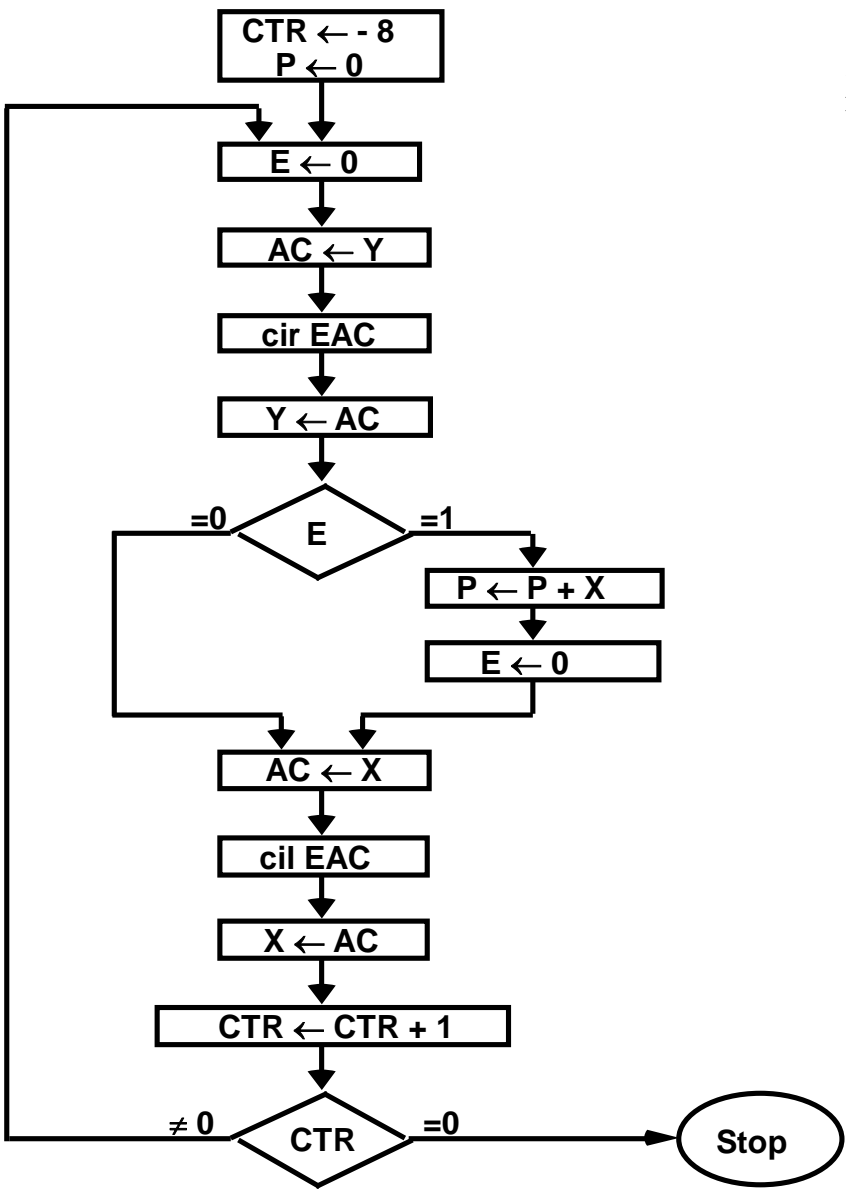
Implementation of Arithmetic and Logic Operations

- Software Implementation
 - Implementation of an operation with a program using machine instruction set
 - Usually when the operation is not included in the instruction set
- Hardware Implementation
 - Implementation of an operation in a computer with one machine instruction

Software Implementation example:

- * Multiplication
 - For simplicity, unsigned positive numbers
 - 8-bit numbers -> 16-bit product

FLOWCHART OF A PROGRAM - Multiplication -



X holds the multiplicand
Y holds the multiplier
P holds the product

Example with four significant digits

X =	0000 1111		P
Y =	<u>0000 1011</u>		<u>0000 0000</u>
	0000 1111		0000 1111
	0001 1110		0010 1101
	0000 0000		0010 1101
	<u>0111 1000</u>		<u>1010 0101</u>
	1010 0101		

ASSEMBLY LANGUAGE PROGRAM - Multiplication -

```

      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; goto ONE
      BUN ZRO      / Bit is zero; goto 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

```

ASSEMBLY LANGUAGE PROGRAM

- Double Precision Addition -

LDA	AL	/ Load A low
ADD	BL	/ Add B low, carry in E
STA	CL	/ Store in C low
CLA		/ Clear AC
CIL		/ Circulate to bring carry into AC(16)
ADD	AH	/ Add A high and carry
ADD	BH	/ Add B high
STA	CH	/ Store in C high
HLT		

ASSEMBLY LANGUAGE PROGRAM

- Logic and Shift Operations -

• Logic operations

- BC instructions : AND, CMA, CLA
- Program for OR operation

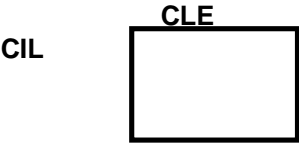
LDA	A	/ Load 1st operand
CMA		/ Complement to get A'
STA	TMP	/ Store in a temporary location
LDA	B	/ Load 2nd operand B
CMA		/ Complement to get B'
AND	TMP	/ AND with A' to get A' AND B'
CMA		/ Complement again to get A OR B

• Shift operations - BC has *Circular Shift* only

- Logical shift-right operation



- Logical shift-left operation



- Arithmetic right-shift operation

CLE	/ Clear E to 0
SPA	/ Skip if AC is positive
CME	/ AC is negative
CIR	/ Circulate E and AC

SUBROUTINES

Subroutine

- A set of common instructions that can be used in a program many times.
- Subroutine *linkage* : a procedure for branching to a subroutine and returning to the main program

Example

Loc.			
		ORG 100	/ Main program
100		LDA X	/ Load X
101		BSA SH4	/ Branch to subroutine
102		STA X	/ Store shifted number
103		LDA Y	/ Load Y
104		BSA SH4	/ Branch to subroutine again
105		STA Y	/ Store shifted number
106		HLT	
107	X,	HEX 1234	
108	Y,	HEX 4321	
			/ Subroutine to shift left 4 times
109	SH4,	HEX 0	/ Store return address here
10A		CIL	/ Circulate left once
10B		CIL	
10C		CIL	
10D		CIL	/ Circulate left fourth time
10E		AND MSK	/ Set AC(13-16) to zero
10F		BUN SH4 I	/ Return to main program
110	MSK,	HEX FFF0	/ Mask operand
		END	

SUBROUTINE PARAMETERS AND DATA LINKAGE

Linkage of Parameters and Data between the Main Program and a Subroutine

- via Registers
- via Memory locations
-

Example: Subroutine performing *LOGICAL OR operation*; Need two parameters

Loc.			
		ORG 200	
200		LDA X	/ Load 1st operand into AC
201		BSA OR	/ Branch to subroutine OR
202		HEX 3AF6	/ 2nd operand stored here
203		STA Y	/ Subroutine returns here
204		HLT	
205	X,	HEX 7B95	/ 1st operand stored here
206	Y,	HEX 0	/ Result stored here
207	OR,	HEX 0	/ Subroutine OR
208		CMA	/ Complement 1st operand
209		STA TMP	/ Store in temporary location
20A		LDA OR I	/ Load 2nd operand
20B		CMA	/ Complement 2nd operand
20C		AND TMP	/ AND complemented 1st operand
20D		CMA	/ Complement again to get OR
20E		ISZ OR	/ Increment return address
20F		BUN OR I	/ Return to main program
210	TMP,	HEX 0	/ Temporary storage
		END	

SUBROUTINE - Moving a Block of Data -

		/ Main program
	BSA MVE	/ Branch to subroutine
	HEX 100	/ 1st address of source data
	HEX 200	/ 1st address of destination data
	DEC -16	/ Number of items to move
	HLT	
MVE,	HEX 0	/ Subroutine MVE
	LDA MVE I	/ Bring address of source
	STA PT1	/ Store in 1st pointer
	ISZ MVE	/ Increment return address
	LDA MVE I	/ Bring address of destination
	STA PT2	/ Store in 2nd pointer
	ISZ MVE	/ Increment return address
	LDA MVE I	/ Bring number of items
	STA CTR	/ Store in counter
	ISZ MVE	/ Increment return address
LOP,	LDA PT1 I	/ Load source item
	STA PT2 I	/ Store in destination
	ISZ PT1	/ Increment source pointer
	ISZ PT2	/ Increment destination pointer
	ISZ CTR	/ Increment counter
	BUN LOP	/ Repeat 16 times
	BUN MVE I	/ Return to main program
PT1,	--	
PT2,	--	
CTR,	--	

• Fortran subroutine

```
SUBROUTINE MVE (SOURCE, DEST, N)
  DIMENSION SOURCE(N), DEST(N)
  DO 20 I = 1, N
20  DEST(I) = SOURCE(I)
  RETURN
END
```

INPUT OUTPUT PROGRAM

Program to Input one Character(Byte)

CIF,	SKI	/ Check input flag
	BUN CIF	/ Flag=0, branch to check again
	INP	/ Flag=1, input character
	OUT	/ Display to ensure correctness
	STA CHR	/ Store character
	HLT	
CHR,	--	/ Store character here

Program to Output a Character

	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 0057	/ Character is "W"

CHARACTER MANIPULATION

Subroutine to Input 2 Characters and pack into a word

IN2,	--	/ Subroutine entry
FST,	SKI	
	BUN FST	
	INP	/ Input 1st character
	OUT	
	BSA SH4	/ Logical Shift left 4 bits
	BSA SH4	/ 4 more bits
SCD,	SKI	
	BUN SCD	
	INP	/ Input 2nd character
	OUT	
	BUN IN2 I	/ Return

PROGRAM INTERRUPT

Tasks of Interrupt Service Routine

- Save the Status of CPU
 Contents of processor registers and Flags
- Identify the source of Interrupt
 Check which flag is set
- Service the device whose flag is set
 (Input Output Subroutine)
- Restore contents of processor registers and flags
- Turn the interrupt facility on
- Return to the running program
 Load PC of the interrupted program

INTERRUPT SERVICE ROUTINE

Loc.			
0	ZRO,	-	/ Return address stored here
1		BUN SRV	/ Branch to service routine
100		CLA	/ Portion of running program
101		ION	/ Turn on interrupt facility
102		LDA X	
103		ADD Y	/ Interrupt occurs here
104		STA Z	/ Program returns here after interrupt
200	SRV,		/ Interrupt service routine
		STA SAC	/ Store content of AC
		CIR	/ Move E into AC(1)
		STA SE	/ Store content of E
		SKI	/ Check input flag
		BUN NXT	/ Flag is off, check next flag
		INP	/ Flag is on, input character
		OUT	/ Print character
		STA PT1 I	/ Store it in input buffer
		ISZ PT1	/ Increment input pointer
	NXT,	SKO	/ Check output flag
		BUN EXT	/ Flag is off, exit
		LDA PT2 I	/ Load character from output buffer
		OUT	/ Output character
		ISZ PT2	/ Increment output pointer
	EXT,	LDA SE	/ Restore value of AC(1)
		CIL	/ Shift it to E
		LDA SAC	/ Restore content of AC
		ION	/ Turn interrupt on
		BUN ZRO I	/ Return to running program
	SAC,	-	/ AC is stored here
	SE,	-	/ E is stored here
	PT1,	-	/ Pointer of input buffer
	PT2,	-	/ Pointer of output buffer

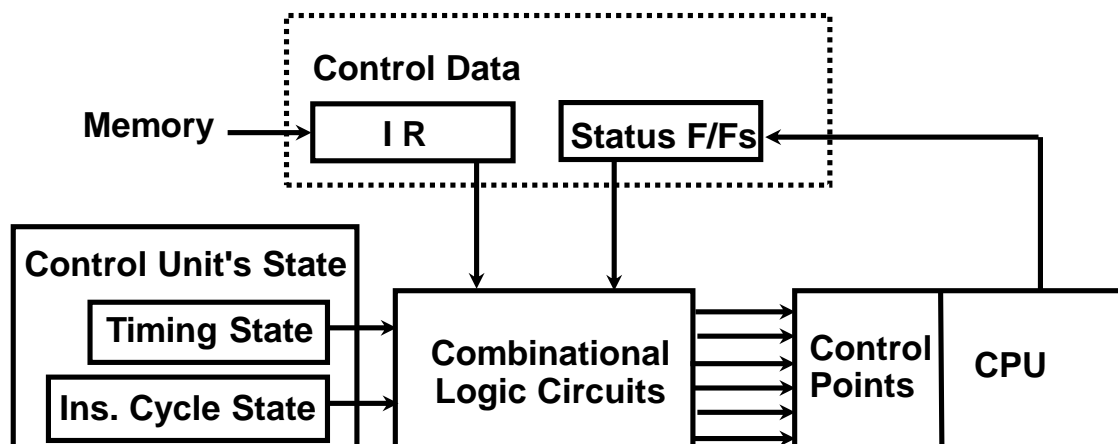
MICROPROGRAMMED CONTROL

- **Control Memory**
- **Sequencing Microinstructions**
- **Microprogram Example**
- **Design of Control Unit**
- **Microinstruction Format**
- **Nanostorage and Nanoprogram**

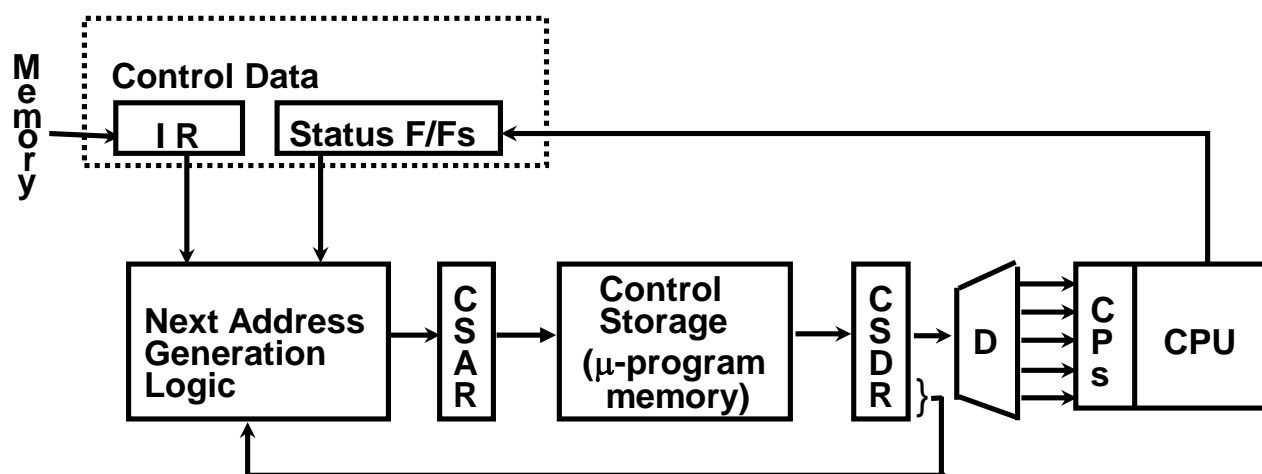
COMPARISON OF CONTROL UNIT IMPLEMENTATIONS

Control Unit Implementation

Combinational Logic Circuits (Hard-wired)



Microprogram



TERMINOLOGY

Microprogram

- Program stored in memory that generates all the control signals required to execute the instruction set correctly
- Consists of microinstructions

Microinstruction

- Contains a control word and a sequencing word
 - Control Word - All the control information required for one clock cycle
 - Sequencing Word - Information needed to decide the next microinstruction address
- Vocabulary to write a microprogram

Control Memory(Control Storage: CS)

- Storage in the microprogrammed control unit to store the microprogram

Writeable Control Memory(Writeable Control Storage:WCS)

- CS whose contents can be modified
 - > Allows the microprogram can be changed
 - > Instruction set can be changed or modified

Dynamic Microprogramming

- Computer system whose control unit is implemented with a microprogram in WCS
- Microprogram can be changed by a systems programmer or a user

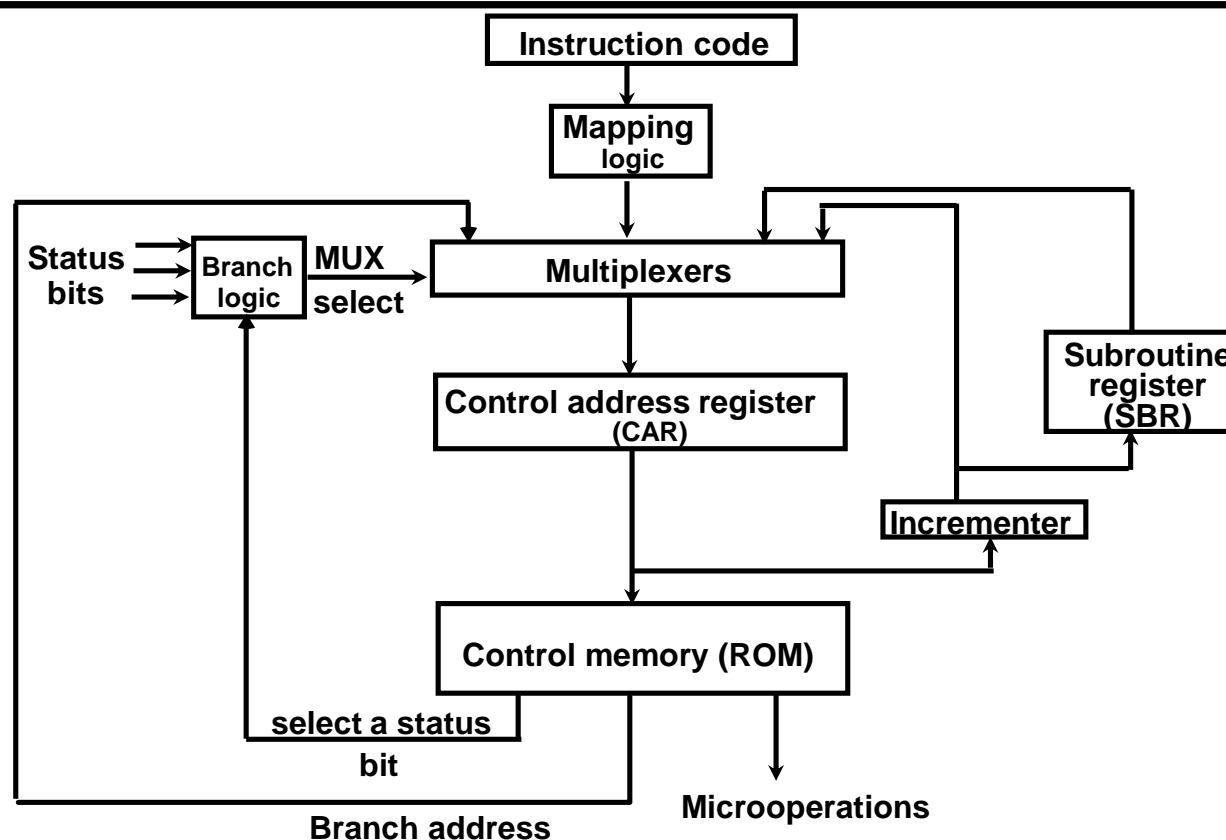
TERMINOLOGY

Sequencer (Microprogram Sequencer)

A Microprogram Control Unit that determines the Microinstruction Address to be executed in the next clock cycle

- In-line Sequencing**
- Branch**
- Conditional Branch**
- Subroutine**
- Loop**
- Instruction OP-code mapping**

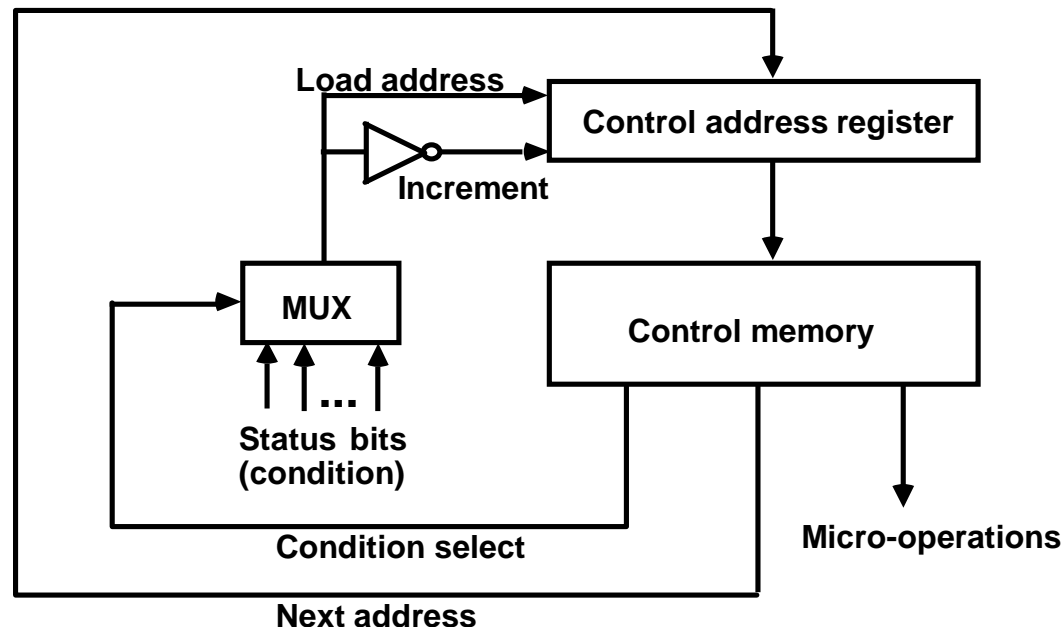
MICROINSTRUCTION SEQUENCING



Sequencing Capabilities Required in a Control Storage

- Incrementing of the control address register
- Unconditional and conditional branches
- A mapping process from the bits of the machine instruction to an address for control memory
- A facility for subroutine call and return

CONDITIONAL BRANCH



Conditional Branch

If *Condition* is true, then *Branch* (address from the next address field of the current microinstruction)
else *Fall Through*

Conditions to Test: O(overflow), N(negative),
Z(zero), C(carry), etc.

Unconditional Branch

Fixing the value of one status bit at the input of the multiplexer to 1

MAPPING OF INSTRUCTIONS

Direct Mapping

OP-codes of Instructions

ADD 0000
AND 0001
LDA 0010
STA 0011
BUN 0100

Address

0000
0001
0010
0011
0100

ADD Routine
AND Routine
LDA Routine
STA Routine
BUN Routine

Control
Storage

Mapping Bits

10 xxxx 010

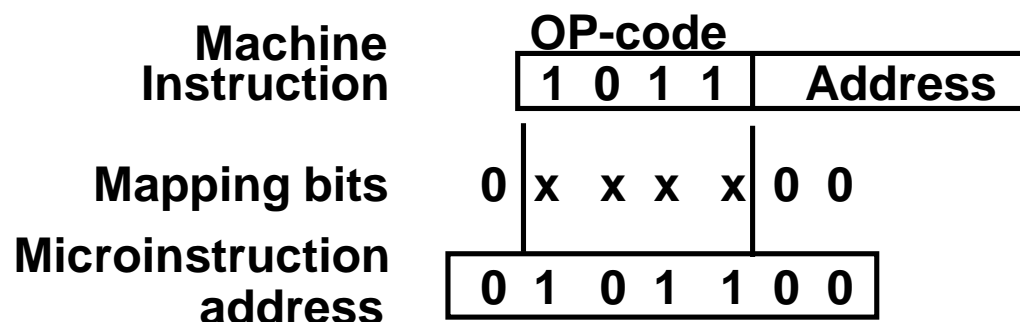
Address

10 0000 010
10 0001 010
10 0010 010
10 0011 010
10 0100 010

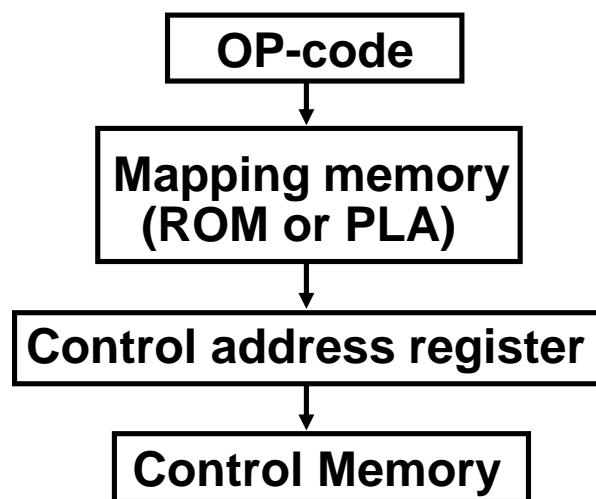
ADD Routine
⋮
AND Routine
⋮
LDA Routine
⋮
STA Routine
⋮
BUN Routine
⋮

MAPPING OF INSTRUCTIONS TO MICROROUTINES

Mapping from the OP-code of an instruction to the address of the Microinstruction which is the starting microinstruction of its execution microprogram

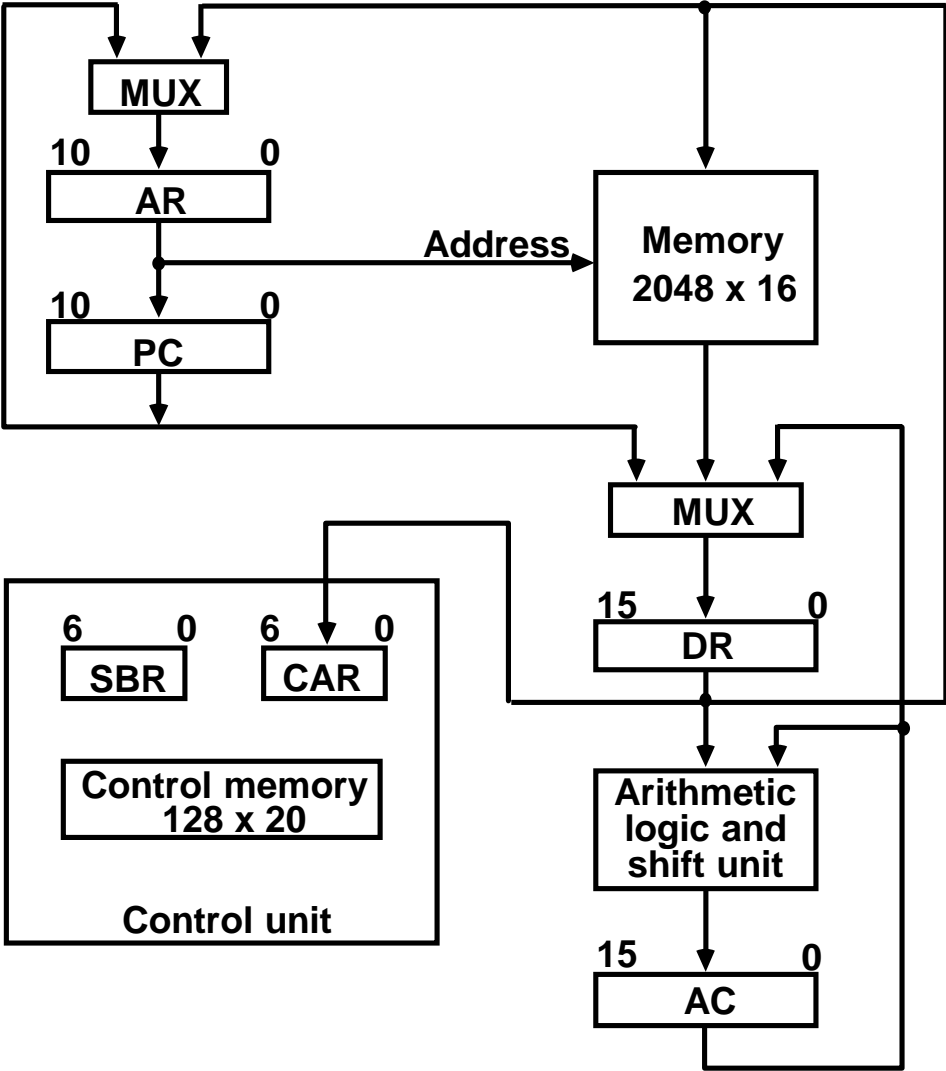


Mapping function implemented by ROM or PLA



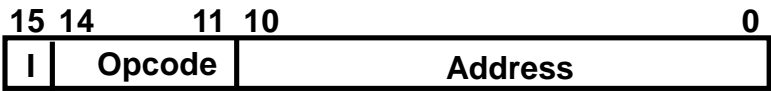
MICROPROGRAM EXAMPLE

Computer Configuration



MACHINE INSTRUCTION FORMAT

Machine instruction format

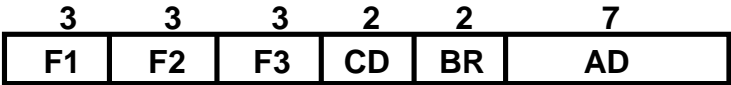


Sample machine instructions

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

EA is the effective address

Microinstruction Format



F1, F2, F3: Microoperation fields
CD: Condition for branching
BR: Branch field
AD: Address field

MICROINSTRUCTION FIELD DESCRIPTIONS - F1,F2,F3

F1	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC + DR$	ADD
010	$AC \leftarrow 0$	CLRAC
011	$AC \leftarrow AC + 1$	INCAC
100	$AC \leftarrow DR$	DRTAC
101	$AR \leftarrow DR(0-10)$	DRTAR
110	$AR \leftarrow PC$	PCTAR
111	$M[AR] \leftarrow DR$	WRITE

F2	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC - DR$	SUB
010	$AC \leftarrow AC \vee DR$	OR
011	$AC \leftarrow AC \wedge DR$	AND
100	$DR \leftarrow M[AR]$	READ
101	$DR \leftarrow AC$	ACTDR
110	$DR \leftarrow DR + 1$	INCDR
111	$DR(0-10) \leftarrow PC$	PCTDR

F3	Microoperation	Symbol
000	None	NOP
001	$AC \leftarrow AC \oplus DR$	XOR
010	$AC \leftarrow AC'$	COM
011	$AC \leftarrow shl\ AC$	SHL
100	$AC \leftarrow shr\ AC$	SHR
101	$PC \leftarrow PC + 1$	INCPC
110	$PC \leftarrow AR$	ARTPC
111	Reserved	

MICROINSTRUCTION FIELD DESCRIPTIONS - CD, BR

CD	Condition	Symbol	Comments
00	Always = 1	U	Unconditional branch
01	DR(15)	I	Indirect address bit
10	AC(15)	S	Sign bit of AC
11	AC = 0	Z	Zero value in AC

BR	Symbol	Function
00	JMP	$CAR \leftarrow AD$ if condition = 1 $CAR \leftarrow CAR + 1$ if condition = 0
01	CALL	$CAR \leftarrow AD$, $SBR \leftarrow CAR + 1$ if condition = 1 $CAR \leftarrow CAR + 1$ if condition = 0
10	RET	$CAR \leftarrow SBR$ (Return from subroutine)
11	MAP	$CAR(2-5) \leftarrow DR(11-14)$, $CAR(0,1,6) \leftarrow 0$

SYMBOLIC MICROINSTRUCTIONS

- Symbols are used in microinstructions as in assembly language
- A symbolic microprogram can be translated into its binary equivalent by a microprogram assembler.

Sample Format

five fields: label; micro-ops; CD; BR; AD

Label: may be empty or may specify a symbolic
 address terminated with a colon

Micro-ops: consists of one, two, or three symbols
 separated by commas

CD: one of {U, I, S, Z}, where U: Unconditional Branch
 I: Indirect address bit
 S: Sign of AC
 Z: Zero value in AC

BR: one of {JMP, CALL, RET, MAP}

AD: one of {Symbolic address, NEXT, empty}

SYMBOLIC MICROPROGRAM - FETCH ROUTINE

During FETCH, Read an instruction from memory and decode the instruction and update PC

Sequence of microoperations in the fetch cycle:

AR ← PC
DR ← M[AR], PC ← PC + 1
AR ← DR(0-10), CAR(2-5) ← DR(11-14), CAR(0,1,6) ← 0

Symbolic microprogram for the fetch cycle:

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

Binary equivalents translated by an assembler

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

SYMBOLIC MICROPROGRAM

- Control Storage: 128 20-bit words
- The first 64 words: Routines for the 16 machine instructions
- The last 64 words: Used for other purpose (e.g., fetch routine and other subroutines)
- Mapping: OP-code XXXX into 0XXXX00, the first address for the 16 routines are 0(0 0000 00), 4(0 0001 00), 8, 12, 16, 20, ..., 60

Partial Symbolic Microprogram

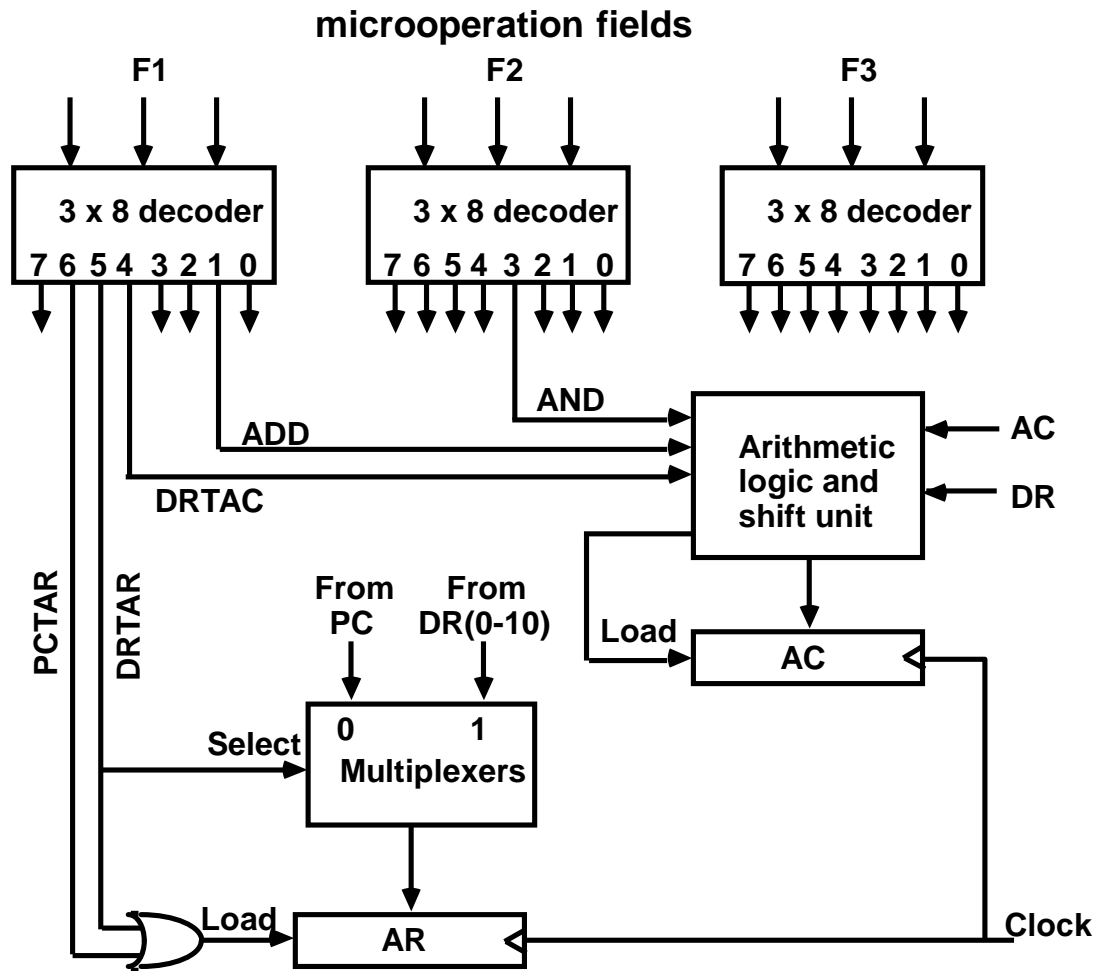
Label	Microops	CD	BR	AD
ADD:	ORG 0			
	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ADD	U	JMP	FETCH
BRANCH:	ORG 4			
	NOP	S	JMP	OVER
	NOP	U	JMP	FETCH
OVER:	NOP	I	CALL	INDRCT
	ARTPC	U	JMP	FETCH
STORE:	ORG 8			
	NOP	I	CALL	INDRCT
	ACTDR	U	JMP	NEXT
	WRITE	U	JMP	FETCH
EXCHANGE:	ORG 12			
	NOP	I	CALL	INDRCT
	READ	U	JMP	NEXT
	ACTDR, DRTAC	U	JMP	NEXT
	WRITE	U	JMP	FETCH
FETCH:	ORG 64			
	PCTAR	U	JMP	NEXT
	READ, INCPC	U	JMP	NEXT
	DRTAR	U	MAP	
INDRCT:	READ	U	JMP	NEXT
	DRTAR	U	RET	

BINARY MICROPROGRAM

Micro Routine	Address		Binary Microinstruction					
	Decimal	Binary	F1	F2	F3	CD	BR	AD
ADD	0	0000000	000	000	000	01	01	1000011
	1	0000001	000	100	000	00	00	0000010
	2	0000010	001	000	000	00	00	1000000
	3	0000011	000	000	000	00	00	1000000
BRANCH	4	0000100	000	000	000	10	00	0000110
	5	0000101	000	000	000	00	00	1000000
	6	0000110	000	000	000	01	01	1000011
	7	0000111	000	000	110	00	00	1000000
STORE	8	0001000	000	000	000	01	01	1000011
	9	0001001	000	101	000	00	00	0001010
	10	0001010	111	000	000	00	00	1000000
	11	0001011	000	000	000	00	00	1000000
EXCHANGE	12	0001100	000	000	000	01	01	1000011
	13	0001101	001	000	000	00	00	0001110
	14	0001110	100	101	000	00	00	0001111
	15	0001111	111	000	000	00	00	1000000
FETCH	64	1000000	110	000	000	00	00	1000001
	65	1000001	000	100	101	00	00	1000010
	66	1000010	101	000	000	00	11	0000000
	67	1000011	000	100	000	00	00	1000100
INDRCT	68	1000100	101	000	000	00	10	0000000

This microprogram can be implemented using ROM

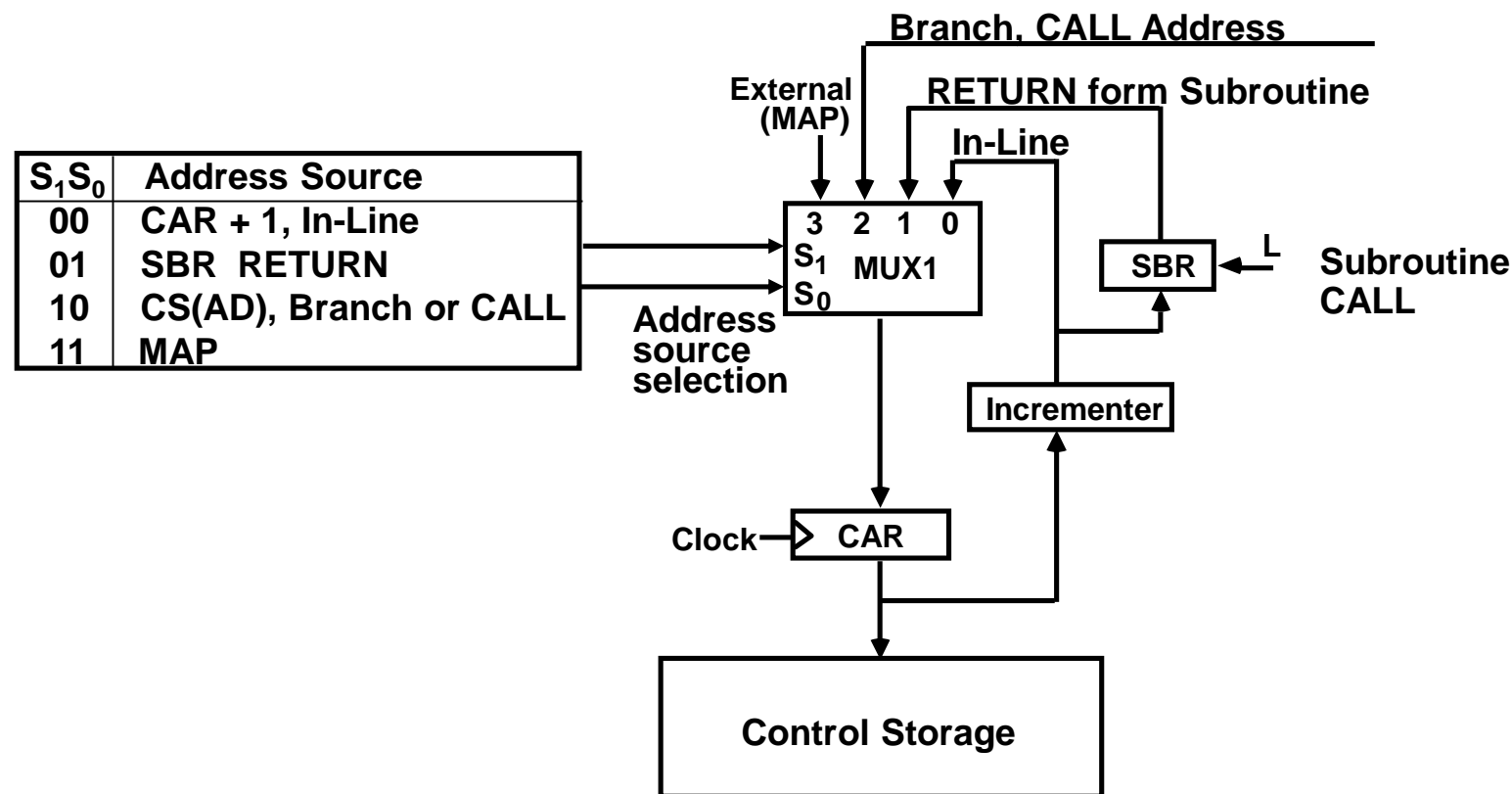
DESIGN OF CONTROL UNIT
- DECODING ALU CONTROL INFORMATION -



Decoding of Microoperation Fields

MICROPROGRAM SEQUENCER

- NEXT MICROINSTRUCTION ADDRESS LOGIC -

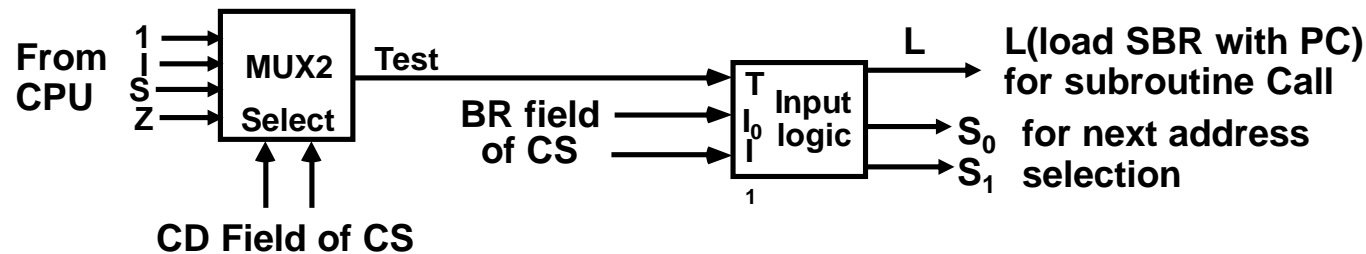


MUX-1 selects an address from one of four sources and routes it into a CAR

- In-Line Sequencing \rightarrow CAR + 1
- Branch, Subroutine Call \rightarrow CS(AD)
- Return from Subroutine \rightarrow Output of SBR
- New Machine instruction \rightarrow MAP

MICROPROGRAM SEQUENCER

- CONDITION AND BRANCH CONTROL -

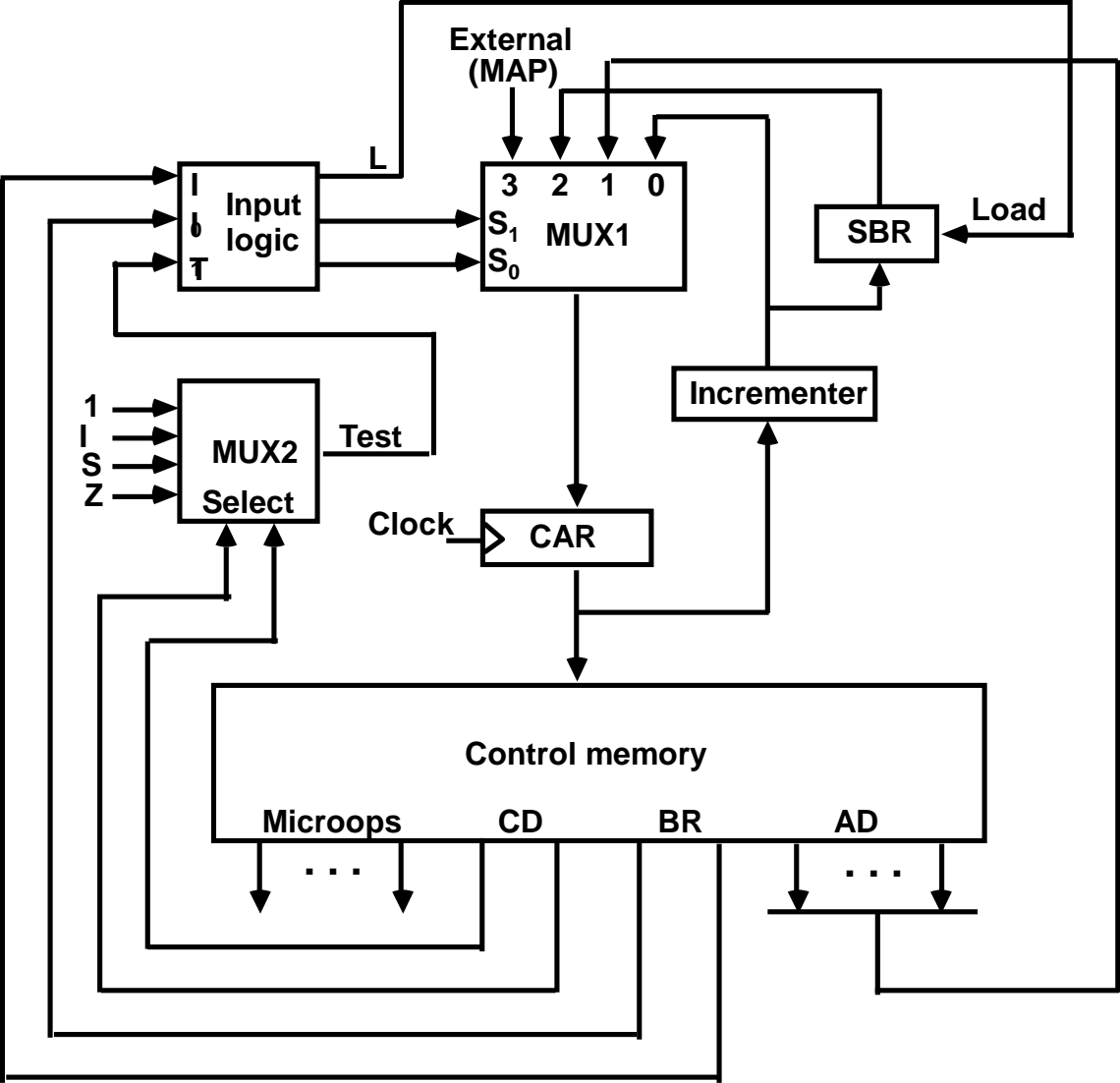


Input Logic

$I_1 I_0 T$	Meaning	Source of Address	$S_1 S_0$	L
000	In-Line	CAR+1	00	0
001	JMP	CS(AD)	01	0
010	In-Line	CAR+1	00	0
011	CALL	CS(AD) and SBR <- CAR+1	01	1
10x	RET	SBR	10	0
11x	MAP	DR(11-14)	11	0

$$\begin{aligned} S_1 &= I_1 \\ S_0 &= I_1 I_0 + I_1' T \\ L &= I_1' I_0 T \end{aligned}$$

MICROPROGRAM SEQUENCER



MICROINSTRUCTION FORMAT

Information in a Microinstruction

- Control Information
- Sequencing Information
- Constant

Information which is useful when feeding into the system

These information needs to be organized in some way for

- Efficient use of the microinstruction bits
- Fast decoding

Field Encoding

- Encoding the microinstruction bits
- Encoding slows down the execution speed due to the decoding delay
- Encoding also reduces the flexibility due to the decoding hardware

HORIZONTAL AND VERTICAL MICROINSTRUCTION FORMAT

Horizontal Microinstructions

Each bit directly controls each micro-operation or each control point

Horizontal implies a long microinstruction word

Advantages: Can control a variety of components operating in parallel.

--> Advantage of efficient hardware utilization

Disadvantages: Control word bits are not fully utilized

--> CS becomes large --> Costly

Vertical Microinstructions

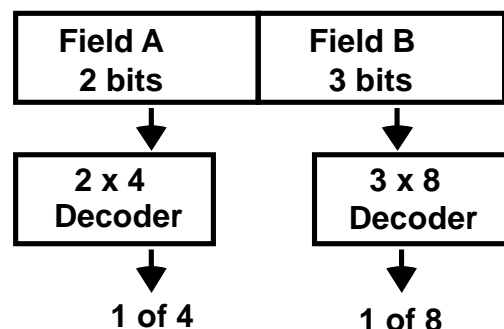
A microinstruction format that is not horizontal

Vertical implies a short microinstruction word

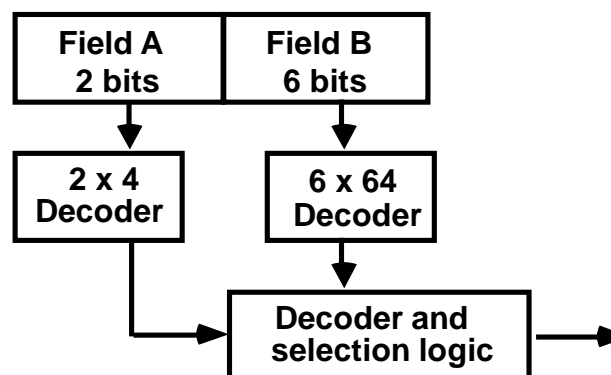
Encoded Microinstruction fields

--> Needs decoding circuits for one or two levels of decoding

One-level decoding



Two-level decoding



NANOSTORAGE AND NANOINSTRUCTION

The decoder circuits in a vertical microprogram storage organization can be replaced by a ROM

=> Two levels of control storage

First level - *Control Storage*

Second level - *Nano Storage*

Two-level microprogram

First level

- *Vertical* format Microprogram

Second level

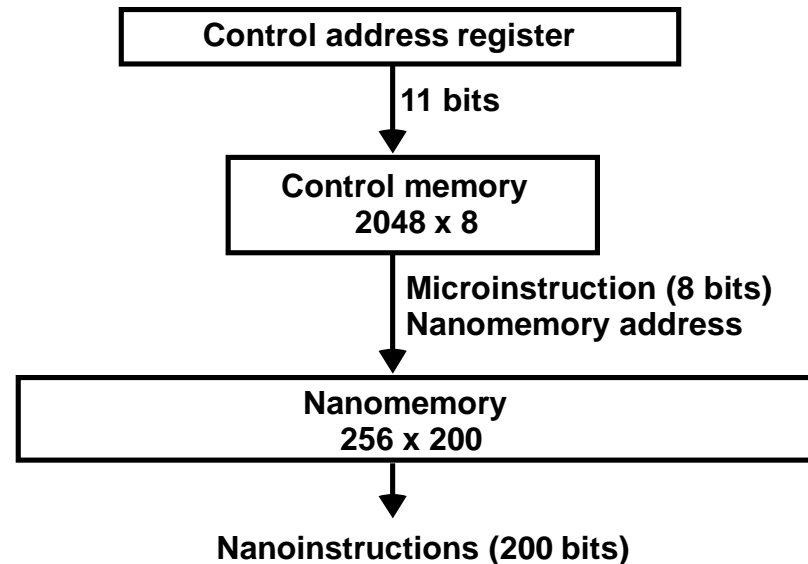
- *Horizontal* format Nanoprogram

- Interprets the microinstruction fields, thus converts a vertical microinstruction format into a horizontal nanoinstruction format.

Usually, the microprogram consists of a large number of short microinstructions, while the nanoprogram contains fewer words with longer nanoinstructions.

TWO-LEVEL MICROPROGRAMMING - EXAMPLE

- * Microprogram: 2048 microinstructions of 200 bits each
- * With 1-Level Control Storage: $2048 \times 200 = 409,600$ bits
- * Assumption:
 - 256 distinct microinstructions among 2048
- * With 2-Level Control Storage:
 - Nano Storage: 256×200 bits to store 256 distinct nanoinstructions
 - Control storage: 2048×8 bits
 - To address 256 nano storage locations 8 bits are needed
- * Total 1-Level control storage: 409,600 bits
- Total 2-Level control storage: 67,584 bits ($256 \times 200 + 2048 \times 8$)



Overview

- **Instruction Set Processor (ISP)**
- **Central Processing Unit (CPU)**
- **A typical computing task consists of a series of steps specified by a sequence of machine instructions that constitute a program.**
- **An instruction is executed by carrying out a sequence of more rudimentary operations.**

Fundamental Concepts

- **Processor fetches one instruction at a time and perform the operation specified.**
- **Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered.**
- **Processor keeps track of the address of the memory location containing the next instruction to be fetched using Program Counter (PC).**
- **Instruction Register (IR)**

Executing an Instruction

- Fetch the contents of the memory location pointed to by the PC. The contents of this location are loaded into the IR (fetch phase).

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

- Assuming that the memory is byte addressable, increment the contents of the PC by 4 (fetch phase).

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

- Carry out the actions specified by the instruction in the IR (execution phase).

Processor Organization

MDR HAS
TWO INPUTS
AND TWO
OUTPUTS

Datapath

Textbook Page 413

Executing an Instruction

- **Transfer a word of data from one processor register to another or to the ALU.**
- **Perform an arithmetic or a logic operation and store the result in a processor register.**
- **Fetch the contents of a given memory location and load them into a processor register.**
- **Store a word of data from a processor register into a given memory location.**

Register Transfers

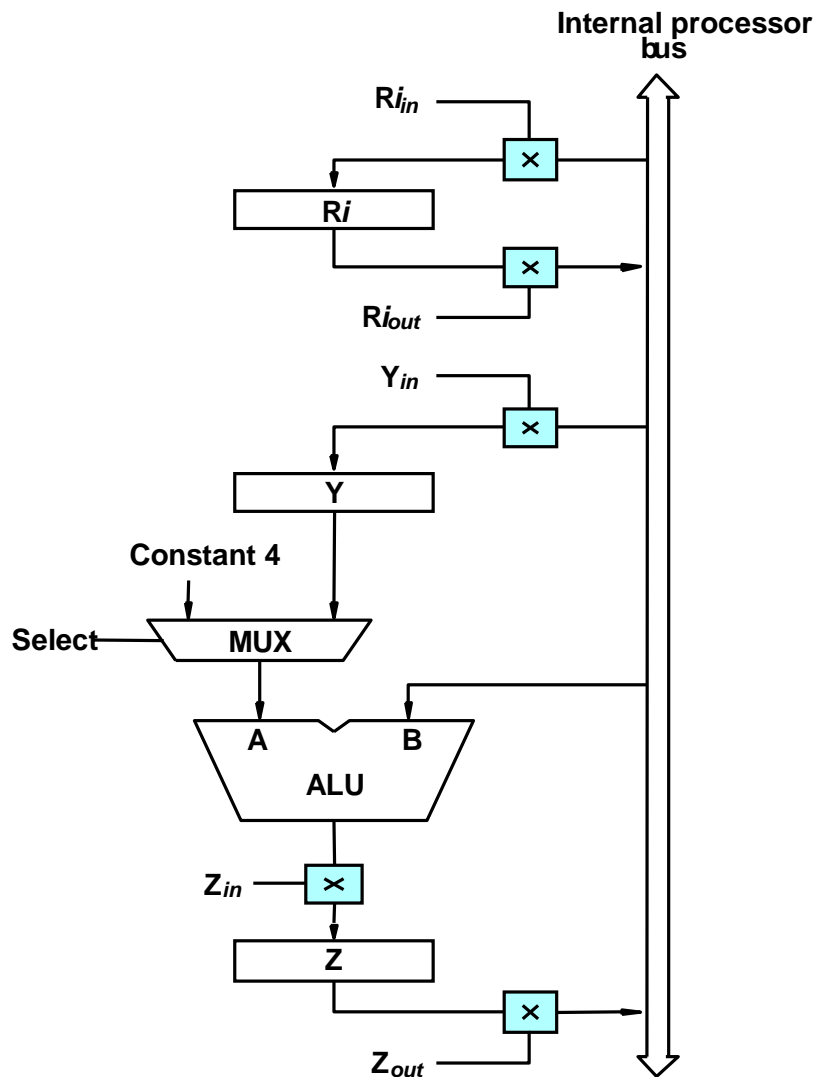


Figure 7.2. Input and output gating for the registers in Figure 7.1.

Register Transfers

- **All operations and data transfers are controlled by the processor clock.**

Figure 7.3. Input and output gating for one register bit.

Performing an Arithmetic or Logic Operation

- The ALU is a combinational circuit that has no internal storage.
- ALU gets the two operands from MUX and bus. The result is temporarily stored in register Z.
- What is the sequence of operations to add the contents of register R1 to those of R2 and store the result in R3?
 1. R1out, Yin
 2. R2out, SelectY, Add, Zin
 3. Zout, R3in

Fetching a Word from Memory

- Address into MAR; issue Read operation; data into MDR.

Figure 7.4. Connection and control signals for register MDR.

Fetching a Word from Memory

- The response time of each memory access varies (cache miss, memory-mapped I/O,...).
- To accommodate this, the processor waits until it receives an indication that the requested operation has been completed (Memory-Function-Completed, MFC).
- **Move (R1), R2**
 - $MAR \leftarrow [R1]$
 - Start a Read operation on the memory bus
 - Wait for the MFC response from the memory
 - Load MDR from the memory bus
 - $R2 \leftarrow [MDR]$

Timing

Assume MAR
is always available
on the address lines
of the memory bus.

$\text{MAR} \leftarrow [\text{R1}]$

Start a Read operation on the memory bus

Wait for the MFC response from the memory

Load MDR from the memory bus

$\text{R2} \leftarrow [\text{MDR}]$

Execution of a Complete Instruction

- **Add (R3), R1**
- **Fetch the instruction**
- **Fetch the first operand (the contents of the memory location pointed to by R3)**
- **Perform the addition**
- **Load the result into R1**

Architecture

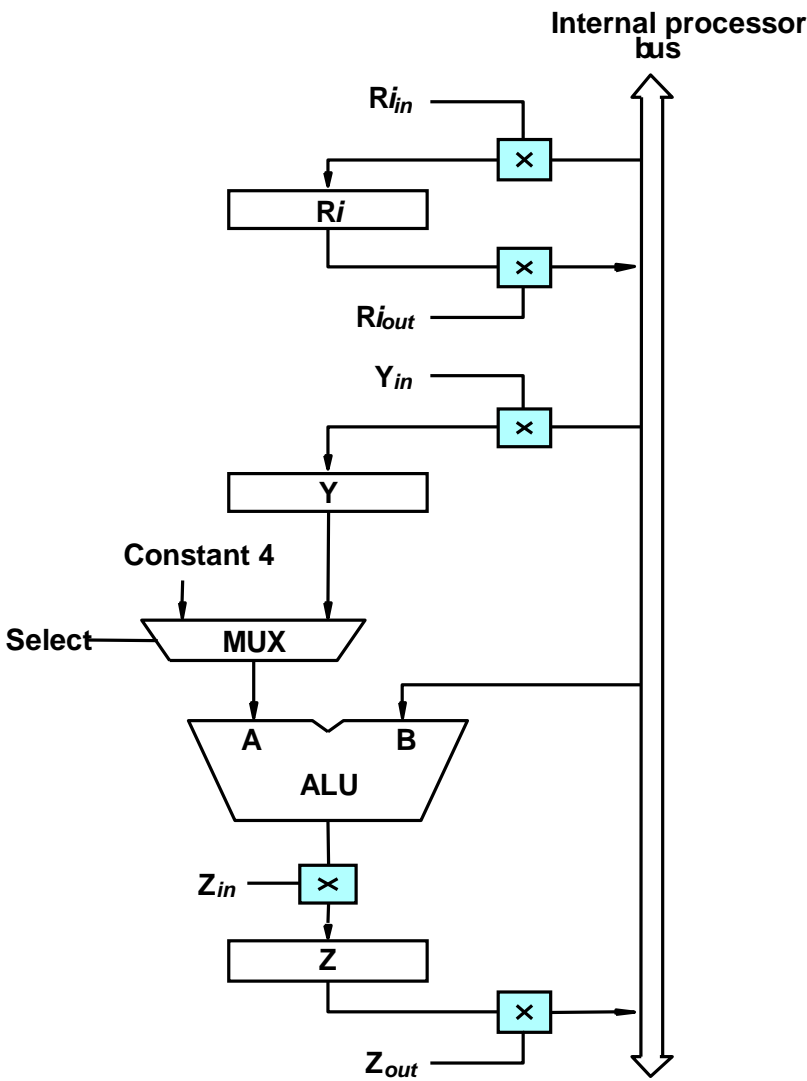


Figure 7.2. Input and output gating for the registers in Figure 7.1.

Execution of a Complete Instruction

Add (R3), R1

Execution of Branch Instructions

- A branch instruction replaces the contents of PC with the branch target address, which is usually obtained by adding an offset X given in the branch instruction.
- The offset X is usually the difference between the branch target address and the address immediately following the branch instruction.
- Conditional branch

Execution of Branch Instructions

Step Action

- 1 PC_{out} , MAR_{in} , Read, Select4, Add, Z_{in}
 - 2 Z_{out} , PC_{in} , Y_{in} , WMF C
 - 3 MDR_{out} , IR_{in}
 - 4 Offset-field-of- IR_{out} , Add, Z_{in}
 - 5 Z_{out} , PC_{in} , End
-

Figure 7.7. Control sequence for an unconditional branch instruction.

Multiple-Bus Organization

Multiple-Bus Organization

- Add R4, R5, R6

Step	Action
1	PC out, R=B, MAR in, Read, IncPC
2	WMF C
3	MDR outB, R=B, IR in
4	R4 outA, R5 outB, SelectA, Add, R6 in, End

Figure 7.9. Control sequence for the instruction. Add R4,R5,R6, for the three-bus organization in Figure 7.8.

Quiz

- **What is the control sequence for execution of the instruction
Add R1, R2
including the instruction fetch phase? (Assume single bus architecture)**

Control Unit Organization

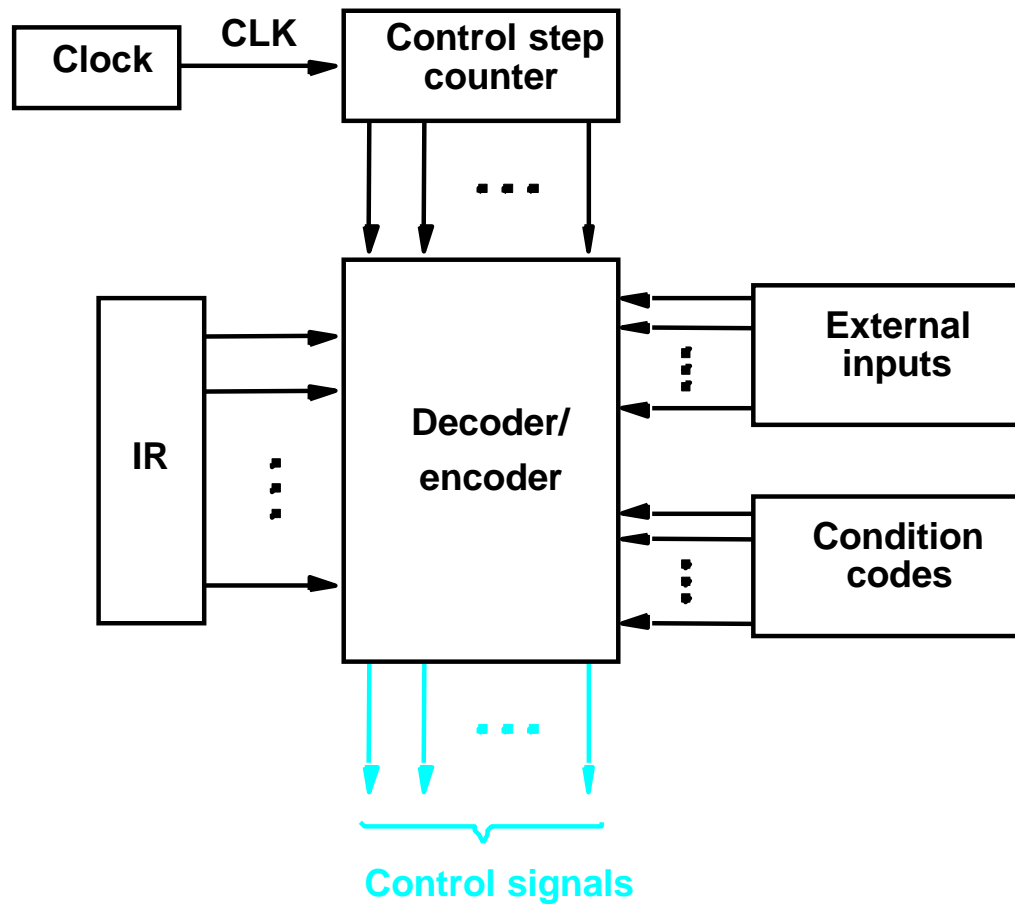


Figure 7.10. Control unit organization.

Detailed Block Description



Generating Z_{in}

- $Z_{in} = T_1 + T_6 \cdot \text{ADD} + T_4 \cdot \text{BR} + \dots$

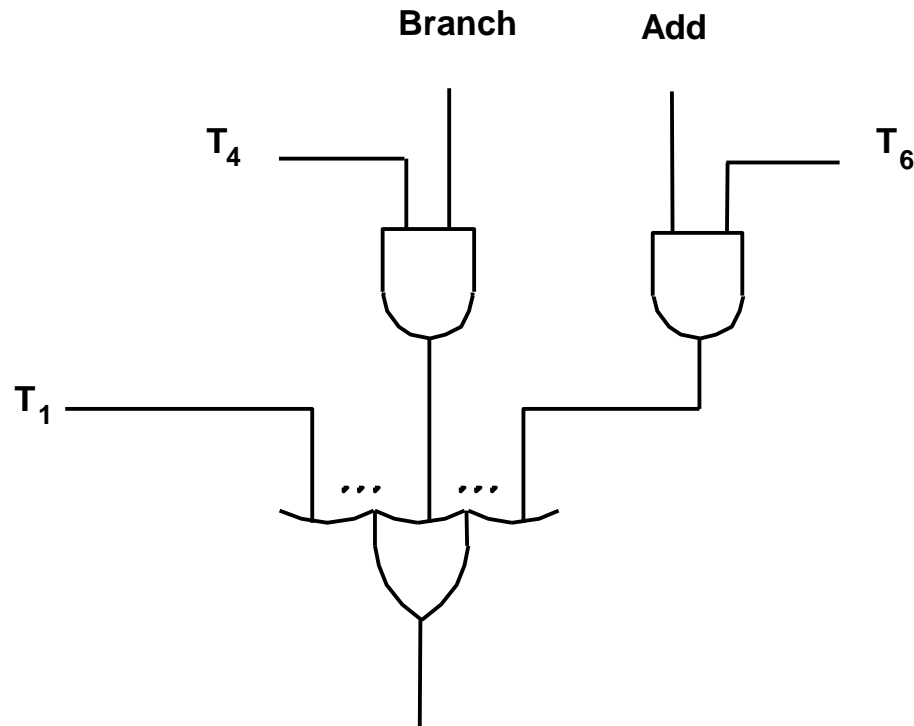


Figure 7.12. Generation of the Z_{in} control signal for the processor in Figure 7.1.

Generating End

- $\text{End} = T_7 \cdot \text{ADD} + T_5 \cdot \text{BR} + (T_5 \cdot N + T_4 \cdot \overline{N}) \cdot \text{BRN} + \dots$

A Complete Processor

Overview

- **Control signals are generated by a program similar to machine language programs.**
- **Control Word (CW); microroutine; microinstruction**

Overview

Overview

- **Control store**

One function
cannot be carried
out by this simple
organization.

Overview

- The previous organization cannot handle the situation when the control unit is required to check the status of the condition codes or external inputs to choose between alternative courses of action.
- Use conditional branch microinstruction.

AddressMicroinstruction	
0	PC _{out} , MAR _{in} , Read,Select4,Add, Z _{in}
1	Z _{out} , PC _{in} , Y _{in} , WMFC
2	MDR _{out} , IR _{in}
3	Branch to starting address of appropriate microroutine
.....	
25	If N=0, then branch to microinstruction n
26	Offset-field-of-IR _{out} , SelectY,Add, Z _{in}
27	Z _{out} , PC _{in} , End

Figure 7.17. Microroutine for the instruction Branch<0.

Overview

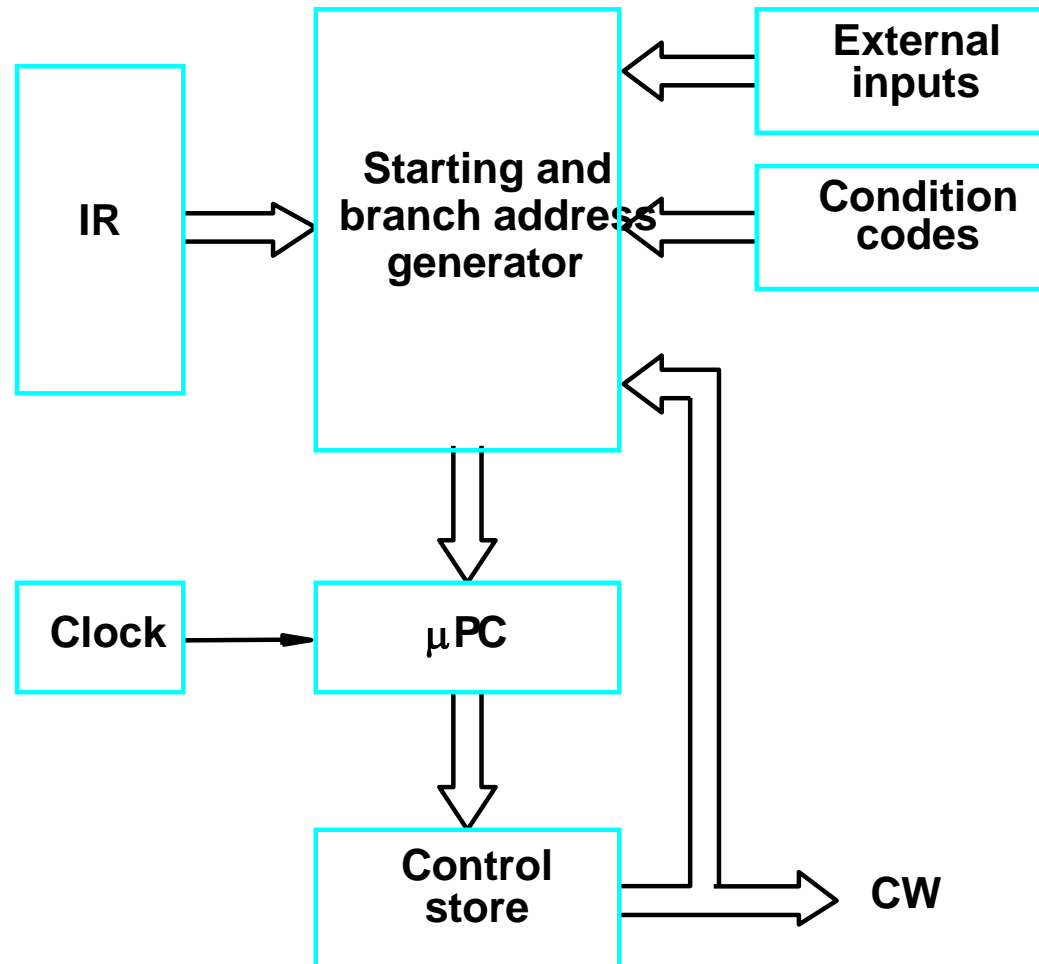


Figure 7.18.

Organization of the control unit to allow conditional branching in the microprogram.

Microinstructions

- A straightforward way to structure microinstructions is to assign one bit position to each control signal.
- However, this is very inefficient.
- The length can be reduced: most signals are not needed simultaneously, and many signals are mutually exclusive.
- All mutually exclusive signals are placed in the same group in binary coding.

Partial Format for the Microinstructions

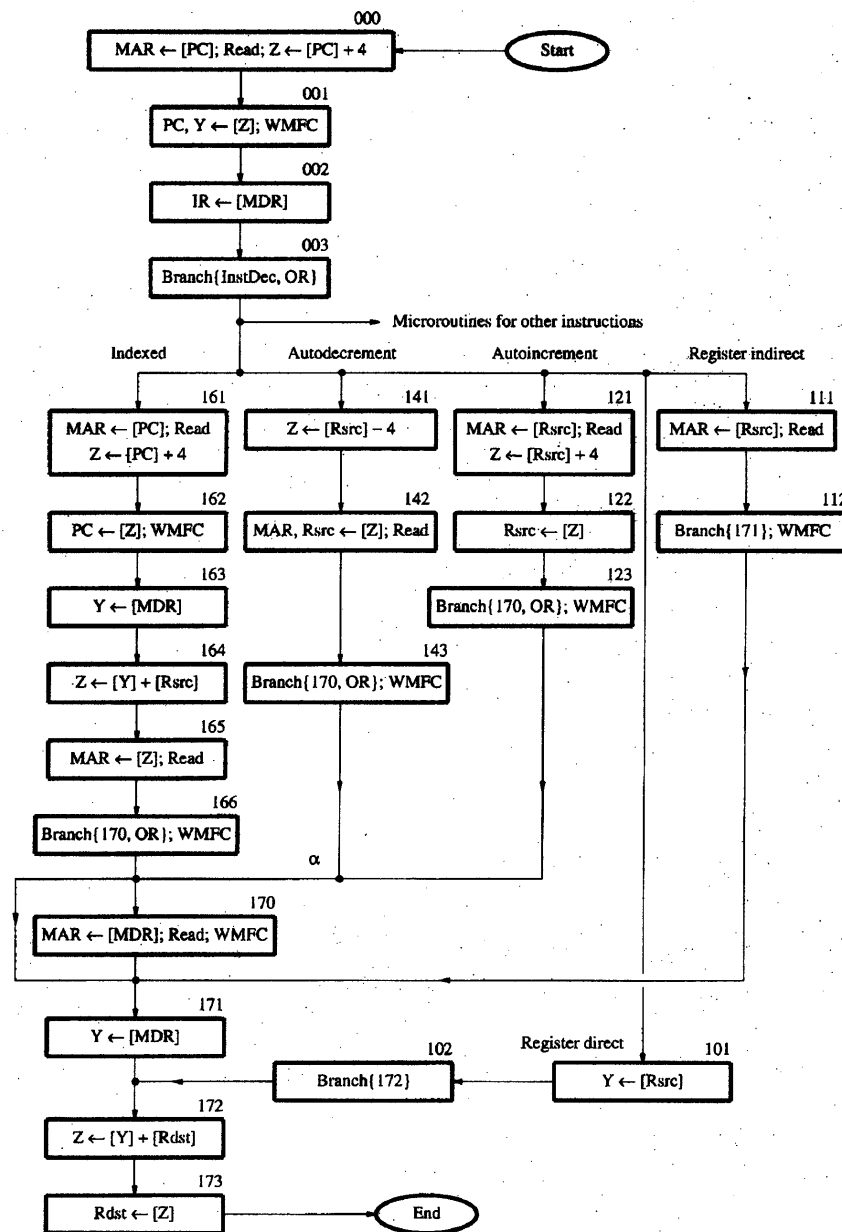
What is the price paid for
this scheme?

Further Improvement

- Enumerate the patterns of required signals in all possible microinstructions. Each meaningful combination of active control signals can then be assigned a distinct code.
- Vertical organization
- Horizontal organization

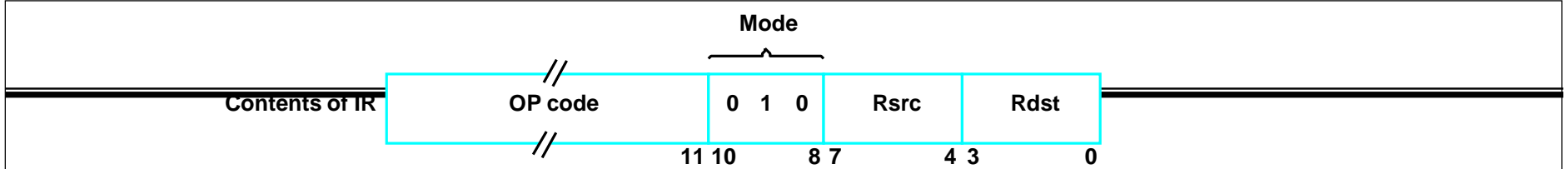
Microprogram Sequencing

- If all microprograms require only straightforward sequential execution of microinstructions except for branches, letting a μ PC governs the sequencing would be efficient.
- However, two disadvantages:
 - Having a separate microroutine for each machine instruction results in a large total number of microinstructions and a large control store.
 - Longer execution time because it takes more time to carry out the required branches.
- Example: Add src, Rdst
- Four addressing modes: register, autoincrement, autodecrement, and indexed (with indirect forms).



- Bit-ORing
- Wide-Branch Addressing
- WMFC

Figure 7.20. Flowchart of a microprogram for the Add src,Rdst instruction.



Address (octal)	Microinstruction
000	$PC_{out} \leftarrow MAR_{in}$, Read, Select4, Add, Z_{in}
001	$Z_{out} \leftarrow PC_{in}$, Y_{in} , WMFC
002	$MDR_{out} \leftarrow IR_{in}$
003	μ Branch ($PC \leftarrow 101$ (from Instruction decoder); $\mu PC_{5,4} \leftarrow [IR_{10,9}]$; $\mu PC_3 \leftarrow [\overline{IR_{10}}] \times [\overline{IR_9}] \times [IR_8]$)
121	$Rsrc_{out} \leftarrow MAR_{in}$, Read, Select4, Add, Z_{in}
122	$Z_{out} \leftarrow Rsrc_{in}$
123	μ Branch ($PC \leftarrow 170$; $\mu PC_0 \leftarrow [\overline{IR_8}]$), WMFC
170	$MDR_{out} \leftarrow MAR_{in}$, Read, WMFC
171	$MDR_{out} \leftarrow Y_{in}$
172	$Rdst_{out} \leftarrow Select4$, Add, Z_{in}
173	$Z_{out} \leftarrow Rdst_{in}$, End

Figure 7.21. Microinstruction for Add (Rsrc)+,Rdst.
Note Microinstruction at location 170 is not executed for this addressing mode.

Microinstructions with Next-Address Field

- The microprogram we discussed requires several branch microinstructions, which perform no useful operation in the datapath.
- A powerful alternative approach is to include an address field as a part of every microinstruction to indicate the location of the next microinstruction to be fetched.
- Pros: separate branch microinstructions are virtually eliminated; few limitations in assigning addresses to microinstructions.
- Cons: additional bits for the address field (around 1/6)

Microinstructions with Next- Address Field

Implementation of the Microroutine

bit-ORing

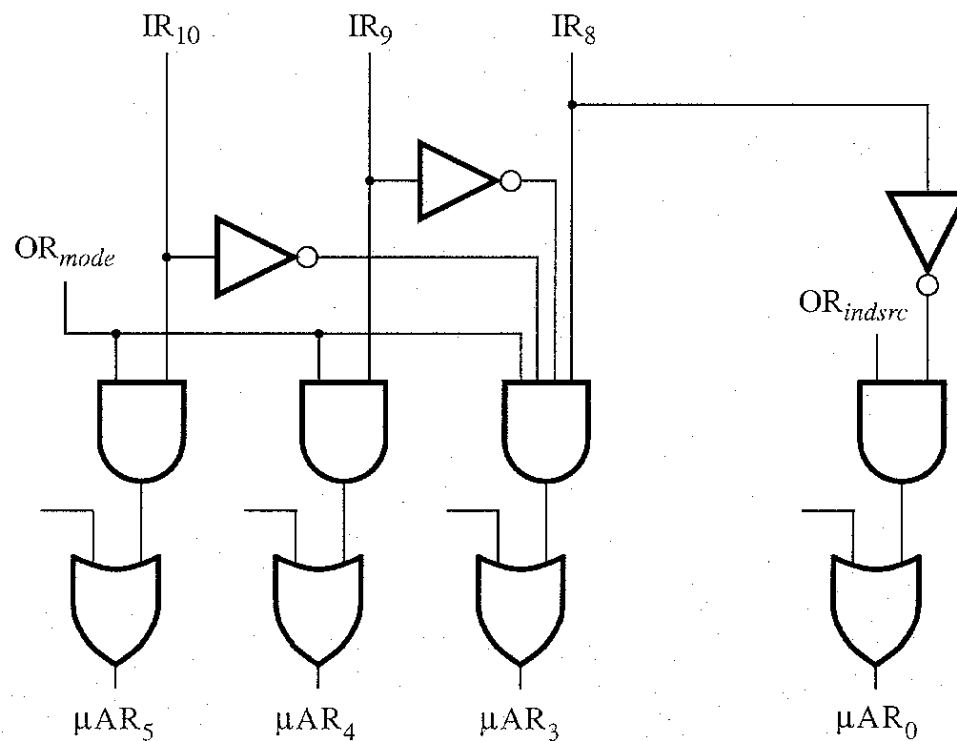


Figure 7.26. Control circuitry for bit-ORing
(part of the decoding circuits in Figure 7.25).

PIPELINING AND VECTOR PROCESSING

- **Parallel Processing**
- **Pipelining**
- **Arithmetic Pipeline**
- **Instruction Pipeline**
- **RISC Pipeline**
- **Vector Processing**
- **Array Processors**

PARALLEL PROCESSING

Execution of *Concurrent Events* in the computing process to achieve faster *Computational Speed*

Levels of Parallel Processing

- Job or Program level
- Task or Procedure level
- Inter-Instruction level
- Intra-Instruction level

PARALLEL COMPUTERS

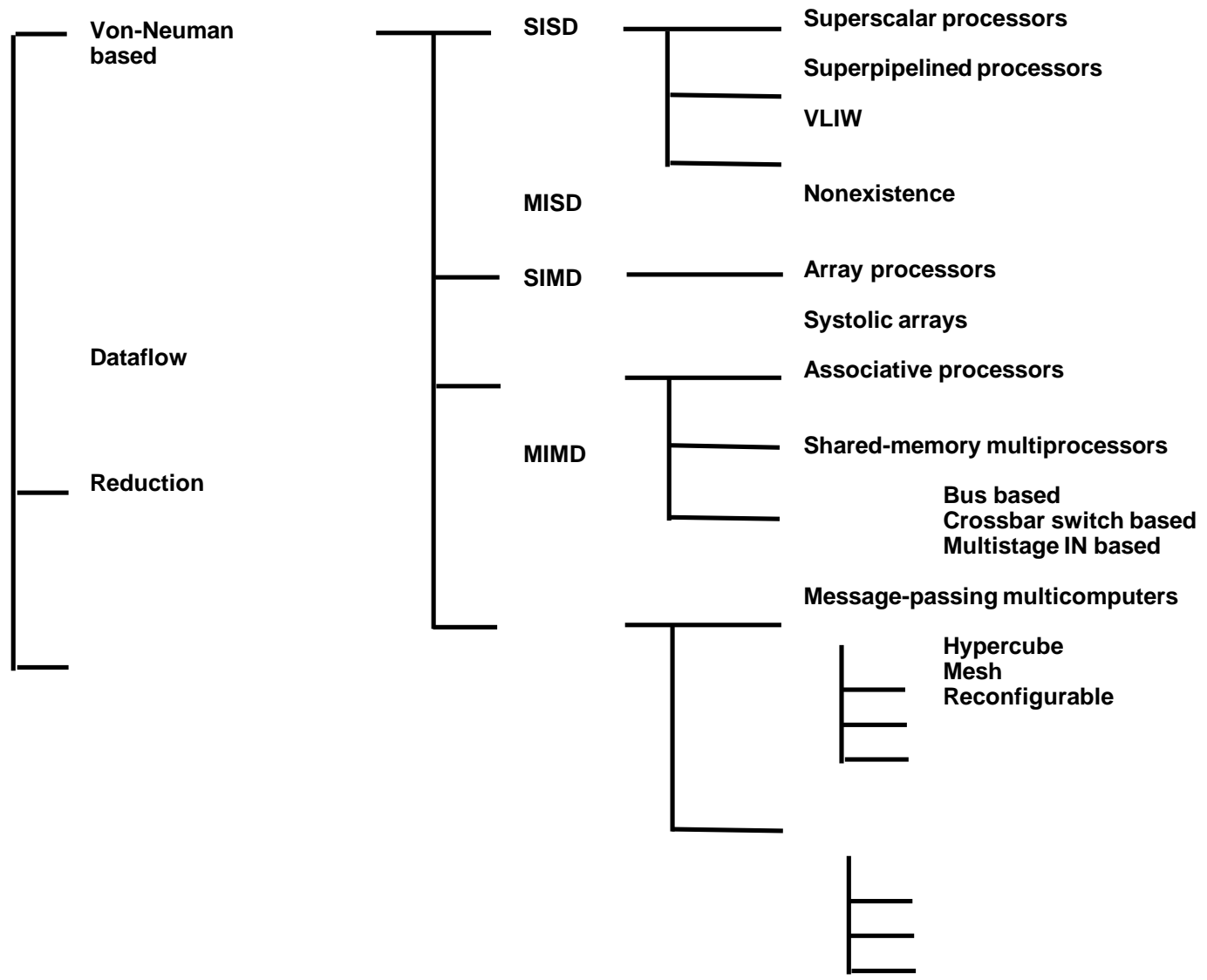
Architectural Classification

– Flynn's classification

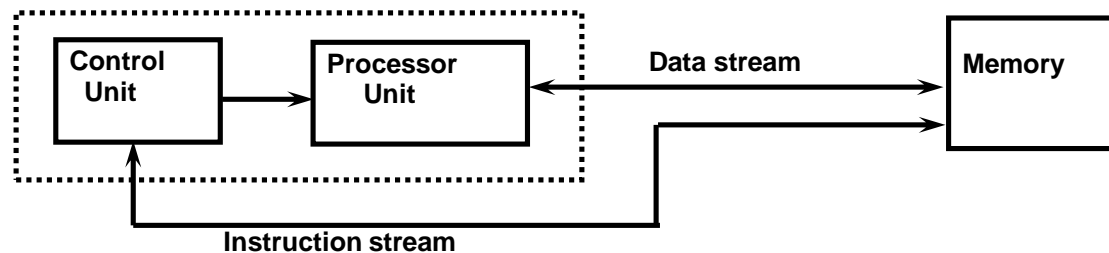
- » Based on the multiplicity of *Instruction Streams* and *Data Streams*
- » Instruction Stream
 - Sequence of Instructions read from memory
- » Data Stream
 - Operations performed on the data in the processor

		Number of <i>Data Streams</i>	
		Single	Multiple
Number of <i>Instruction Streams</i>	Single	SISD	SIMD
	Multiple	MISD	MIMD

COMPUTER ARCHITECTURES FOR PARALLEL PROCESSING



SISD COMPUTER SYSTEMS



Characteristics

- Standard von Neumann machine
- Instructions and data are stored in memory
- One operation at a time

Limitations

Von Neumann bottleneck

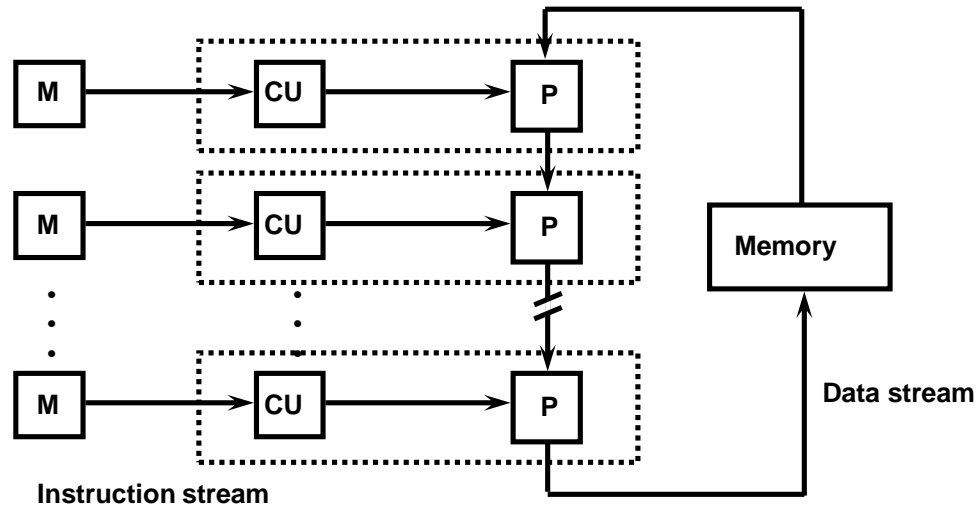
Maximum speed of the system is limited by the *Memory Bandwidth* (bits/sec or bytes/sec)

- Limitation on *Memory Bandwidth*
- Memory is shared by CPU and I/O

SISD PERFORMANCE IMPROVEMENTS

- **Multiprogramming**
- **Spooling**
- **Multifunction processor**
- **Pipelining**
- **Exploiting instruction-level parallelism**
 - **Superscalar**
 - **Superpipelining**
 - **VLIW (Very Long Instruction Word)**

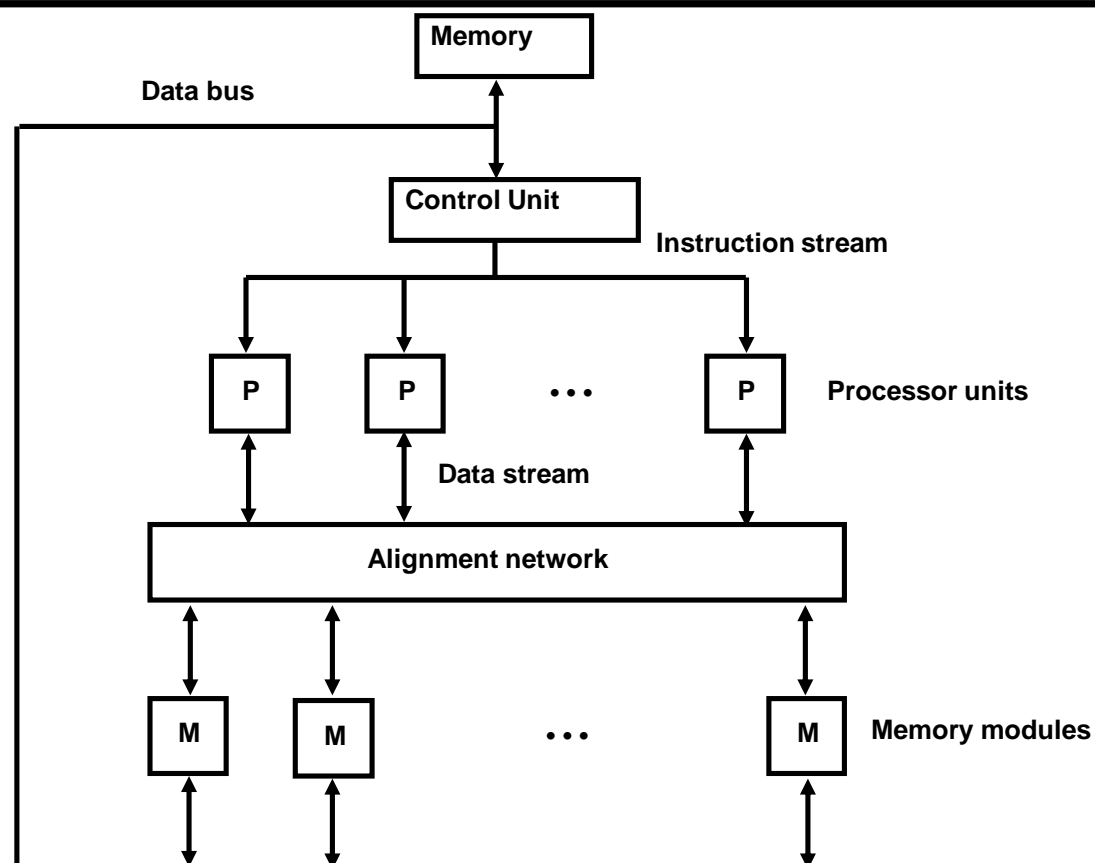
MISD COMPUTER SYSTEMS



Characteristics

- There is no computer at present that can be classified as MISD

SIMD COMPUTER SYSTEMS



Characteristics

- Only one copy of the program exists
- A single controller executes one instruction at a time

TYPES OF SIMD COMPUTERS

Array Processors

- The control unit broadcasts instructions to all PEs, and all active PEs execute the same instructions
- ILLIAC IV, GF-11, Connection Machine, DAP, MPP

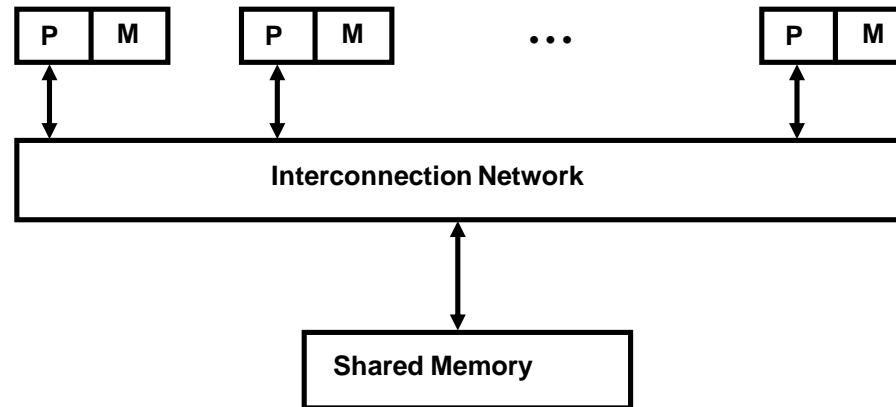
Systolic Arrays

- Regular arrangement of a large number of very simple processors constructed on VLSI circuits
- CMU Warp, Purdue CHiP

Associative Processors

- Content addressing
- Data transformation operations over many sets of arguments with a single instruction
- STARAN, PEPE

MIMD COMPUTER SYSTEMS



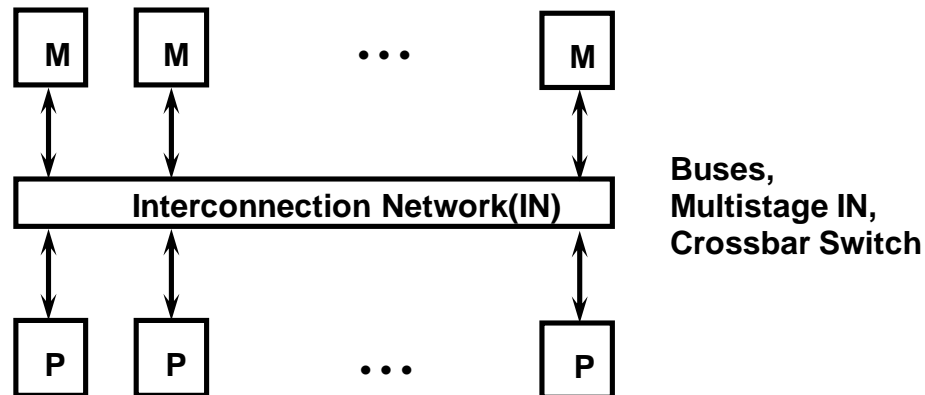
Characteristics

- Multiple processing units
- Execution of multiple instructions on multiple data

Types of MIMD computer systems

- Shared memory multiprocessors
- Message-passing multicomputers

SHARED MEMORY MULTIPROCESSORS



Characteristics

All processors have equally direct access to one large memory address space

Example systems

Bus and cache-based systems

- Sequent Balance, Encore Multimax

Multistage IN-based systems

- Ultracomputer, Butterfly, RP3, HEP

Crossbar switch-based systems

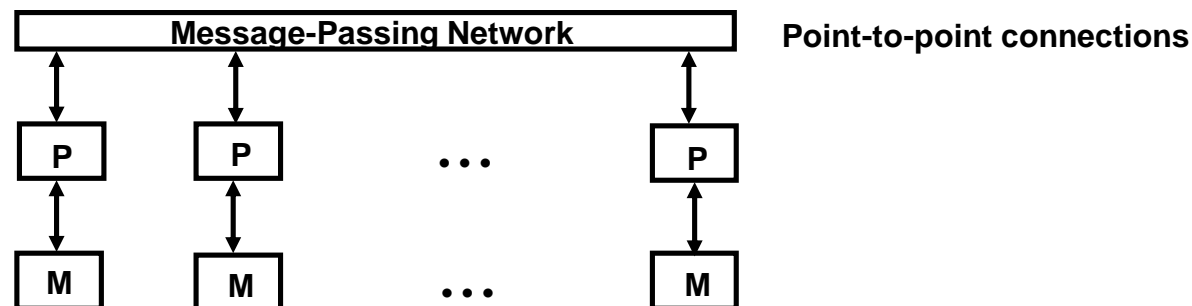
- C.mmp, Alliant FX/8

Limitations

Memory access latency

Hot spot problem

MESSAGE-PASSING MULTICOMPUTER



Characteristics

- Interconnected computers
- Each processor has its own memory, and communicate via message-passing

Example systems

- Tree structure: Teradata, DADO
- Mesh-connected: Rediflow, Series 2010, J-Machine
- Hypercube: Cosmic Cube, iPSC, NCUBE, FPS T Series, Mark III

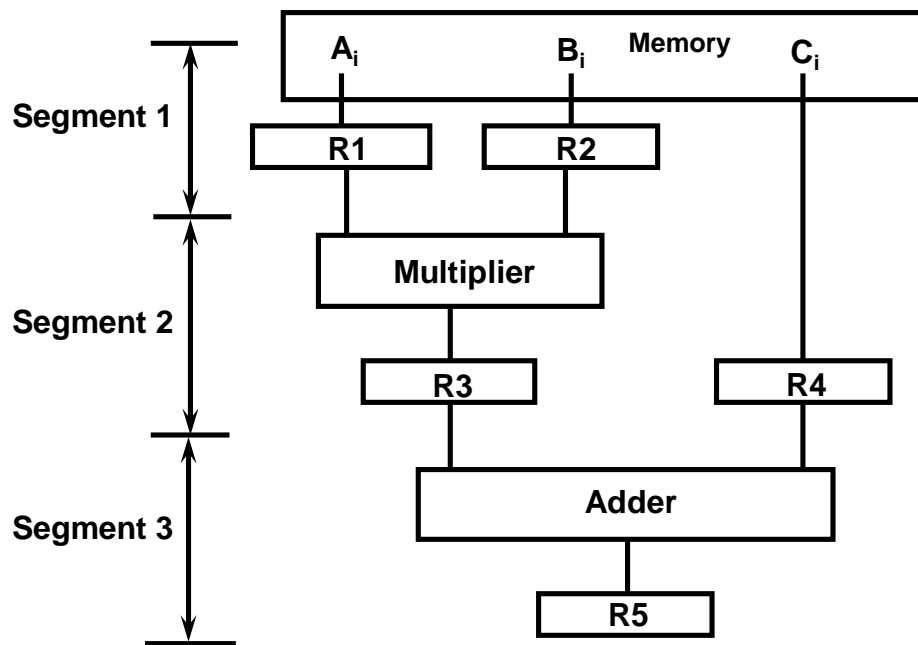
Limitations

- Communication overhead
- Hard to programming

PIPELINING

A technique of decomposing a sequential process into suboperations, with each subprocess being executed in a partial dedicated segment that operates concurrently with all other segments.

$$A_i * B_i + C_i \quad \text{for } i = 1, 2, 3, \dots, 7$$



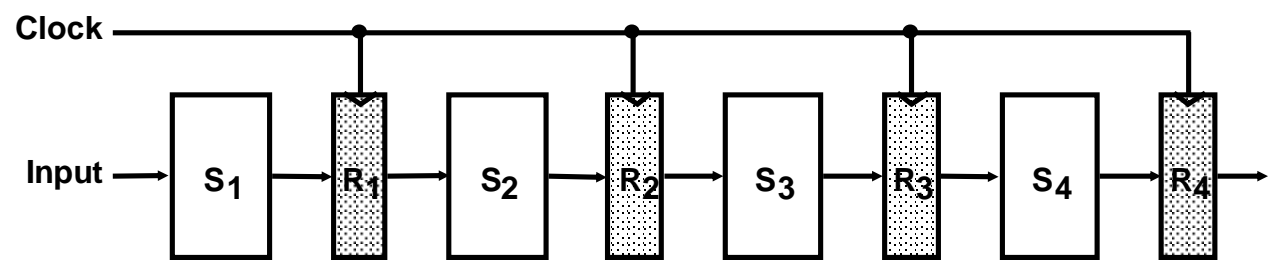
$R1 \leftarrow A_i, R2 \leftarrow B_i$	Load A_i and B_i
$R3 \leftarrow R1 * R2, R4 \leftarrow C_i$	Multiply and load C_i
$R5 \leftarrow R3 + R4$	Add

OPERATIONS IN EACH PIPELINE STAGE

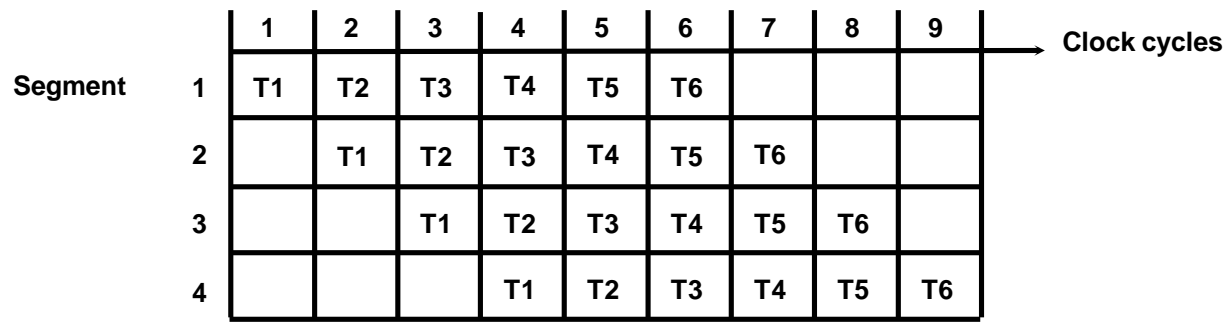
Clock Pulse	Segment 1			Segment 2		Segment 3
Number	R1	R2	R3	R4	R5	
1	A1	B1				
2	A2	B2	A1 * B1	C1		
3	A3	B3	A2 * B2	C2	A1 * B1 + C1	
4	A4	B4	A3 * B3	C3	A2 * B2 + C2	
5	A5	B5	A4 * B4	C4	A3 * B3 + C3	
6	A6	B6	A5 * B5	C5	A4 * B4 + C4	
7	A7	B7	A6 * B6	C6	A5 * B5 + C5	
8				A7 * B7	C7	A6 * B6 + C6
9						A7 * B7 + C7

GENERAL PIPELINE

General Structure of a 4-Segment Pipeline



Space-Time Diagram



PIPELINE SPEEDUP

n: Number of tasks to be performed

Conventional Machine (Non-Pipelined)

t_n: Clock cycle

τ₁: Time required to complete the n tasks

$$\tau_1 = n * t_n$$

Pipelined Machine (k stages)

t_p: Clock cycle (time to complete each suboperation)

τ_k: Time required to complete the n tasks

$$\tau_k = (k + n - 1) * t_p$$

Speedup

S_k: Speedup

$$S_k = n * t_n / (k + n - 1) * t_p$$

$$\lim_{n \rightarrow \infty} S_k = \frac{t_n}{t_p} \quad (= k, \text{ if } t_n = k * t_p)$$

PIPELINE AND MULTIPLE FUNCTION UNITS

Example

- 4-stage pipeline
- suboperation in each stage; $t_p = 20\text{nS}$
- 100 tasks to be executed
- 1 task in non-pipelined system; $20 \times 4 = 80\text{nS}$

Pipelined System

$$(k + n - 1) \times t_p = (4 + 99) \times 20 = 2060\text{nS}$$

Non-Pipelined System

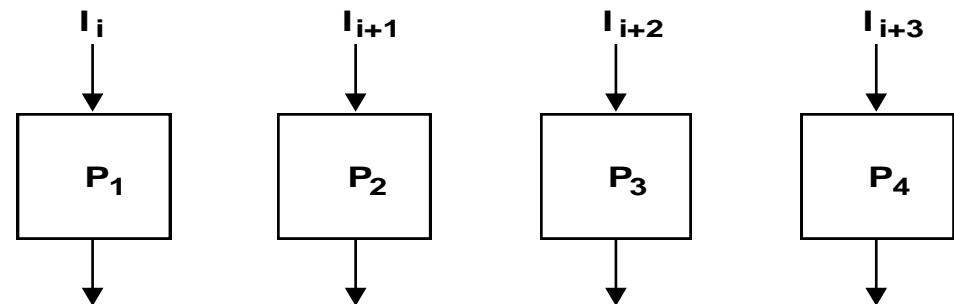
$$n \times k \times t_p = 100 \times 80 = 8000\text{nS}$$

Speedup

$$S_k = 8000 / 2060 = 3.88$$

4-Stage Pipeline is basically identical to the system with 4 identical function units

Multiple Functional Units



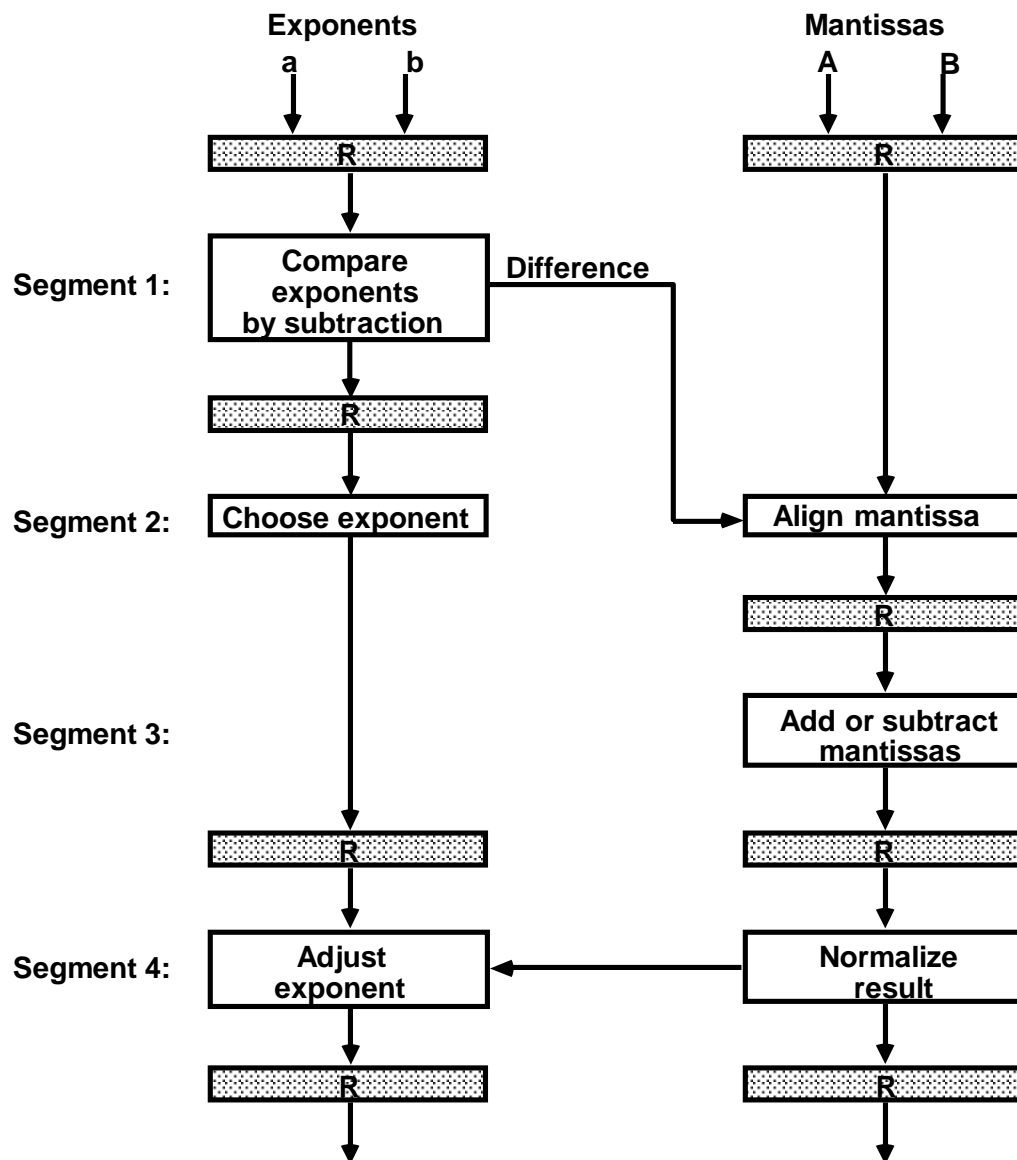
ARITHMETIC PIPELINE

Floating-point adder

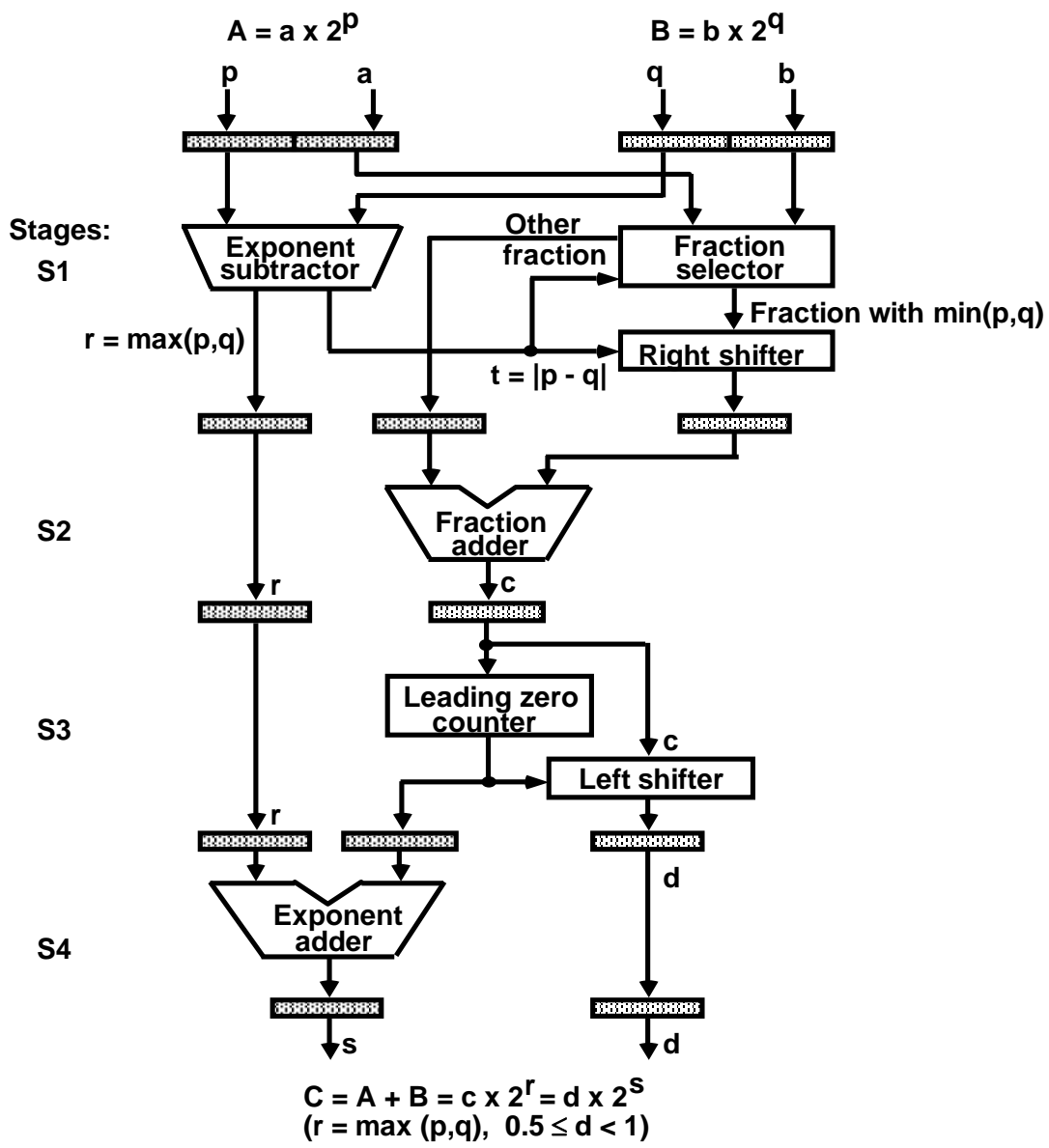
$$X = A \times 2^a$$

$$Y = B \times 2^b$$

- [1] Compare the exponents
- [2] Align the mantissa
- [3] Add/sub the mantissa
- [4] Normalize the result



4-STAGE FLOATING POINT ADDER



INSTRUCTION CYCLE

Six Phases* in an Instruction Cycle

- [1] Fetch an instruction from memory**
- [2] Decode the instruction**
- [3] Calculate the effective address of the operand**
- [4] Fetch the operands from memory**
- [5] Execute the operation**
- [6] Store the result in the proper place**

- * Some instructions skip some phases**
- * Effective address calculation can be done in the part of the decoding phase**
- * Storage of the operation result into a register is done automatically in the execution phase**

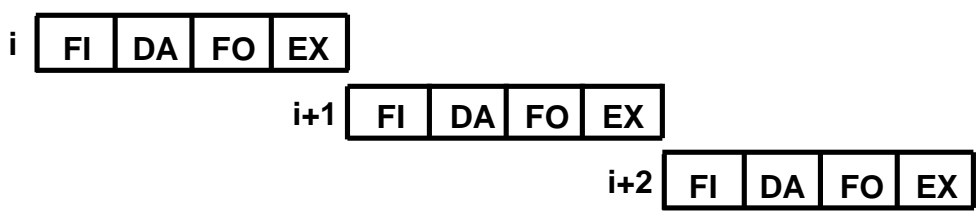
==> 4-Stage Pipeline

- [1] FI: Fetch an instruction from memory**
- [2] DA: Decode the instruction and calculate the effective address of the operand**
- [3] FO: Fetch the operand**
- [4] EX: Execute the operation**

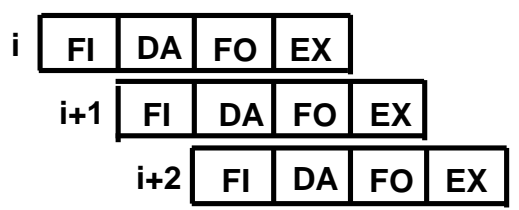
INSTRUCTION PIPELINE

Execution of Three Instructions in a 4-Stage Pipeline

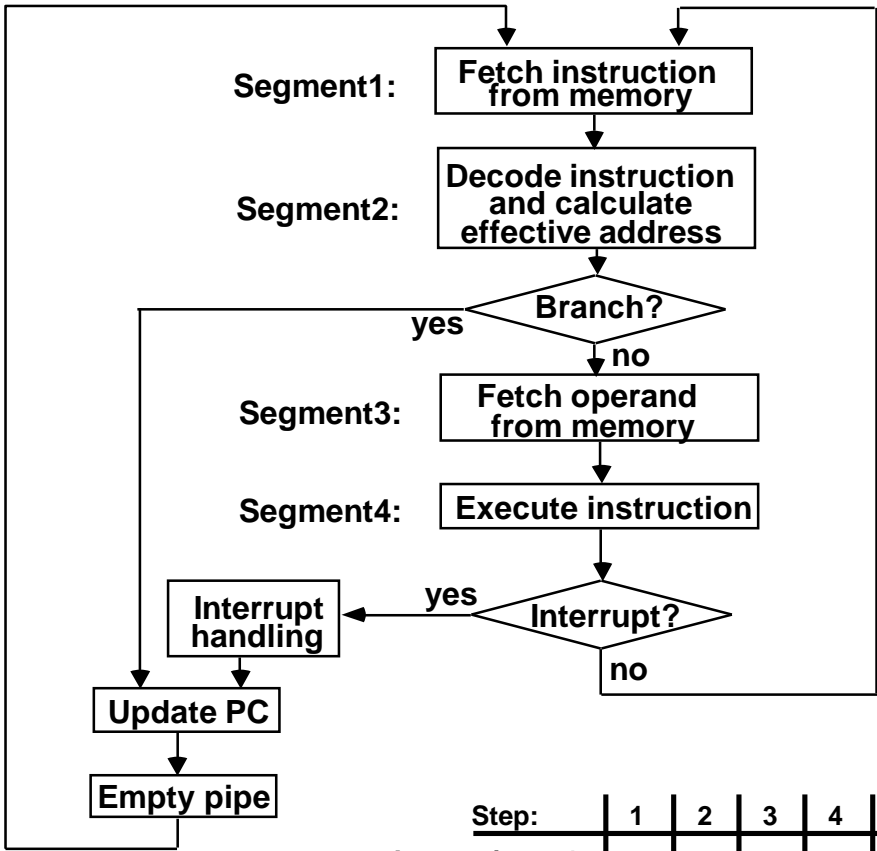
Conventional



Pipelined



INSTRUCTION EXECUTION IN A 4-STAGE PIPELINE



Step:		1	2	3	4	5	6	7	8	9	10	11	12	13
Instruction	1	FI	DA	FO	EX									
	2		FI	DA	FO	EX								
(Branch)	3			FI	DA	FO	EX							
	4				FI	-	-	FI	DA	FO	EX			
	5					-	-	-	FI	DA	FO	EX		
	6									FI	DA	FO	EX	
	7										FI	DA	FO	EX

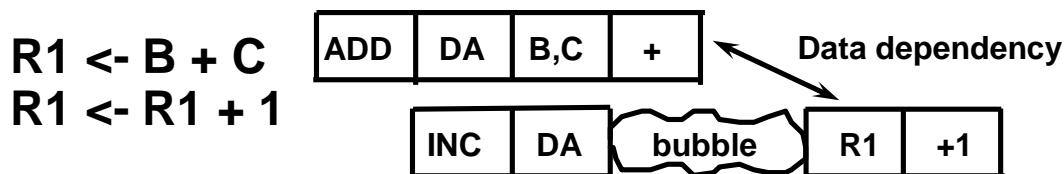
MAJOR HAZARDS IN PIPELINED EXECUTION

Structural hazards(Resource Conflicts)

Hardware Resources required by the instructions in simultaneous overlapped execution cannot be met

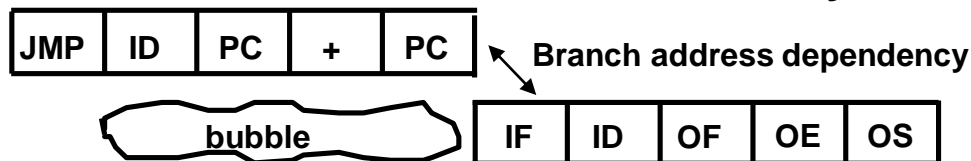
Data hazards (Data Dependency Conflicts)

An instruction scheduled to be executed in the pipeline requires the result of a previous instruction, which is not yet available



Control hazards

Branches and other instructions that change the PC make the fetch of the next instruction to be delayed



Hazards in pipelines may make it necessary to **stall** the pipeline



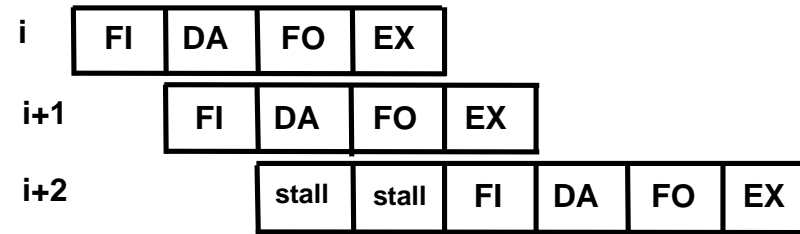
Pipeline Interlock:
Detect Hazards Stall until it is cleared

STRUCTURAL HAZARDS

Structural Hazards

Occur when some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute

Example: With one memory-port, a data and an instruction fetch cannot be initiated in the same clock



The Pipeline is stalled for a structural hazard
 <- Two Loads with one port memory
 -> Two-port memory will serve without stall

DATA HAZARDS

Data Hazards

Occurs when the execution of an instruction depends on the results of a previous instruction

```
ADD    R1, R2, R3
SUB    R4, R1, R5
```

Data hazard can be dealt with either hardware techniques or software technique

Hardware Technique

Interlock

- hardware detects the data dependencies and delays the scheduling of the dependent instruction by stalling enough clock cycles

Forwarding (bypassing, short-circuiting)

- Accomplished by a data path that routes a value from a source (usually an ALU) to a user, bypassing a designated register. This allows the value to be produced to be used at an earlier stage in the pipeline than would otherwise be possible

Software Technique

Instruction Scheduling(compiler) for *delayed load*

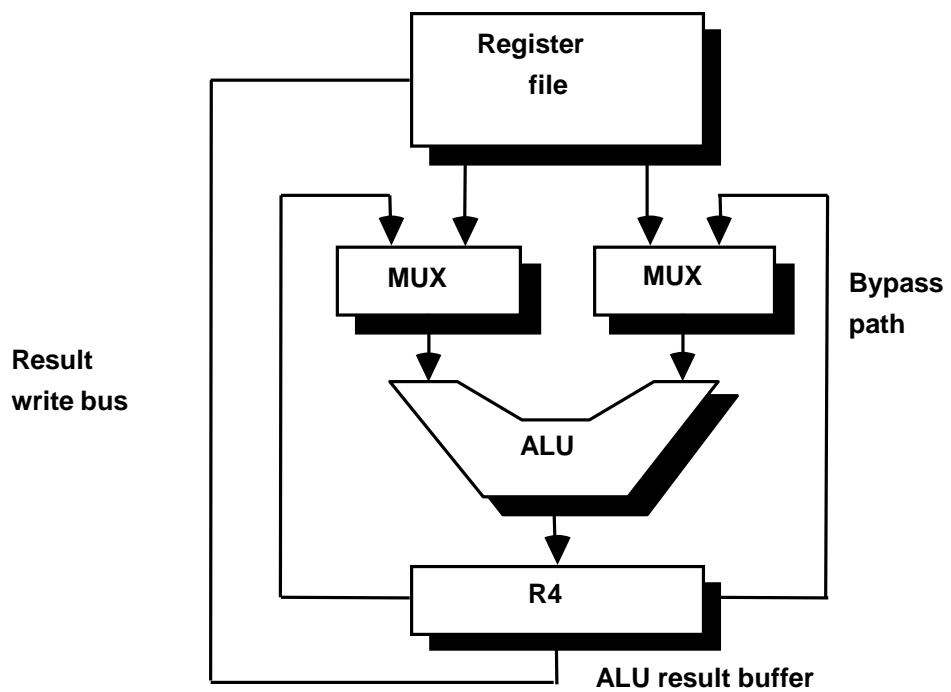
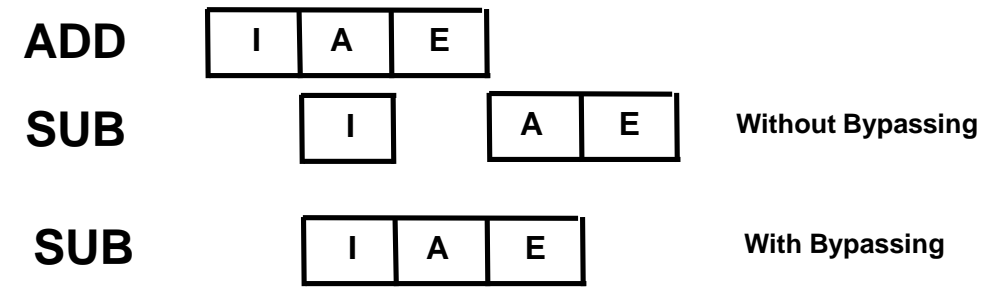
FORWARDING HARDWARE

Example:

```
ADD    R1, R2, R3
SUB     R4, R1, R5
```

3-stage Pipeline

- I: Instruction Fetch
- A: Decode, Read Registers, ALU Operations
- E: Write the result to the destination register



INSTRUCTION SCHEDULING

$a = b + c;$
 $d = e - f;$

Unscheduled code:

```

    LW    Rb, b
    LW    Rc, c
→ ADD    Ra, Rb, Rc
→ SW     a, Ra
    LW    Re, e
    LW    Rf, f
→ SUB    Rd, Re, Rf
→ SW     d, Rd
    
```

Scheduled Code:

```

    LW    Rb, b
    LW    Rc, c
    LW    Re, e
    ADD    Ra, Rb, Rc
    LW    Rf, f
    SW     a, Ra
    SUB    Rd, Re, Rf
→ SW     d, Rd
    
```

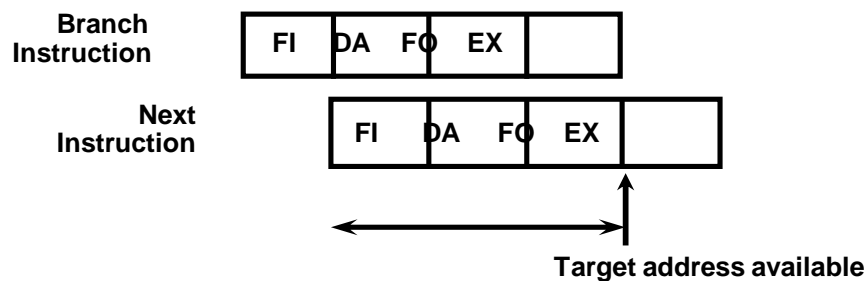
Delayed Load

A load requiring that the following instruction not use its result

CONTROL HAZARDS

Branch Instructions

- Branch target address is not known until the branch instruction is completed



- Stall -> waste of cycle times

Dealing with Control Hazards

- * Prefetch Target Instruction
- * Branch Target Buffer
- * Loop Buffer
- * Branch Prediction
- * Delayed Branch

CONTROL HAZARDS

Prefetch Target Instruction

- Fetch instructions in both streams, branch not taken and branch taken
- Both are saved until branch branch is executed. Then, select the right instruction stream and discard the wrong stream

Branch Target Buffer(BTB; Associative Memory)

- Entry: Addr of previously executed branches; Target instruction and the next few instructions
- When fetching an instruction, search BTB.
- If found, fetch the instruction stream in BTB;
- If not, new stream is fetched and update BTB

Loop Buffer(High Speed Register file)

- Storage of entire loop that allows to execute a loop without accessing memory

Branch Prediction

- Guessing the branch condition, and fetch an instruction stream based on the guess. Correct guess eliminates the branch penalty

Delayed Branch

- Compiler detects the branch and rearranges the instruction sequence by inserting useful instructions that keep the pipeline busy in the presence of a branch instruction

RISC PIPELINE

RISC

- Machine with a very fast clock cycle that executes at the rate of one instruction per cycle
- <- Simple Instruction Set
 - Fixed Length Instruction Format
 - Register-to-Register Operations

Instruction Cycles of Three-Stage Instruction Pipeline

Data Manipulation Instructions

- I: Instruction Fetch
- A: Decode, Read Registers, ALU Operations
- E: Write a Register

Load and Store Instructions

- I: Instruction Fetch
- A: Decode, Evaluate Effective Address
- E: Register-to-Memory or Memory-to-Register

Program Control Instructions

- I: Instruction Fetch
- A: Decode, Evaluate Branch Address
- E: Write Register(PC)

DELAYED LOAD

LOAD: R1 ← M[address 1]
LOAD: R2 ← M[address 2]
ADD: R3 ← R1 + R2
STORE: M[address 3] ← R3

Three-segment pipeline timing

Pipeline timing with data conflict

clock cycle	1	2	3	4	5	6
Load R1	I	A	E			
Load R2		I	A	E		
Add R1+R2			I	A	E	
Store R3				I	A	E

Pipeline timing with delayed load

clock cycle	1	2	3	4	5	6	7
Load R1	I	A	E				
Load R2		I	A	E			
NOP			I	A	E		
Add R1+R2				I	A	E	
Store R3					I	A	E

The data dependency is taken care by the compiler rather than the hardware

DELAYED BRANCH

Compiler analyzes the instructions before and after the branch and rearranges the program sequence by inserting useful instructions in the delay steps

Using no-operation instructions

Clock cycles:	1	2	3	4	5	6	7	8	9	10
1. Load	I	A	E							
2. Increment		I	A	E						
3. Add			I	A	E					
4. Subtract				I	A	E				
5. Branch to X					I	A	E			
6. NOP						I	A	E		
7. NOP							I	A	E	
8. Instr. in X								I	A	E

Rearranging the instructions

Clock cycles:	1	2	3	4	5	6	7	8
1. Load	I	A	E					
2. Increment		I	A	E				
3. Branch to X			I	A	E			
4. Add				I	A	E		
5. Subtract					I	A	E	
6. Instr. in X						I	A	E

VECTOR PROCESSING

Vector Processing Applications

- Problems that can be efficiently formulated in terms of vectors
 - Long-range weather forecasting
 - Petroleum explorations
 - Seismic data analysis
 - Medical diagnosis
 - Aerodynamics and space flight simulations
 - Artificial intelligence and expert systems
 - Mapping the human genome
 - Image processing

Vector Processor (computer)

Ability to process vectors, and related data structures such as matrices and multi-dimensional arrays, much faster than conventional computers

Vector Processors may also be pipelined

VECTOR PROGRAMMING

```

DO 20 I = 1, 100
20  C(I) = B(I) + A(I)
    
```

Conventional computer

```

Initialize I = 0
20  Read A(I)
    Read B(I)
    Store C(I) = A(I) + B(I)
    Increment I = i + 1
    If I ≤ 100 goto 20
    
```

Vector computer

```

C(1:100) = A(1:100) + B(1:100)
    
```

VECTOR INSTRUCTIONS

f1: $V * V$
f2: $V * S$
f3: $V \times V * V$
f4: $V \times S * V$

V: Vector operand
S: Scalar operand

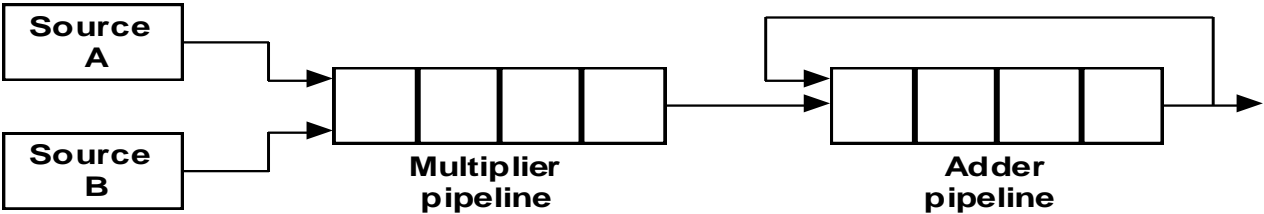
Type	Mnemonic	Description (I = 1, ..., n)
f1	VSQR	Vector square root $B(I) * \text{SQR}(A(I))$
	VSIN	Vector sine $B(I) * \sin(A(I))$
	VCOM	Vector complement $A(I) * A(I)$
f2	VSUM	Vector summation $S * \sum A(I)$
	VMAX	Vector maximum $S * \max\{A(I)\}$
f3	VADD	Vector add $C(I) * A(I) + B(I)$
	VMPY	Vector multiply $C(I) * A(I) * B(I)$
	VAND	Vector AND $C(I) * A(I) . B(I)$
	VLAR	Vector larger $C(I) * \max(A(I), B(I))$
	VTGE	Vector test > $C(I) * 0$ if $A(I) < B(I)$ $C(I) * 1$ if $A(I) > B(I)$
f4	SADD	Vector-scalar add $B(I) * S + A(I)$
	SDIV	Vector-scalar divide $B(I) * A(I) / S$

VECTOR INSTRUCTION FORMAT

Vector Instruction Format

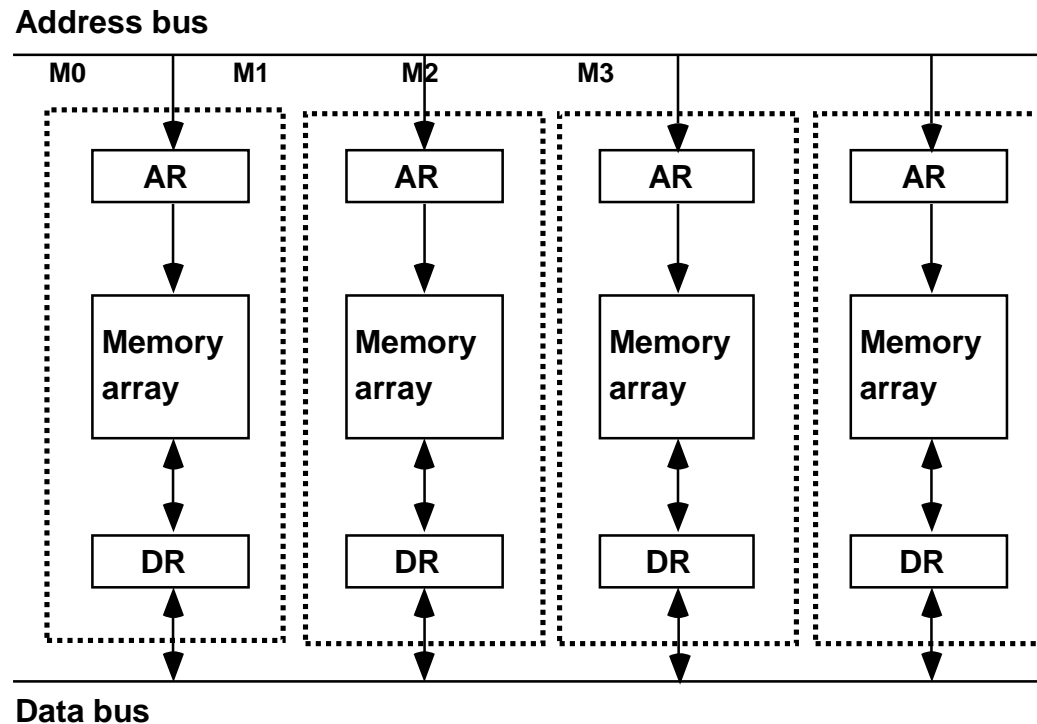
Operation code	Base address source 1	Base address source 2	Base address destination	Vector length
----------------	-----------------------	-----------------------	--------------------------	---------------

Pipeline for Inner Product



MULTIPLE MEMORY MODULE AND INTERLEAVING

Multiple Module Memory



Address Interleaving

Different sets of addresses are assigned to different memory modules

MULTIPROCESSORS

- **Characteristics of Multiprocessors**
- **Interconnection Structures**
- **Interprocessor Arbitration**
- **Interprocessor Communication
and Synchronization**
- **Cache Coherence**

TERMINOLOGY

Parallel Computing

Simultaneous use of multiple processors, all components of a single architecture, to solve a task. Typically processors identical, single user (even if machine multiuser)

Distributed Computing

Use of a network of processors, each capable of being viewed as a computer in its own right, to solve a problem. Processors may be heterogeneous, multiuser, usually individual task is assigned to a single processors

Concurrent Computing

All of the above?

TERMINOLOGY

Supercomputing

Use of fastest, biggest machines to solve big, computationally intensive problems. Historically machines were vector computers, but parallel/vector or parallel becoming the norm

Pipelining

Breaking a task into steps performed by different units, and multiple inputs stream through the units, with next input starting in a unit when previous input done with the unit but not necessarily done with the task

Vector Computing

Use of vector processors, where operation such as multiply broken into several steps, and is applied to a stream of operands (“vectors”). Most common special case of pipelining

Systolic

Similar to pipelining, but units are not necessarily arranged linearly, steps are typically small and more numerous, performed in lockstep fashion. Often used in special-purpose hardware such as image or signal processors

SPEEDUP AND EFFICIENCY

A: Given problem

$T^*(n)$: Time of best sequential algorithm to solve an instance of A of size n on 1 processor

$T_p(n)$: Time needed by a given parallel algorithm and given parallel architecture to solve an instance of A of size n, using p processors

Note: $T^*(n) \leq T_1(n)$

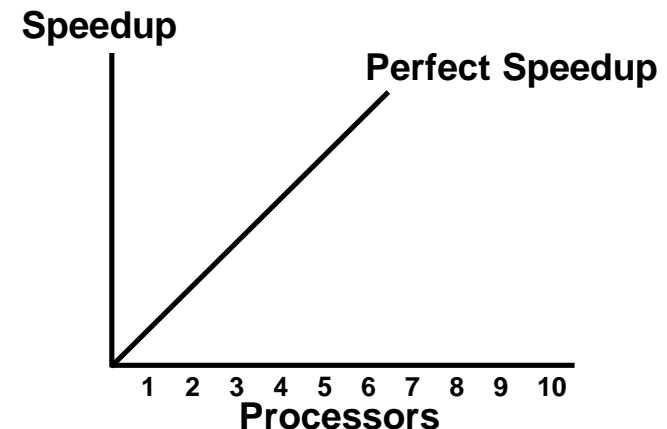
Speedup: $T^*(n) / T_p(n)$

Efficiency: $T^*(n) / [pT_p(n)]$

Speedup should be between 0 and p, and

Efficiency should be between 0 and 1

Speedup is *linear* if there is a constant $c > 0$ so that speedup is always at least cp.



AMDAHL'S LAW

Given a program

f : Fraction of time that represents operations
that must be performed serially

Maximum Possible Speedup: S

$$S \leq \frac{1}{f + (1 - f) / p}, \text{ with } p \text{ processors}$$

$$S < 1 / f, \text{ with unlimited number of processors}$$

- Ignores possibility of new algorithm, with much smaller f
- Ignores possibility that more of program is run from higher speed memory such as Registers, Cache, Main Memory
- Often problem is scaled with number of processors, and f is a function of size which may be decreasing (Serial code may take constant amount of time, independent of size)

FLYNN'S HARDWARE TAXONOMY

I: Instruction Stream

D: Data Stream

$$\begin{bmatrix} M \\ S \end{bmatrix} I \quad \begin{bmatrix} M \\ S \end{bmatrix} D$$

SI: Single Instruction Stream

- All processors are executing the same instruction in the same cycle
- Instruction may be conditional
- For Multiple processors, the control processor issues an instruction

MI: Multiple Instruction Stream

- Different processors may be simultaneously executing different instructions

SD: Single Data Stream

- All of the processors are operating on the same data items at any given time

MD: Multiple Data Stream

- Different processors may be simultaneously operating on different data items

SISD : standard serial computer

MISD : very rare

MIMD and **SIMD** : Parallel processing computers

COUPLING OF PROCESSORS

Tightly Coupled System

- **Tasks and/or processors communicate in a highly synchronized fashion**
- **Communicates through a common shared memory**
- **Shared memory system**

Loosely Coupled System

- **Tasks or processors do not communicate in a synchronized fashion**
- **Communicates by message passing packets**
- **Overhead for data exchange is high**
- **Distributed memory system**

GRANULARITY OF PARALLELISM

Granularity of Parallelism

Coarse-grain

- A task is broken into a handful of pieces, each of which is executed by a powerful processor
- Processors may be heterogeneous
- Computation/communication ratio is very high

Medium-grain

- Tens to few thousands of pieces
- Processors typically run the same code
- Computation/communication ratio is often hundreds or more

Fine-grain

- Thousands to perhaps millions of small pieces, executed by very small, simple processors or through pipelines
- Processors typically have instructions broadcasted to them
- Compute/communicate ratio often near unity

MEMORY

Shared (Global) Memory

- A Global Memory Space accessible by all processors
- Processors may also have some local memory

Distributed (Local, Message-Passing) Memory

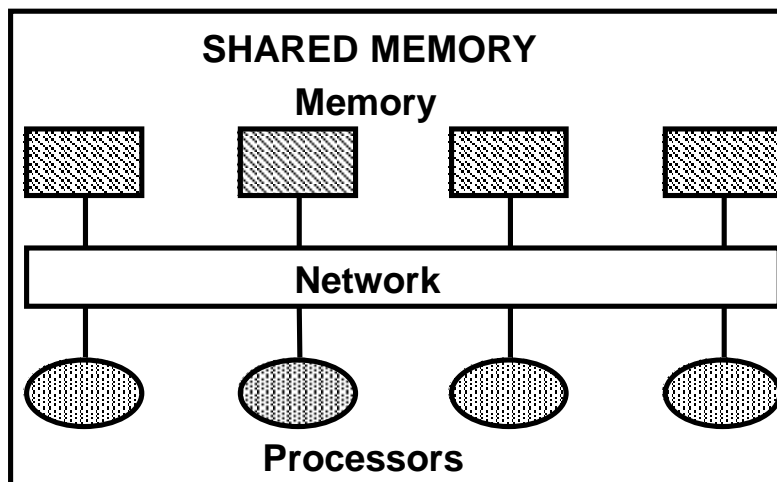
- All memory units are associated with processors
- To retrieve information from another processor's memory a message must be sent there

Uniform Memory

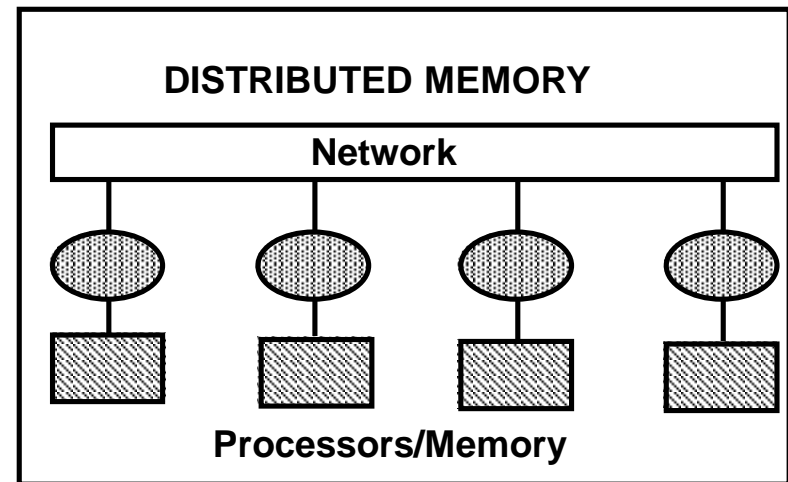
- All processors take the same time to reach all memory locations

Nonuniform (NUMA) Memory

- Memory access is not uniform

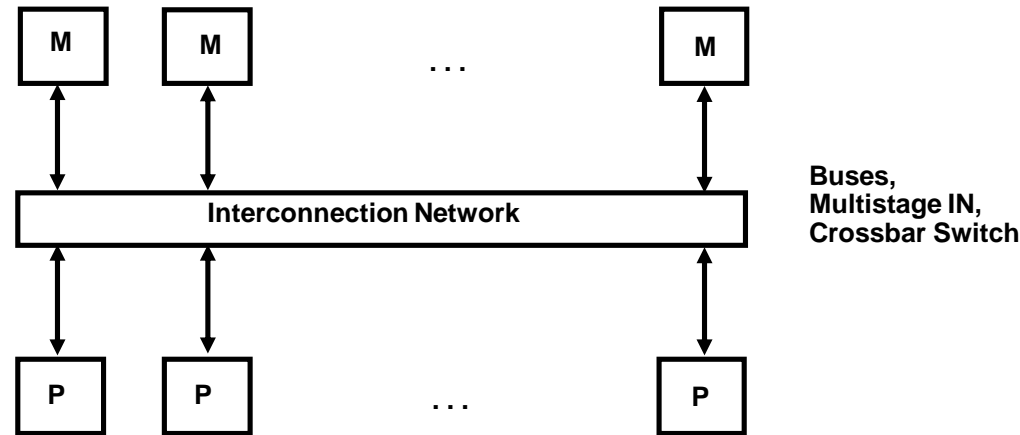


Computer Organization



Computer Architecture

SHARED MEMORY MULTIPROCESSORS



Characteristics

All processors have equally direct access to one large memory address space

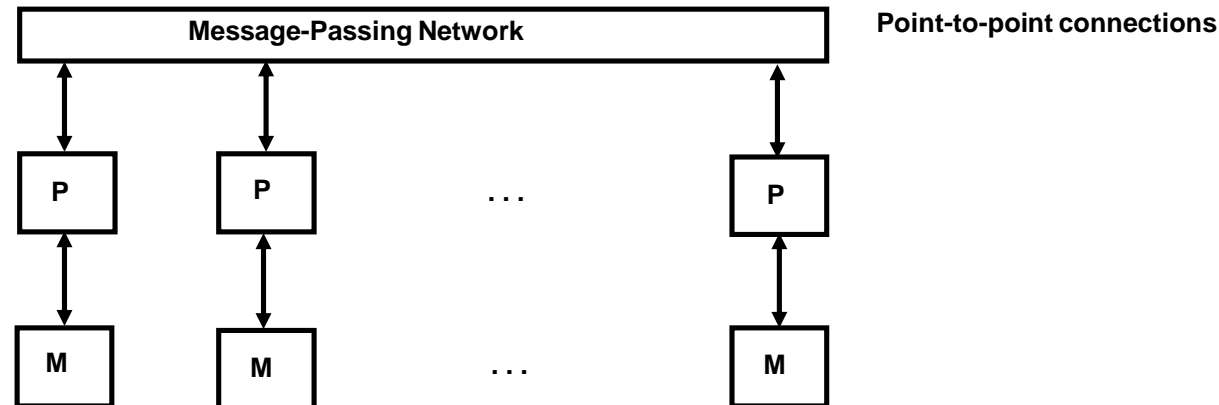
Example systems

- Bus and cache-based systems: Sequent Balance, Encore Multimax
- Multistage IN-based systems: Ultracomputer, Butterfly, RP3, HEP
- Crossbar switch-based systems: C.mmp, Alliant FX/8

Limitations

Memory access latency; Hot spot problem

MESSAGE-PASSING MULTIPROCESSORS



Characteristics

- Interconnected computers
- Each processor has its own memory, and communicate via message-passing

Example systems

- Tree structure: Teradata, DADO
- Mesh-connected: Rediflow, Series 2010, J-Machine
- Hypercube: Cosmic Cube, iPSC, NCUBE, FPS T Series, Mark III

Limitations

- Communication overhead; Hard to programming

INTERCONNECTION STRUCTURES

- * Time-Shared Common Bus
- * Multiport Memory
- * Crossbar Switch
- * Multistage Switching Network
- * Hypercube System

Bus

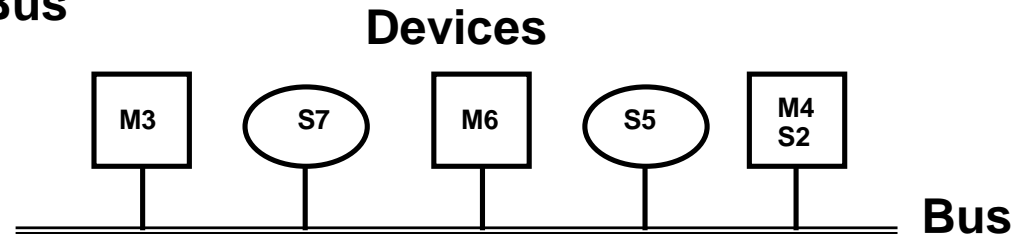
All processors (and memory) are connected to a common bus or busses

- Memory access is fairly uniform, but not very scalable

BUS

- A collection of signal lines that carry module-to-module communication
- Data highways connecting several digital system elements

Operations of Bus



M3 wishes to communicate with S5

- [1] M3 sends signals (address) on the bus that causes S5 to respond
- [2] M3 sends data to S5 or S5 sends data to M3(determined by the command line)

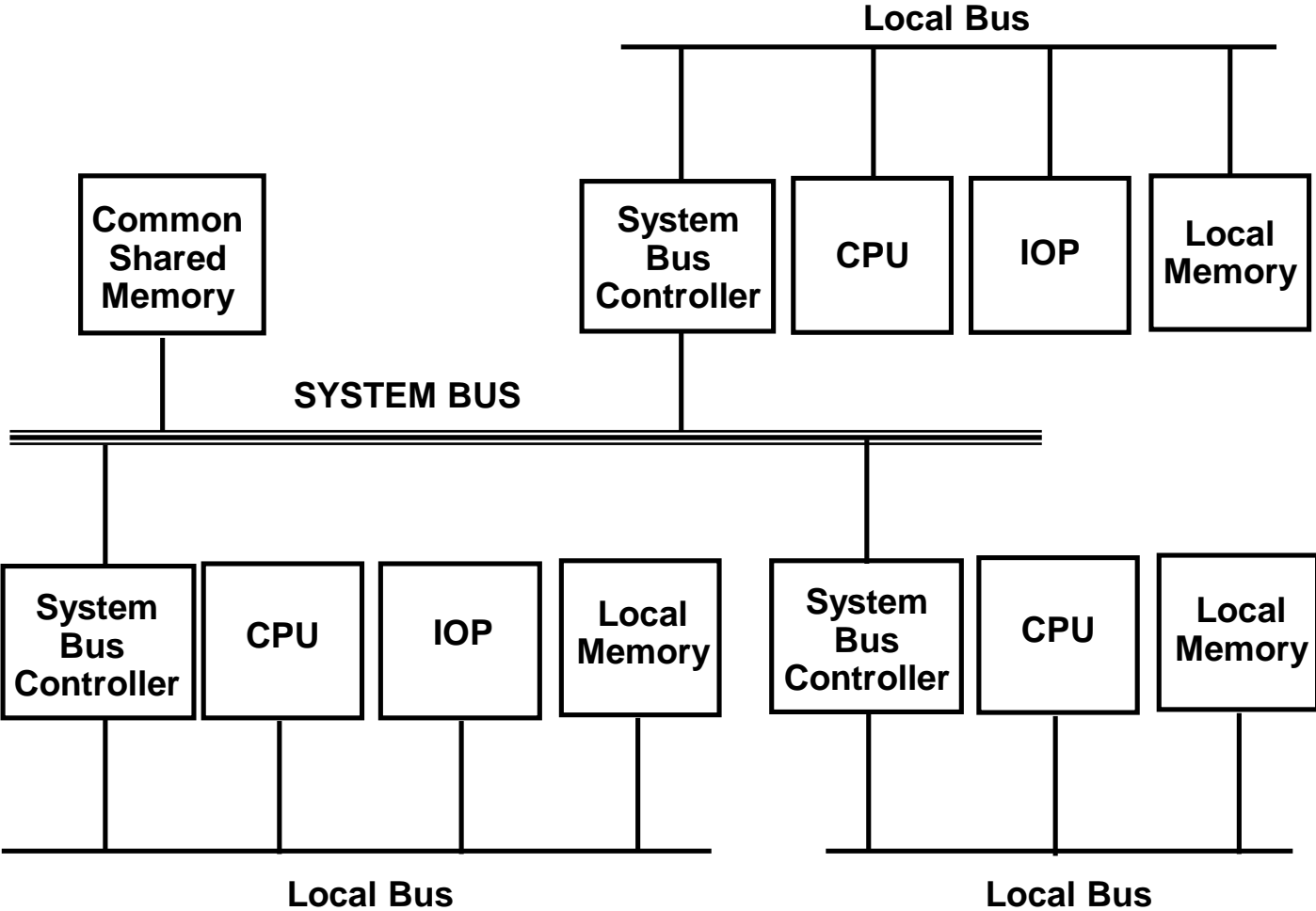
Master Device: Device that initiates and controls the communication

Slave Device: Responding device

Multiple-master buses

- > Bus conflict
- > need bus arbitration

SYSTEM BUS STRUCTURE FOR MULTIPROCESSORS



MULTIPORT MEMORY

Multiport Memory Module

- Each port serves a CPU

Memory Module Control Logic

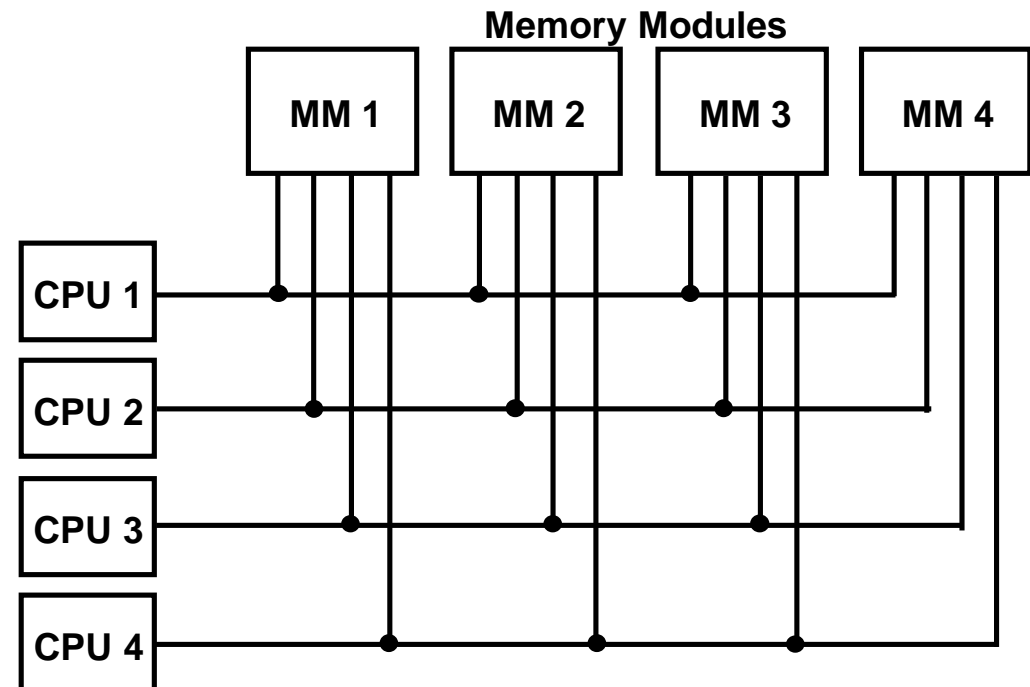
- Each memory module has control logic
- Resolve memory module conflicts Fixed priority among CPUs

Advantages

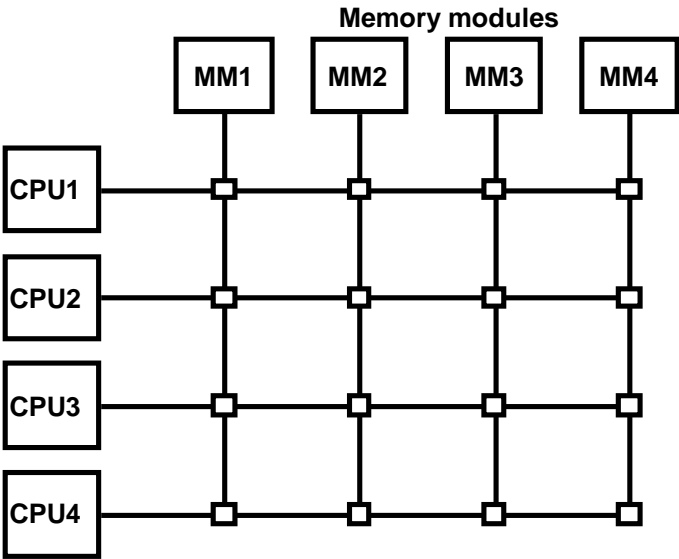
- Multiple paths -> high transfer rate

Disadvantages

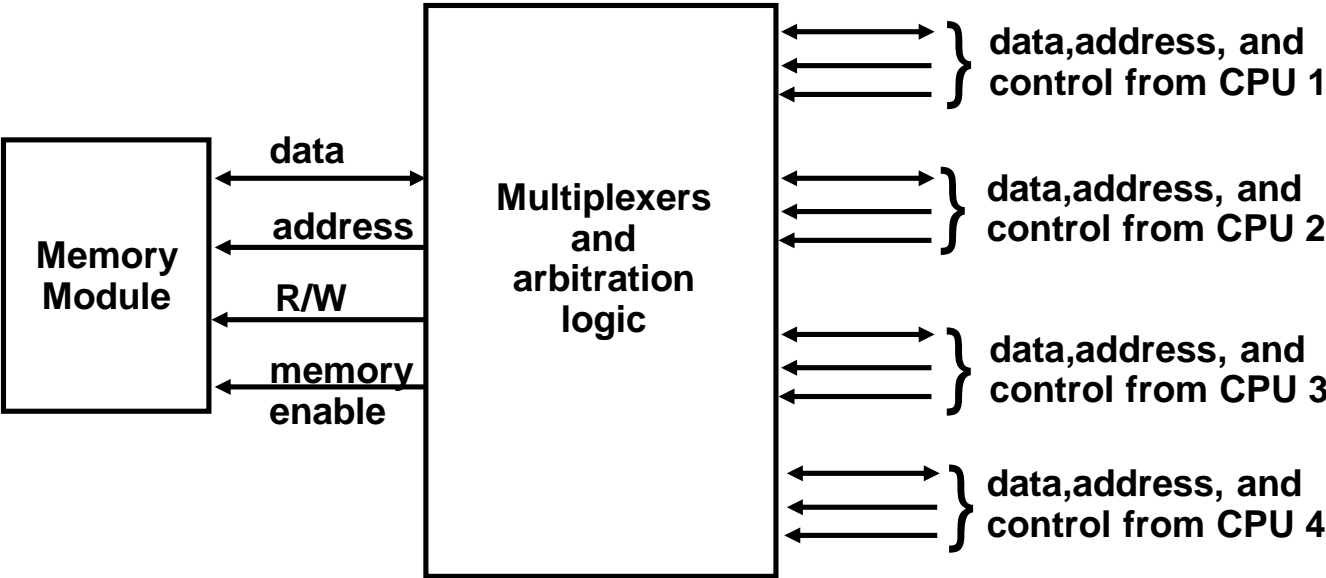
- Memory control logic
- Large number of cables and connections



CROSSBAR SWITCH

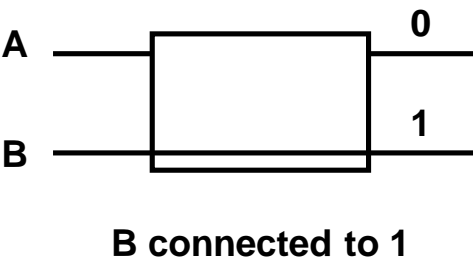
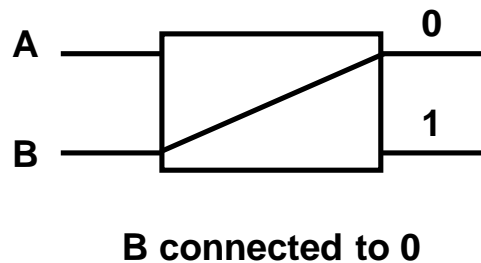
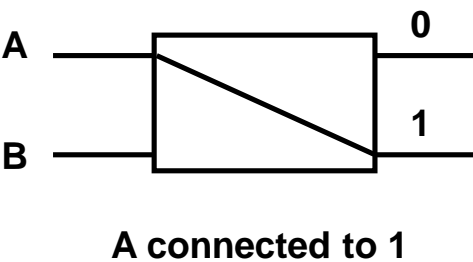
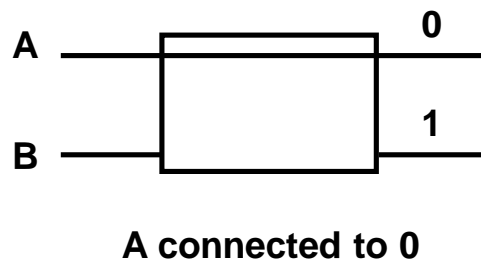


Block Diagram of Crossbar Switch



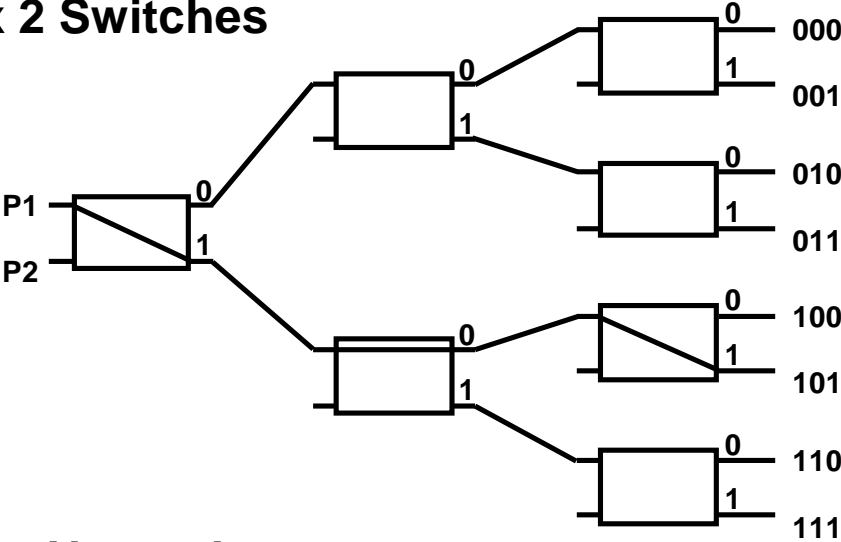
MULTISTAGE SWITCHING NETWORK

Interstage Switch

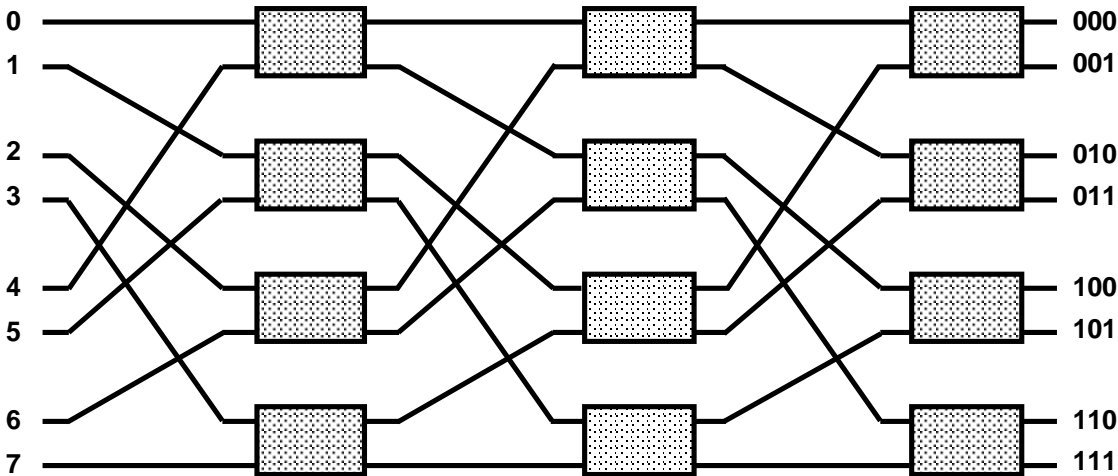


MULTISTAGE INTERCONNECTION NETWORK

Binary Tree with 2 x 2 Switches



8x8 Omega Switching Network



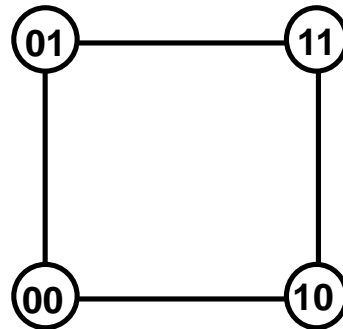
HYPERCUBE INTERCONNECTION

n-dimensional hypercube (binary n-cube)

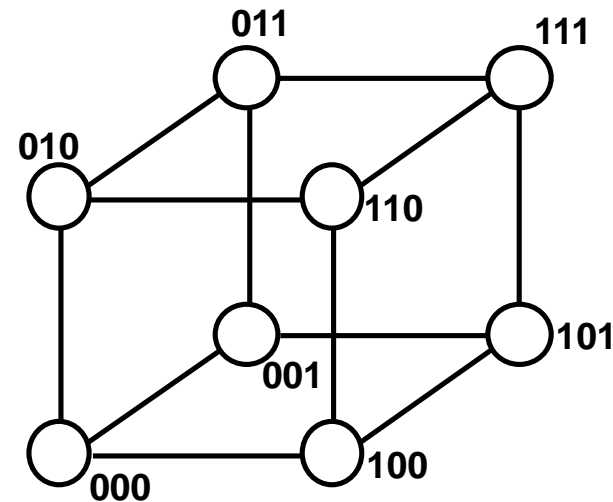
- $p = 2^n$
- processors are conceptually on the corners of a n-dimensional hypercube, and each is directly connected to the n neighboring nodes
- Degree = n



One-cube



Two-cube



Three-cube

INTERPROCESSOR ARBITRATION

Bus

- Board level bus
- Backplane level bus
- Interface level bus

System Bus - A Backplane level bus

- Printed Circuit Board
- Connects CPU, IOP, and Memory
- Each of CPU, IOP, and Memory board can be plugged into a slot in the backplane(system bus)
- Bus signals are grouped into 3 groups

Data, Address, and Control(plus power)

- Only one of CPU, IOP, and Memory can be granted to use the bus at a time
- Arbitration mechanism is needed to handle multiple requests

e.g. IEEE standard 796 bus	
	- 86 lines
Data:	16(multiple of 8)
Address:	24
Control:	26
Power:	20

SYNCHRONOUS & ASYNCHRONOUS DATA TRANSFER

Synchronous Bus

Each data item is transferred over a time slice known to both source and destination unit

- Common clock source
- Or separate clock and synchronization signal is transmitted periodically to synchronize the clocks in the system

Asynchronous Bus

- * Each data item is transferred by *Handshake* mechanism
 - Unit that transmits the data transmits a control signal that indicates the presence of data
 - Unit that receiving the data responds with another control signal to acknowledge the receipt of the data
- * Strobe pulse - supplied by one of the units to indicate to the other unit when the data transfer has to occur

BUS SIGNALS

- Bus signal allocation
- address
 - data
 - control
 - arbitration
 - interrupt
 - timing
 - power, ground

IEEE Standard 796 Multibus Signals

Data and address	
Data lines (16 lines)	DATA0 - DATA15
Address lines (24 lines)	ADRS0 - ADRS23
Data transfer	
Memory read	MRDC
Memory write	MWTC
IO read	IORC
IO write	IOWC
Transfer acknowledge	TACK (XACK)
Interrupt control	
Interrupt request	INT0 - INT7
interrupt acknowledge	INTA

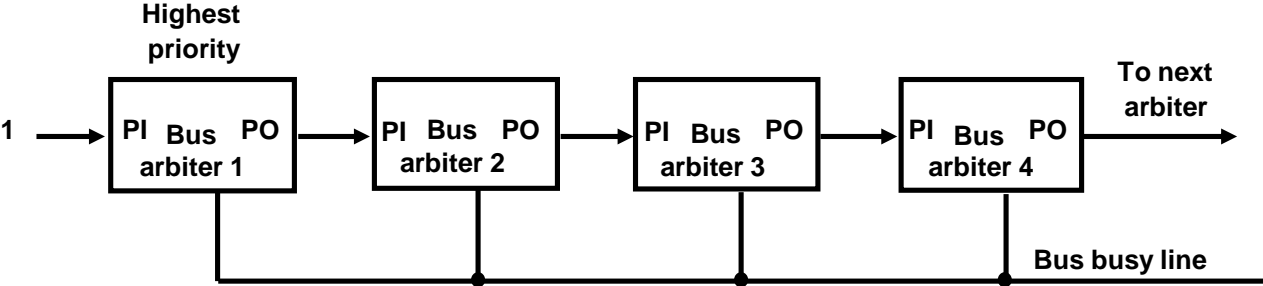
BUS SIGNALS

IEEE Standard 796 Multibus Signals (Cont'd)

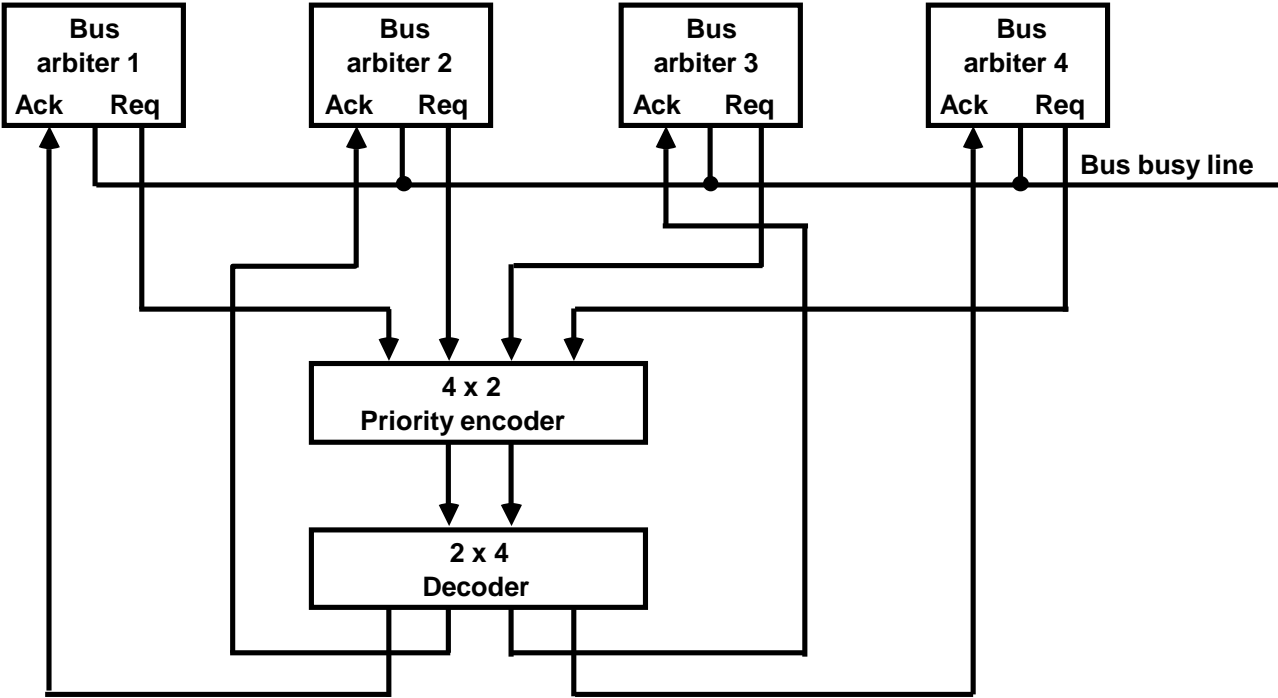
Miscellaneous control	
Master clock	CCLK
System initialization	INIT
Byte high enable	BHEN
Memory inhibit (2 lines)	INH1 - INH2
Bus lock	LOCK
Bus arbitration	
Bus request	BREQ
Common bus request	CBRQ
Bus busy	BUSY
Bus clock	BCLK
Bus priority in	BPRN
Bus priority out	BPRO
Power and ground (20 lines)	

INTERPROCESSOR ARBITRATION STATIC ARBITRATION

Serial Arbitration Procedure



Parallel Arbitration Procedure



INTERPROCESSOR ARBITRATION DYNAMIC ARBITRATION

Priorities of the units can be dynamically changeable while the system is in operation

Time Slice

Fixed length time slice is given sequentially to each processor, round-robin fashion

Polling

Unit address polling - Bus controller advances the address to identify the requesting unit

LRU

FIFO

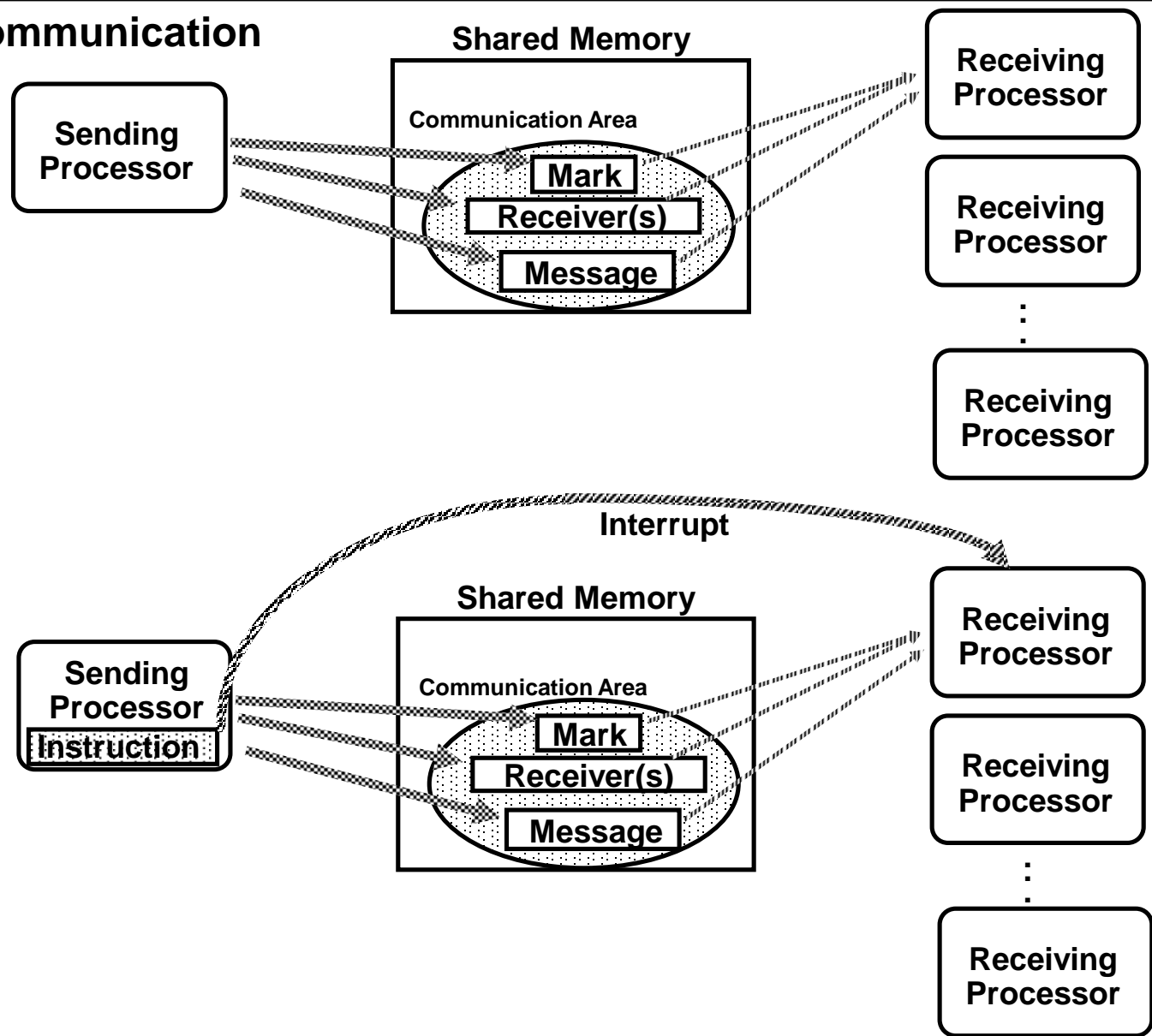
Rotating Daisy Chain

Conventional Daisy Chain - Highest priority to the nearest unit to the bus controller

Rotating Daisy Chain - Highest priority to the unit that is nearest to the unit that has most recently accessed the bus(it becomes the bus controller)

INTERPROCESSOR COMMUNICATION

Interprocessor Communication



INTERPROCESSOR SYNCHRONIZATION

Synchronization

Communication of control information between processors

- To enforce the correct sequence of processes
- To ensure mutually exclusive access to shared writable data

Hardware Implementation

Mutual Exclusion with a *Semaphore*

Mutual Exclusion

- One processor to exclude or lock out access to shared resource by other processors when it is in a *Critical Section*
- Critical Section is a program sequence that, once begun, must complete execution before another processor accesses the same shared resource

Semaphore

- A binary variable
- 1: A processor is executing a critical section, that not available to other processors
- 0: Available to any requesting processor
- Software controlled Flag that is stored in memory that all processors can be access

SEMAPHORE

Testing and Setting the Semaphore

- Avoid two or more processors test or set the same semaphore
- May cause two or more processors enter the same critical section at the same time
- Must be implemented with an indivisible operation

R <- M[SEM]	/ Test semaphore /
M[SEM] <- 1	/ Set semaphore /

These are being done while *locked*, so that other processors cannot test and set while current processor is being executing these instructions

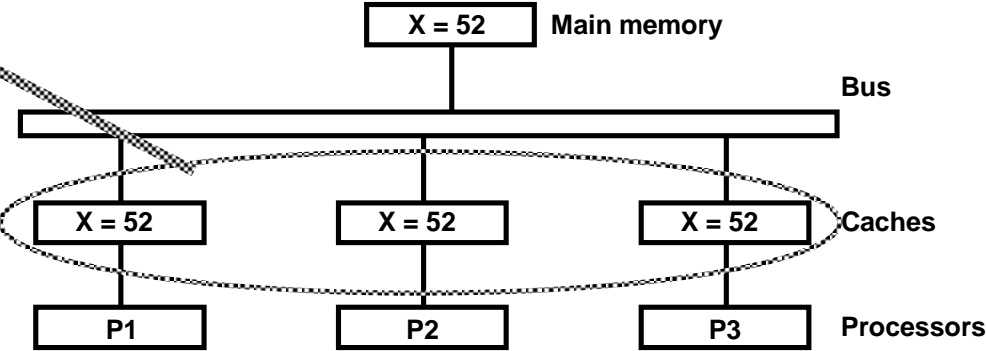
If R=1, another processor is executing the critical section, the processor executed this instruction does not access the shared memory

If R=0, available for access, set the semaphore to 1 and access

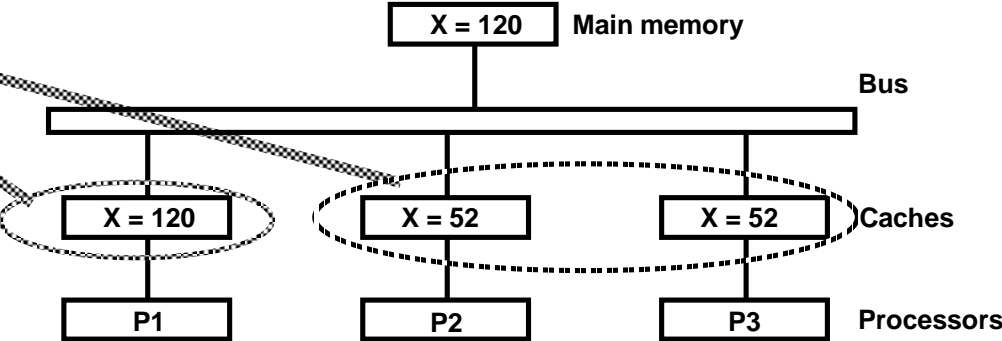
The last instruction in the program must clear the semaphore

CACHE COHERENCE

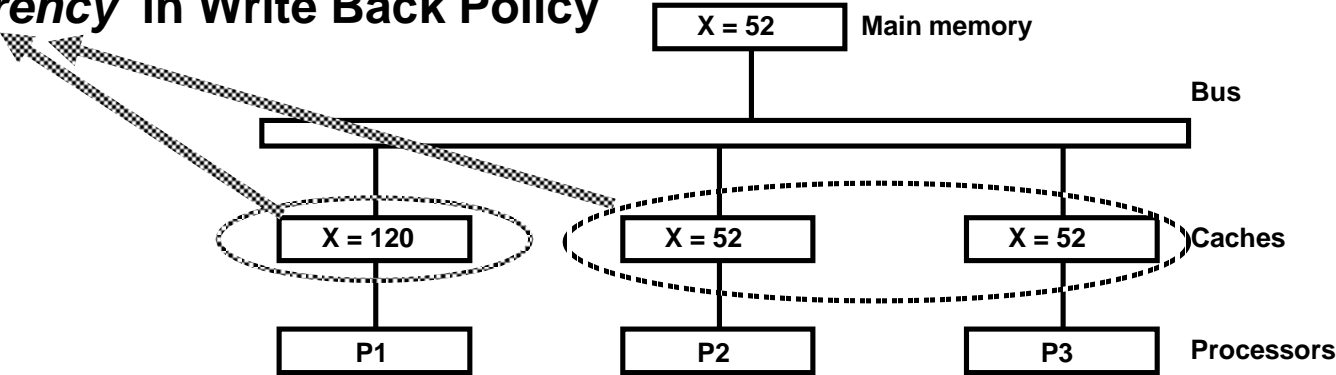
Caches are Coherent



Cache Incoherency in Write Through Policy



Cache Incoherency in Write Back Policy



MAINTAINING CACHE COHERENCY

Shared Cache

- Disallow private cache
- Access time delay

Software Approaches

* Read-Only Data are Cacheable

- Private Cache is for Read-Only data
- Shared Writable Data are not cacheable
- Compiler tags data as cacheable and noncacheable
- Degrade performance due to software overhead

* Centralized Global Table

- Status of each memory block is maintained in CGT: RO(Read-Only); RW(Read and Write)
- All caches can have copies of RO blocks
- Only one cache can have a copy of RW block

Hardware Approaches

* Snoopy Cache Controller

- Cache Controllers monitor all the bus requests from CPUs and IOPs
- All caches attached to the bus monitor the write operations
- When a word in a cache is written, memory is also updated (write through)
- Local snoopy controllers in all other caches check their memory to determine if they have a copy of that word; If they have, that location is marked invalid(future reference to this location causes cache miss)

PARALLEL COMPUTING

Grosche's Law

Grosch's Law states that the speed of computers is proportional to the square of their cost. Thus if you are looking for a fast computer, you are better off spending your money buying one large computer than two small computers and connecting them.

Grosch's Law is true within classes of computers, but not true between classes. Computers may be priced according to Groach's Law, but the Law cannot be true asymptotically.

Minsky's Conjecture

Minsky's conjecture states that the speedup achievable by a parallel computer increases as the logarithm of the number of processing elements, thus making large-scale parallelism unproductive.

Many experimental results have shown linear speedup for over 100 processors.

PARALLEL COMPUTING

History

History tells us that the speed of traditional single CPU Computers has increased 10 folds every 5 years. Why should great effort be expended to devise a parallel computer that will perform tasks 10 times faster when, by the time the new architecture is developed and implemented, single CPU computers will be just as fast.

Utilizing parallelism is better than waiting.

Amdahl's Law

A small number of sequential operations can effectively limit the speedup of a parallel algorithm.

Let f be the fraction of operations in a computation that must be performed sequentially, where $0 < f < 1$. Then the maximum speedup S achievable by a parallel computer with p processors performing the computation is $S < 1 / [f + (1 - f) / p]$. For example, if 10% of the computation must be performed sequentially, then the maximum speedup achievable is 10, no matter how many processors a parallel computer has.

There exist some parallel algorithms with almost no sequential operations. As the problem size(n) increases, f becomes smaller ($f \rightarrow 0$ as $n \rightarrow \infty$). In this case, $\lim_{n \rightarrow \infty} S = p$.

PARALLEL COMPUTING

Pipelined Computers are Sufficient

Most supercomputers are vector computers, and most of the successes attributed to supercomputers have accomplished on pipelined vector processors, especially Cray-1 and Cyber-205.

If only vector operations can be executed at high speed, supercomputers will not be able to tackle a large number of important problems. The latest supercomputers incorporate both pipelining and high level parallelism (e.g., Cray-2)

Software Inertia

Billions of dollars worth of FORTRAN software exists. Who will rewrite them? Virtually no programmers have any experience with a machine other than a single CPU computer. Who will retrain them ?

INTERCONNECTION NETWORKS

Switching Network (Dynamic Network)

Processors (and Memory) are connected to routing switches like in telephone system

- Switches might have queues(combining logic), which improve functionality but increase latency
- Switch settings may be determined by message headers or preset by controller
- Connections can be packet-switched or circuit-switched(remain connected as long as it is needed)
- Usually NUMA, blocking, often scalable and upgradable

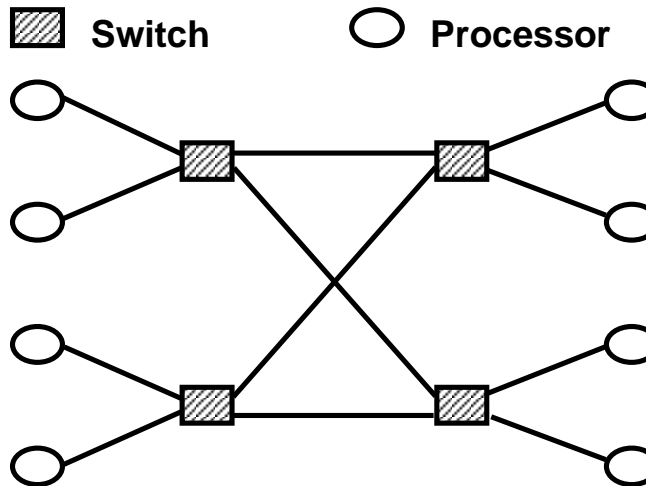
Point-Point (Static Network)

Processors are directly connected to only certain other processors and must go multiple hops to get to additional processors

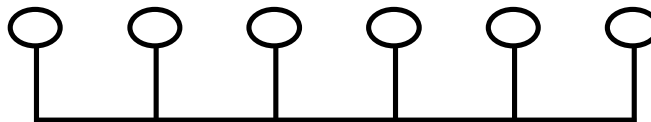
- Usually distributed memory
- Hardware may handle only single hops, or multiple hops
- Software may mask hardware limitations
- Latency is related to graph diameter, among many other factors
- Usually NUMA, nonblocking, scalable, upgradable
- Ring, Mesh, Torus, Hypercube, Binary Tree

INTERCONNECTION NETWORKS

Multistage Interconnect



Bus



INTERCONNECTION NETWORKS

Static Topology - Direct Connection

- Provide a direct inter-processor communication path
- Usually for distributed-memory multiprocessor

Dynamic Topology - Indirect Connection

- Provide a physically separate switching network for inter-processor communication
- Usually for shared-memory multiprocessor

Direct Connection

Interconnection Network

A graph $G(V,E)$

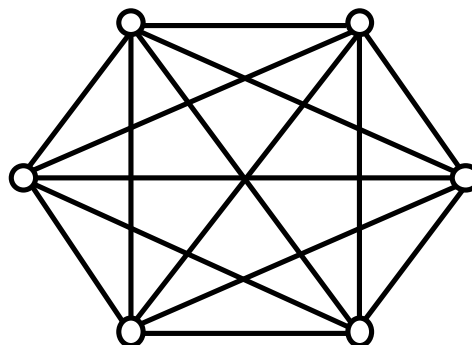
V: a set of processors (nodes)

E: a set of wires (edges)

Performance Measures: - degree, diameter, etc

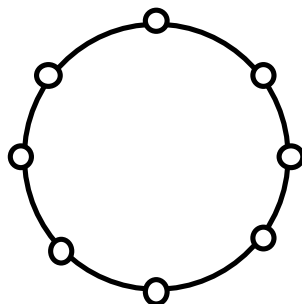
INTERCONNECTION NETWORKS

Complete connection



- Every processor is directly connected to every other processors
- Diameter = 1, Degree = $p - 1$
- # of wires = $p (p - 1) / 2$; dominant cost
- Fan-in/fanout limitation makes it impractical for large p
- Interesting as a theoretical model because algorithm bounds for this model are automatically lower bounds for all direct connection machines

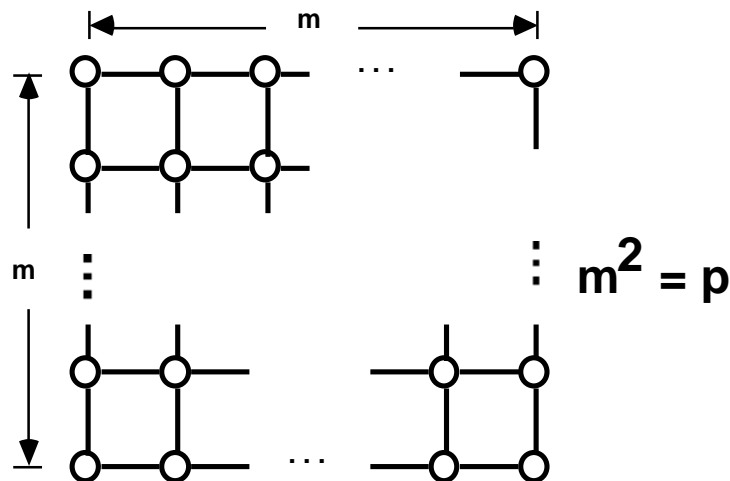
Ring



- Degree = 2, (not a function of p)
- Diameter = $\lfloor p/2 \rfloor$

INTERCONNECTION NETWORKS

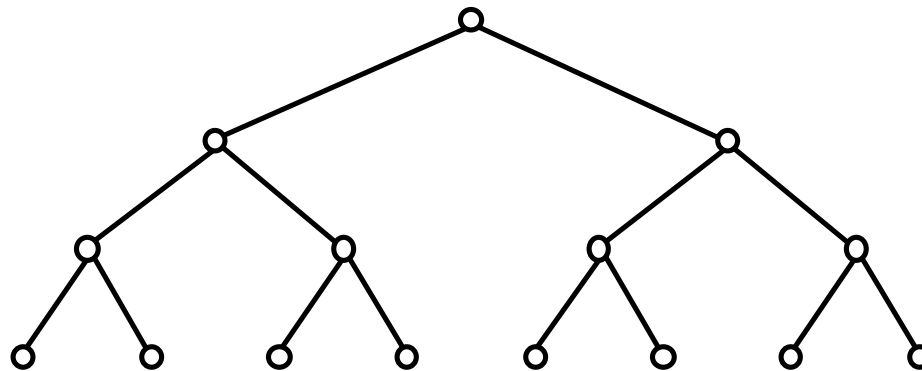
• 2-Mesh



- Degree = 4
- Diameter = $2(m - 1)$
- In general, an n -dimensional mesh has
diameter = $d (p^{1/n} - 1)$
- Diameter can be halved by having wrap-around connections (-> Torus)
- Ring is a 1-dimensional mesh with wrap-around connection

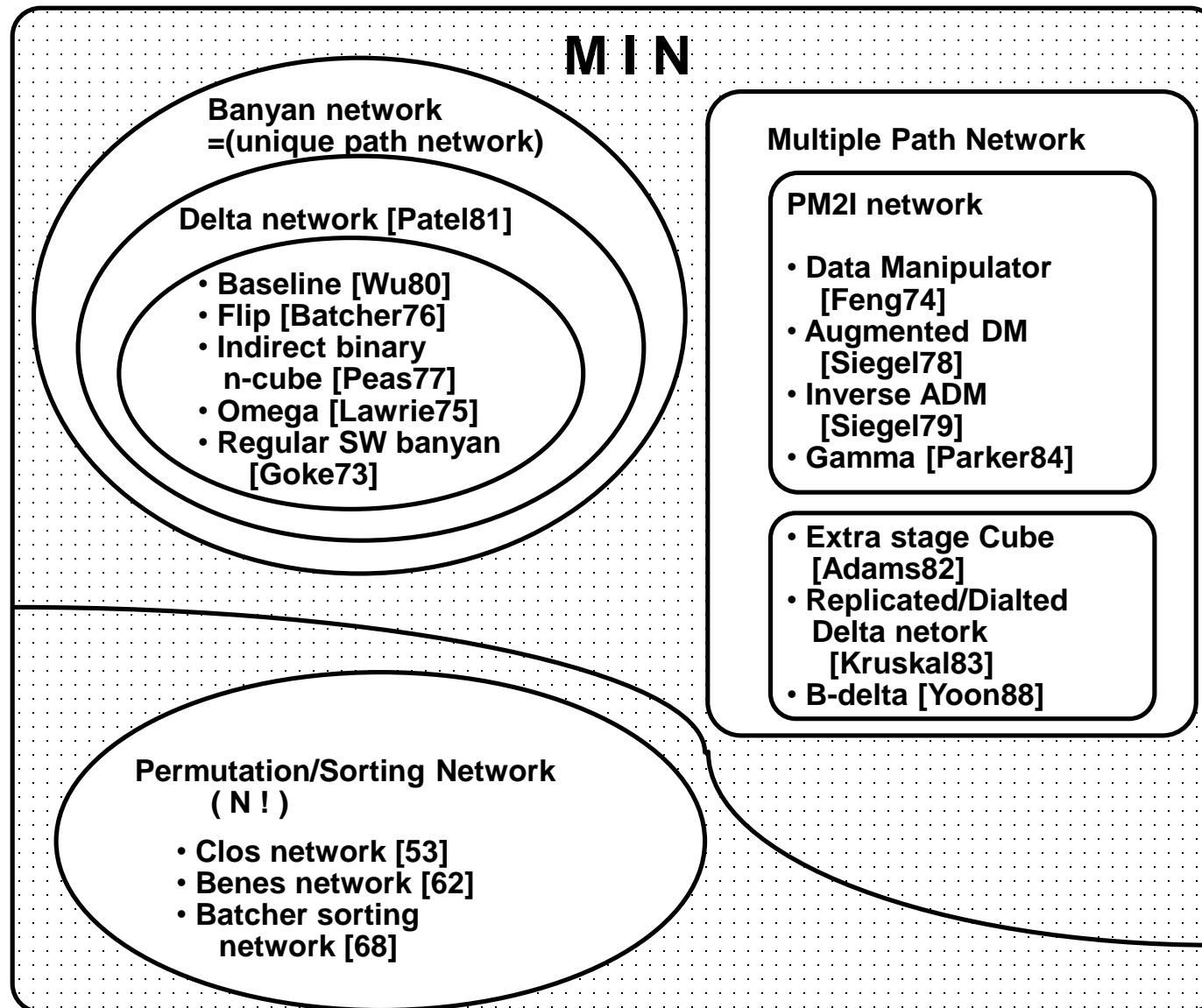
INTERCONNECTION NETWORK

Binary Tree



- Degree = 3
- Diameter = $2 \log \frac{p+1}{2}$

MIN SPACE



SOME CURRENT PARALLEL COMPUTERS

DM-SIMD

- AMT DAP
- Goodyear MPP
- Thinking Machines CM series
- MasPar MP1
- IBM GF11

SM-MIMD

- Alliant FX
- BBN Butterfly
- Encore Multimax
- Sequent Balance/Symmetry
- CRAY 2, X-MP, Y-MP
- IBM RP3
- U. Illinois CEDAR

DM-MIMD

- Intel iPSC series, Delta machine
- NCUBE series
- Meiko Computing Surface
- Carnegie-Mellon/ Intel iWarp

14. Feedback on Curriculum Design and development:

15.CO/PO attainment, analysis and Action taken report:

Name of the subject: CO&A									Yr/Sem:-II/I	
Batch: 21		Academic Year: 2022-23							Branch: CSE-A	
Course Attainment										
Final Direct Course Attainment									Final Indirect Course Attainment Calculation	
	Mid 1	Mid 2	Quiz 1	Quiz 2	Assign-1	Assign-2	Internal	University		
CO 1									CO 1	
CO 2									CO 2	
CO 3									CO 3	
CO 4									CO 4	
CO 5									CO 5	
Attainment									Final Indirect Course attainment	
Weightage										
Direct Total Attainment										
final direct course attainment										
Weightage										
Total Attainment										
Course Attainment										

CO to PO Attainment (2022-23)

Course Outcomes (CO's)	Program Outcomes (PO's)												PSO's		
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO 1	PSO 2	PSO 3
TOTAL															

Formula: $(1/3 * \text{average of PO} * \text{course attainment})$

