

ADVANCED COMPUTER ARCHITECTURE

(A Practical Approach)

• Parallel Algorithms • Parallel Programming • Super Computers

For :

- B. Tech. (CSE)-7th sem., GGSIPU.
- M. Tech (CSE)-1st sem., GGSIPU.
- B. Tech (CSE)-7th & 8th sem., UPTU.
- B. Tech (CSE)-5th sem., Pune Univ.
- MCA (SE)-5th sem., GGSIPU.
- B. Tech (IT)-7th & 8th sem., UPTU.
- M. Tech (CSE)-1st sem., P.T.U.
- B.E. (CSE)-7th sem., KUD.

Er. RAJIV CHOPRA

B.E. (CSE), M.I.T. & M. Tech. (IT)
(USIT, GGSIPU, CSE/IT – Delhi)

Asst. Professor
GTBIT, Delhi

CCEH-(Crezone Certified Ethical Hacker)



S. CHAND & COMPANY PVT. LTD.
(AN ISO 9001 : 2008 COMPANY)
RAM NAGAR, NEW DELHI-110 055

S.CHAND & COMPANY PVT. LTD.



(An ISO 9001 : 2008 Company)

Head Office: 7361, RAM NAGAR, NEW DELHI - 110 055

Phone: 23672080-81-82, 9899107446, 9911310888

Fax: 91-11-23677446

Shop at: schandgroup.com; e-mail: info@schandgroup.com

Branches :

Ahmedabad	: 1st Floor, Heritage, Near Gujarat Vidhyapeeth, Ashram Road, Ahmedabad - 380 014, Ph: 27541965, 27542369, ahmedabad@schandgroup.com
Bengaluru	: No. 6, Ahuja Chambers, 1st Cross, Kumara Krupa Road, Bengaluru - 560 001, Ph: 22268048, 22354008, bangalore@schandgroup.com
Bhopal	: Bajaj Tower, Plot No. 283, Lala Lajpat Rai Colony, Raisen Road, Bhopal - 462 011, Ph: 4274723, 4209587, bhopal@schandgroup.com
Chandigarh	: S.C.O. 2419-20, First Floor, Sector - 22-C (Near Aroma Hotel), Chandigarh -160 022, Ph: 2725443, 2725446, chandigarh@schandgroup.com
Chennai	: No.1, Whites Road, Opposite Express Avenue, Royapettah, Chennai - 600014 Ph: 28410027, 28410058, chennai@schandgroup.com
Coimbatore	: 1790, Trichy Road, LGB Colony, Ramanathapuram, Coimbatore -6410045, Ph: 2323620, 4217136 coimbatore@schandgroup.com (Marketing Office)
Cuttack	: 1st Floor, Bhartia Tower, Badambadi, Cuttack - 753 009, Ph: 2332580; 2332581, cuttack@schandgroup.com
Dehradun	: 1st Floor, 20, New Road, Near Dwarka Store, Dehradun - 248 001, Ph: 2711101, 2710861, dehradun@schandgroup.com
Guwahati	: Dilip Commercial (1st floor), M.N. Road, Pan Bazar, Guwahati - 781 001, Ph: 2738811, 2735640 guwahati@schandgroup.com
Hyderabad	: Padma Plaza, H.No. 3-4-630, Opp. Ratna College, Narayanaguda, Hyderabad - 500 029, Ph: 27550194, 27550195, hyderabad@schandgroup.com
Jaipur	: 1st Floor, Nand Plaza, Hawa Sadak, Ajmer Road, Jaipur - 302 006, Ph: 2219175, 2219176, jaipur@schandgroup.com
Jalandhar	: Mai Hiran Gate, Jalandhar - 144 008, Ph: 2401630, 5000630, jalandhar@schandgroup.com
Kochi	: Kachapilly Square, Mullassery Canal Road, Ernakulam, Kochi - 682 011, Ph: 2378740, 2378207-08, cochin@schandgroup.com
Kolkata	: 285/J, Bipin Bihari Ganguli Street, Kolkata - 700 012, Ph: 22367459, 22373914, kolkata@schandgroup.com
Lucknow	: Mahabeer Market, 25 Gwynne Road, Aminabad, Lucknow - 226 018, Ph: 4076971, 4026791, 4065646, 4027188, lucknow@schandgroup.com
Mumbai	: Blackie House, 11nd Floor, 103/5, Walchand Hirachand Marg, Opp. G.P.O., Mumbai - 400 001, Ph: 22690881, 22610885, mumbai@schandgroup.com
Nagpur	: Karnal Bagh, Near Model Mill Chowk, Nagpur - 440 032, Ph: 2720523, 2777666 nagpur@schandgroup.com
Patna	: 104, Citicentre Ashok, Mahima Palace , Govind Mitra Road, Patna - 800 004, Ph: 2300489, 2302100, patna@schandgroup.com
Pune	: 291, Flat No.-16, Ganesh Gayatri Complex, 11nd Floor, Somwarpeth, Near Jain Mandir, Pune - 411 011, Ph: 64017298, pune@schandgroup.com (Marketing Office)
Raipur	: Kailash Residency, Plot No. 4B, Bottle House Road, Shankar Nagar, Raipur - 492 007, Ph: 2443142, Mb. : 09981200834, raipur@schandgroup.com (Marketing Office)
Ranchi	: Flat No. 104, Sri Draupadi Smriti Apartments, (Near of Jaipal Singh Stadium) Neel Ratan Street, Upper Bazar, Ranchi - 834 001, Ph: 2208761, ranchi@schandgroup.com (Marketing Office)
Siliguri	: 122, Raja Ram Mohan Roy Road, East Vivekanandapally, P.O., Siliguri, Siliguri-734001, Dist. Jalpaiguri, (W.B.) Ph. 0353-2520750, siliguri@schandgroup.com (Marketing Office)
Visakhapatnam	: No. 49-54-15/53/8, Plot No. 7, 1st Floor, Opp. Radhakrishna Towers, Seethammadhara North Extn., Visakhapatnam - 530 013, Ph-2782609 (M) 09440100555, visakhapatnam@schandgroup.com (Marketing Office)

© 2009, Rajiv Chopra

All rights reserved. No part of this publication may be reproduced or copied in any material form (including photo copying or storing it in any medium in form of graphics, electronic or mechanical means and whether or not transient or incidental to some other use of this publication) without written permission of the copyright owner.
Any breach of this will entail legal action and prosecution without further notice.

Jurisdiction : All disputes with respect to this publication shall be subject to the jurisdiction of the Courts, tribunals and forums of New Delhi only.

First Edition 2009

Subsequent Editions and Reprints 2010, 2011

Revised Third Edition 2013

ISBN : 81-219-3077-4

Code : 20B 115

PRINTED IN INDIA

By Rajendra Ravindra Printers Pvt. Ltd., 7361, Ram Nagar, New Delhi -110 055

and published by S. Chand & Company Pvt. Ltd., 7361, Ram Nagar, New Delhi -110 055.

PREFACE TO THE THIRD EDITION

Many books on Advanced Computer Architecture are available in the market but they are not complete, are very formal and dry. My attempt is to make ACA very simple so that a student feels as if the teacher is sitting behind him and guiding him. This text is bolstered with many examples and CASE STUDIES. Also included in this text are the experiments to be performed in ACA lab. Every effort has been made to alleviate the treatment of the book for easy flow of understanding of the students as well as the professors alike.

Any suggestions to further enhance the quality of the book will be highly acknowledged.

Er. RAJIV CHOPRA

E-mail.raj_74chopra2004@yahoo.com

Disclaimer : While the author of this book have made every effort to avoid any mistake or omission and have used their skill, expertise and knowledge to the best of their capacity to provide accurate and updated information. The author and S. Chand does not give any representation or warranty with respect to the accuracy or completeness of the contents of this publication and are selling this publication on the condition and understanding that they shall not be made liable in any manner whatsoever. S.Chand and the author expressly disclaim all and any liability/responsibility to any person, whether a purchaser or reader of this publication or not, in respect of anything and everything forming part of the contents of this publication. S. Chand shall not be responsible for any errors, omissions or damages arising out of the use of the information contained in this publication. Further, the appearance of the personal name, location, place and incidence, if any; in the illustrations used herein is purely coincidental and work of imagination. Thus the same should in no manner be termed as defamatory to any individual.

ACKNOWLEDGEMENTS

Writing a good book not only requires sincere efforts and hard work but also the blessings of the omnipresent GOD. I thank the support given by S. Bhatnagar, Mr. Madan, his colleagues and the entire team of S. Chand for publishing this book in a nice format. I also acknowledge the support given by my parents-DR. J. R. CHOPRA (Ex-Professor of Physics) and Mrs. Sushma Chopra (Ex-Principal) during the writing of this book. Lastly, I acknowledge my twin babies- Arjeesh (son) and Arshitha (daughter) and my wife Shakti Chopra for their patience and motivations.

Er. Rajiv Chopra

Syllabus of GGSIPU

Code No.: ETCS 403

L T C

Paper : Advanced Computer Architecture (B. Tech. – CSE) 3 1 4

INSTRUCTION TO PAPER SETTERS :

1. Questions No. 1 should be compulsory and cover the entire syllabus. This question should have objective or short answer type questions. It should be of 25 marks.
2. Apart from question no. 1, rest of the paper shall consist of four units as per the syllabus. Every unit should have two questions. However, student may be asked to attempt only 1 questions from each unit. Each question should be of 12.5 marks.

UNIT - I

1. **Parallel computer models** : The state of computing, Multiprocessors and multicamputers, Multivector and SIMD computers, Architectural development tracks.
2. **Program and network properties** : Conditions of parallelism, Data and resource dependences, Hardware and software parallelism, Program partitioning and scheduling, Grain size and latency, Program flow mechanisms, Control flow versus data flow, Data flow architecture, Demand driven mechanisms, Comparisons of flow mechanisms

[No. of Hrs.: 11]

UNIT - II

3. **System Interconnect Architectures** : Network properties and routing, Static interconnection networks, Dynamic interconnection Networks, Multiprocessor system interconnects, Hierarchical bus systems, Crossbar switch and multiport memory, Multistage and combining network.
4. **Processors and Memory Hierarchy** : Advanced processor technology, Instruction-set Architecture, CISC Scalar Processors, RISC Scalar Processors, Superscalar Processors, VLIW Architectures, Vector and Symbolic processors.
5. **Memory Technology** : Hierarchical memory technology, Inclusion, Coherence and Locality, Memory capacity planning, Virtual Memory Technology. [No. of Hrs . : 11]

UNIT - III

6. **Backplane Bus System** : Backplane bus specification, Addressing and timing protocols, Arbitration transaction and interrupt, Cache addressing models, Direct mapping and associative caches.
7. **Pipelining** : Linear pipeline processor, Nonlinear pipeline processor, Instruction pipeline design, Mechanisms for instruction pipelining, Dynamic instruction scheduling, Branch handling techniques, Arithmetaic Pipeline Design, Computer arithmetic principles, Static arithmetic pipeline, Multifunctional arithmetic pipelines. [No. of Hrs. 11]

UNIT - IV

8. **Vector Processing Principles** : Vector instruction types, Vector-access memory schemes.
9. **Synchronous Parallel Processing** : SIMD Architecture and Programming Principles, SIMD Parallel Algorithms, SIMD Computers and Performance Enhancement.

CONTENTS

UNIT - I

Chapter 1 : Parallel Computer Models	1 – 56
1.0 Introduction to Parallel Computing	1
1.1 Need for Parallel Computing	1
1.2 Constraints of Conventional Architecture	2
1.2.1 Von Neumann of sequential Machines Architecture	2
1.2.2 Limitations	3
1.3 Computer Generations	3
1.4 The State of Computing	5
1.5 Evolution of Parallel Processors	6
1.5.1 Parallel Processing basic terminologies	6
1.5.2 Features of Parallel Processors	7
1.5.3 Evolution	7
1.5.4 Future trends	8
1.6 Parallelism in Uniprocessor System	8
1.6.1 Uniprocessor Architecture	8
1.6.2 Parallel Processing Mechanisms for Uniprocessors	9
1.6.2.1 Multiple functional units	10
1.6.2.2 Parallelism and pipelining within CPU	11
1.6.2.3 Overlapped CPU and I/O operations	11
1.6.2.4 Use of hierarchical memory system	11
1.6.2.5 Balancing of Subsystem Bandwidths	12
1.6.3 Multiprogramming and Time Sharing	14
1.7 Multiprocessors and Multicomputers	16
— UMA, NUMA, COMA and NORMA models	
1.8 Multivector and SIMD computers	22
1.9 Parallel Architectural Classification	
Schemes	23
1.9.1 Flynn's classification	24
1.9.2 Feng's classification	25
1.9.3 Handler's classification	28
1.9.4 Classification based on coupling between processing elements	29
1.9.5 Classification based on Modes of Access Memory	29
1.9.6 Classification based on Grain Size	29
1.10 Instruction Level Parallelism and Thread Level Parallelism	30
1.11 Performance of Parallel Processors—Metrics and Measures	30
1.12 Distributed Processing	37
1.13 Principles of Scalable Performance	38
1.14 Speed up Performance Laws	38
1.14.1 Amdahl's Law (for a fixed work load)	38

1.14.2 Gustafson's Law (for scaled problems)	40
1.14.3 Sun and Ni's Law	41
1.15 Case Study of Intel Itanium Processor	43
● Summary	44
● MCQs	44
● Conceptual Short Questions with Answers	46
● Exercise Questions	54
Chapter 2 : Program and Network Properties	57 – 88
2.0 Introduction	57
2.1 Conditions of Parallelism–Bernstein's Conditions	57
2.2 Types of Dependencies	60
2.3 Hardware and Software Parallelism	62
2.4 Program Partitioning and Scheduling	62
2.4.1 Levels of parallelism	63
2.5 Program Flow Mechanisms	68
2.6 Control Flow, Data Flow, Reduction Computers–Tabular form	68
2.7 Comparison of Control Flow, Data Flow and Demand Driven Computers	79
● Summary	80
● MCQs	80
● Conceptual Short Questions with Answers	81
● Exercise Questions	87

UNIT - II

Chapter 3 : System Interconnect Architectures	89 – 132
3.0 Introduction	89
3.1 Network Properties	89
3.2 Routing	90
3.3 Static versus Dynamic Interconnection Network	92
3.4 Network Topologies for Multiprocessor	92
3.5 Interprocessor Communication Network	93
3.5.1 Multiport Memory Model	109
3.5.2 Memory Contention	109
3.6 Structure of Parallel Computers	111
3.6.1 Shared Memory Multiprocessors	112
3.6.2 Message Passing Multicomputers	113
3.6.3 Pipelined Parallel computers	114
3.6.4 Array Processors	114
3.6.5 MMS	115
3.7 Comparison of Parallel Computer Architectures	117
● Summary	117
● MCQs	117
● Conceptual Short Questions with Answers	119
● Exercise Questions	130

Chapter 4 : Types of Processors	133 – 156
4.0 Introduction	133
4.1 Advanced Processor Technology	133
4.2 Instruction-Set Architectures	136
4.3 CISC Scalar Processors	136
4.4 RISC Scalar Processors	136
4.5 Comparison of CISC ad RISC—Tabular form	137
4.6 Superscalar Processors	138
4.7 VLIW Architectures	143
4.8 Comparison of Superscalar and VLIW—Tabular form	145
4.9 Vector and Symbolic Processors	145
4.10 Case Study on Pentium Processor (CISC)	146
4.11 Case Study on SPARC (RISC)	147
● Summary	148
● MCQs	149
● Conceptual Short Questions with Answers	150
● Exercise Questions	155
Chapter 5 : Memory Technology	157 – 175
5.0 Introduction	157
5.1 Hierarchical Memory Technology	157
5.2 Inclusion	158
5.3 Coherence	159
5.4 Locality of reference	160
5.5 Memory Capacity Planning	160
5.6 Virtual Memory Technology	162
5.7 Page Replacement Algorithms	164
5.7.1 FIFO algorithm : (First In First Out Algorithm)	164
5.7.2. Optimal page replacement algorithm	165
5.7.3. LRU (Least Recently used Algorithms)	166
5.7.4. Least frequently used Algorithm (MFU)	167
5.7.5. Most frequently used Algorithm (MFU)	167
5.7.6. Belady's Anomaly	167
● Summary	167
● MCQs	168
● Conceptual Short Questions with Answers	169
● Exercise Questions	174

UNIT - III

Chapter 6 : Backplane Bus System	176 – 217
6.0 Introduction	176
6.1 Backplane Bus Specification	176
6.2 Arbitration Schemes	178
6.3 Interrupt	180

6.4	Cache Addressing Models	180
6.4.1	Direct-mapping cache	181
6.4.2	Fully-associative cache	182
6.4.3	Set-associative cache	182
6.4.4	Sector-mapping cache	183
6.5	Cache performance issues	187
6.5.1	Cycle Counts	187
6.5.2	Hit Ratios	187
6.5.3	Block Sizes	187
6.5.4	Set Numbers	188
6.5.5	Other performance factors	188
6.6	Interleaved memory organization	188
6.7	Multi-Core Architectures & Cache Coherence problem	192
●	Summary	204
●	MCQs	204
●	Conceptual Short Questions with Answers	205
●	Exercise Questions	217

Chapter 7 : Pipelining **218 - 280**

7.0	Introduction	218
7.1	Pipeline–Principle and Implementation	218
7.1.1	Linear pipeline processor	218
7.1.1.1	Asynchronous Model	218
7.1.1.2	Synchronous Model	219
7.1.2	Clock Period	220
7.1.3	Speed up	220
7.1.4	Efficiency	220
7.1.5	Throughput	221
7.2	Non-linear pipeline processor	221
7.3	Classification of Pipeline Processor	222
7.3.1	Based on levels of processing–Handler's classification	222
7.3.1.1	Arithematic pipelining	222
7.3.1.2	Instruction pipelining	222
7.3.1.3	Processor pipelining	222
7.3.2	Based on Pipeline Configurations–Ramamoorthy and Li's classification	223
7.3.2.1	Unifunctional versus multifunctional pipelines	223
7.3.2.2	Static versus dynamic pipelines	223
7.3.2.3	Scalar versus vector pipelines	224
7.3.3	General Pipelines	224
7.3.3.1	Reservation Tables	225
7.3.3.2	Latency Analysis	226
7.3.3.3	Collision-free scheduling–steps to follow	227
7.3.3.4	Problems based on Reservation tables	231
7.3.4	Arithematic Pipeline Design	235

7.3.4.1	Computer Arithmetic Principles	235
7.3.4.1.1	Fixed-Point operations	236
7.3.4.1.2	Floating-Point operations	236
7.3.4.1.3	Elementary functions	236
7.3.5	Arithmetic Pipeline Stages	236
7.3.6	Different pipeline design	237
7.3.6.1	Multiply Pipeline Design	237
7.3.6.2	Division Pipeline Design	239
7.3.7	Instruction Pipeline Design	241
7.3.7.1	Phases of Instruction Execution	241
7.3.7.2	Pipelined Instruction Processing	241
7.3.8	Mechanisms for Instruction Pipelining	242
7.3.8.1	Prefetch Buffers	242
7.3.8.2	Multiple functional units	243
7.3.8.3	Internal Data Forwarding and Register Tagging	244
7.3.9	Pipeline Hazards	247
7.3.9.1	Data Hazard	247
7.3.9.2	Control Hazard	248
7.3.9.3	Hazard Resolving Techniques	249
7.3.9.4	Problems based on Pipeline Hazard	249
7.3.10	Dynamic Instruction Scheduling	250
7.3.10.1	Static Scheduling	250
7.3.10.2	Register Scoreboarding	250
7.3.10.3	Tomasulo's Approach	250
7.3.11	Advanced Pipelining	255
7.3.11.1	Loop scheduling—Unrolling	255
7.3.11.2	Out-of-order Execution versus in-order Execution	256
7.3.11.3	Instruction scheduling	257
7.3.11.4	Trace Scheduling	257
7.3.11.5	Software Pipelining	257
●	Summary	260
●	MCQs	260
●	Conceptual Short Questions with Answers	261
●	Exercise Questions	276

UNIT - IV

Chapter 8 : Vector Processing	281 – 305	
8.0	Introduction	281
8.1	Comparison of Vector and Array Processors	282
8.2	Basic Vector Architecture and it's classification	283
8.3	Vector processing related terminology	286
8.4	Vector Instruction Types	287
8.5	Vector Performance Modeling	290

8.6	Vectorization	293
8.7	Design of a Vectorizing Compiler	293
8.8	Optimization of Vector Functions	294
8.9	Case Study : The Cray family and Cray-1	298
	● Summary	300
	● MCQs	300
	● Conceptual Short Questions with Answers	301
	● Exercise Questions	304
Chapter 9 : Synchronous Parallel Processing (SIMD)		306 – 329
9.0	Introduction	306
9.1	SIMD architecture	306
9.2	Masking and Data-Routing Mechanisms	308
9.3	Inter-PE communication	309
	9.3.1 Operation Mode	309
	9.3.2 Control Strategy	309
	9.3.3 Switching Methodology	309
	9.3.4 Network Topology	310
9.4	SIMD (Array Processors)-Inter-Connection networks	310
	9.4.1 Static Versus Dynamic Network	310
	9.4.2 Cube Interconnection Networks	311
	9.4.3 Hypercube Interconnection Networks	311
	9.4.4 Mesh connected Illiac Network	311
9.5	SIMD Parallel Algorithms	312
	9.5.1 Matrix Multiplication	312
	9.5.2 2D-Pipeline Systolic Arrays	313
	9.5.3 Sorting in Parallel	314
	9.5.4 Fast Fourier Transform (FFT)	321
	● Summary	323
	● MCQs	323
	● Conceptual Short Questions with Answers	324
	● Exercise Questions	328
Chapter 10 : Parallel Algorithms and Programming		330 – 367
10.0	Introduction	330
10.1	Need of Parallel Programming	330
10.2	Characteristic of Parallel Algorithms	330
10.3	Parallel Programming Techniques	333
10.4	Parallel Programming Models	333
	10.4.1 Message Passing Programming	332
	10.4.2 Shared Memory Programming	335
	10.4.3 Data Parallel Programming	337
10.5	Parallel Algorithms for Multiprocessors	337
	10.5.1 Classification of Parallel Algorithms	337

10.5.2 Synchronized Parallel Algorithms	338
10.5.3 Asynchronized Parallel Algorithm	339
10.6 Performance of Parallel Algorithms	341
10.7 Parallel Programming languages	341
10.7.1 Fortran-90	341
10.7.2 Occam	342
10.7.3 C-Linda	344
10.7.4 CCC	346
10.8 Problems based on Parallel Algorithms	348
10.8.1 Problem of Prime Numbers	348
10.8.2 Searching	350
10.8.3 Graph Algorithm-Moore's shortest path algorithm	351
10.8.4 Quick Sort Problem	353
10.8.5 Dictionary operations (Searches)	355
10.8.6 PRAM Model(s) Parallel Random Access Machine	355
10.8.7 Message Passing Libraries for Parallel Programming	358
● Summary	359
● MCQs	359
● Conceptual Short Questions with Answers	360
● Exercise Questions	366

UNIT - V

Chapter 11 : Multithreaded Architecture	368 – 402
11.0 Introduction	368
11.1 Latency Hiding Techniques	369
11.2 Principles of Multithreading	371
11.3 Multithreading Issues and Solutions	372
11.4 Multithreaded Architectures	373
11.5 Cluster Computing (CC)	374
11.5.1 Characteristics of CC	374
11.5.2 Why to use a cluster ?	374
11.5.3 Cluster types	375
11.5.4 How to do it ?	375
11.5.5 Pros and cons of CC	375
11.6 Neural Computing	376
11.7 Neural Network Software Packages	377
11.8 Annie Project (Applications of Neural Network for Industry in Europe) Project	381
11.9 Grid Computing	396
11.10 Comparison of Cluster Computing and Grid computing—Tabular form	397
11.11 Advantages and Disadvantages of Grids	397
● Summary	397
● MCQs	397

● Conceptual Short Questions with Answers	399
● Exercise Questions	402
Chapter 12 : Operating System Issues	403 – 418
12.0 Introduction	403
12.1 Classification of multiprocessor O.S.	403
12.2 Software Requirements of multiprocessor O.S.	406
12.3 O.S. requirements	406
12.4 Distributed scheduler-PVM (A case study)	407
12.5 PThreads in shared Memory systems (A case study)	412
● Summary	416
● MCQs	417
● Conceptual Short Questions with Answers	417
● Exercise Questions	418
Chapter 13 : OPENMP and MPI	419 – 434
13.0 Introduction	419
13.1 Goals of OPENMP	419
13.2. Fork – Join Model of Parallel Execution	419
13.3 OpenMP Clauses	421
13.4 Parallel Region Construct	422
13.5 Work-Sharing Constructs	423
13.6 Directive Binding and Nesting Rules	428
13.7 Message Passing Interfaces (MPI)	429
13.8 MPI Implementations	430
13.9 Advantages of MPI	434
Chapter 14 : OPTICAL COMPUTING – A CASE STUDY	435 – 444
APPENDIX	445–456
A : Glossary	445
B : Lecture plan of ACA	447
C : University's End-term Exam.	449
Question papers :	
● GGSIPU, Delhi	● MDU
● KUD, Karnataka	● IGNOU, Delhi
● DU, Delhi (DCE/DTU)	● RGTU
● UPTECH	
D : Suggested Experiments in ACA/Parallel Programming Lab. with Solutions	491
E : Bibliography	496
F : Latest developments	497
G : Jugene — Fastest Europe Supercomputer	498
H : Deeper thought	499

CHAPTER

1 PARALLEL COMPUTER MODELS

1.0 INTRODUCTION TO PARALLEL COMPUTING

Maximization of CPU performance has always been the lust for any computer architect. But certainly this is not an easy task. This is due to several reasons. One of the reasons is the simultaneous operation of complex multiple data paths. We remember Sidney Fembach's statement : "Today's large computers (mainframes) would have been considered 'Supercomputer', 10 to 20 years ago. Similarly, today's supercomputers will be considered 'State-of-the art' standard equipment 10 to 20 years from now."

There are two approaches for improving the performance of a computer :

1. By improving the performance of a single computer.
2. By Parallel Processing.

Basically, we can achieve parallelism **within the CPU and with many CPUs**. Within the CPU, we have just one CPU. Such an architecture is called as a uniprocessor system. Also, it is known as Von Neumann's architecture (that is, conventional system). In such a system, we can increase the speed by many methods like increasing the clock rate (from MHz to GHz) or by using faster memories and caches, pipelining at instruction level etc. But this approach has already reached its maximum limit. So, we go for the second approach. **In second approach, we use many CPUs (also called as processing elements) to solve a given problem.** The architecture used to solve these problems are known as advanced computer architectures and the algorithms are known as Parallel Algorithms. Programming of these computers is known as Parallel Programming. In fact, advanced computer architectures are centered around the concept of parallel processing.

1.1 NEED FOR PARALLEL COMPUTING

There are several reasons as to why we need parallel computing. Some of the reasons are given below :

1. To refine physical models such as those of complex molecules and crystals, to verify new theories and to solve newer problems of science we need computers which can perform several billions of arithmetic calculations per second. This requires parallel computers (or Supercomputers).
2. To model global weather forecasts, we need large sets of non-linear partial differential equations. Hundreds of mega-flops' speed and large memory size have been found essential to solve such real-time weather forecast equations.
3. To model interaction between atoms and understand micro behaviour of materials, quantum and structural chemists need high speed computing.

4. Molecular biologists require to model very complex genetic structures and need interactive three dimensional graphics facilities to aid their understanding.
5. To create realistic graphic images to display results of simulation of aircraft landing and spacecraft docking, requires very high speed computation.
6. To save time and cost - Time is saved as parallel computing is very fast. Cost is saved as multiple cheap computing resources can also be used.
7. To provide concurrency; i.e., parallel execution of tasks.

Note :

1. **Mega Flops.** A machine which is capable of performing one floating-point operation every 10 n sec has a speed of $\left(\frac{1}{10^{-8}}\right) = 100$ Mega flops i.e., 100 million (mega) floating-point operations per second.
2. **Concurrency.** To do multiple things at the same time.

FP add or subtract operations take less time. The slowest is FP divide operation. Older CPUs taken many clock cycles to complete one FLOP and so. Even at a high clock speeds, their Flop rate can be low.

1.2 CONSTRAINTS OF CONVENTIONAL ARCHITECTURE

1.2.1 Von Neumann's Architecture

As we know that a single computer consists of an input unit, a program memory, ALU and an output unit. The arithmetic and logic unit (ALU) and the control unit (CU) together forms a CPU. A CPU is also named as a processing element or PE or a microprocessor (μ P). In 1940, John Von Neumann proposed a stored-program concept based computer which is also known as a sequential computer or a uniprocessor or Von-Neumann's machine or the conventional computer. It is shown below in Fig 1.1.

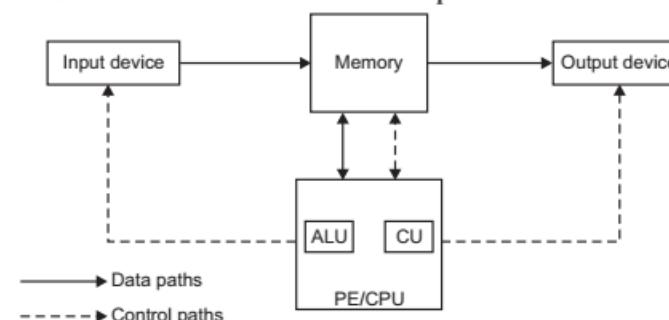


Fig. 1.1. Von Neumann's conventional computer.

Working. The PE retrieves one instruction of the program at a time, interprets it and executes it. The operation of this computer is thus sequential. At a time, PE can execute only one instruction. The speed of this sequential computer is thus limited by the speed at which a PE can retrieve instructions and data from memory and the speed at which it can process retrieved data. To increase the speed of processing of data one may interconnect many such sequential computers to work together. Such a computer which consists of a number of interconnected sequential computers which cooperatively execute a single program to solve a problem is called as a parallel computer.

Let us now tabulate some differences between sequential (or Von Neumann or conventional) computers versus Parallel computer (or Supercomputers). They are as follows.

Table 1.1. Differences between Sequential and Parallel computers.

Sequential or Von Neumann Computers	Parallel or Supercomputers
<ol style="list-style-type: none"> 1. Are uniprocessor systems <i>i.e.</i>, they have only 1 CPU (or PE). 2. They can execute only 1 instruction at a time. 3. Their speed is limited as we know that the speed of light ($c = 3 \times 10^8 \text{ m/s}$) is also limited. 4. It is quite expensive to make single CPU faster. 5. These computers are found in Laboratories of colleges, industries. For eg: Pentium μP 	<ol style="list-style-type: none"> 1. Are multiprocessor systems <i>i.e.</i>, they have many CPU (or PEs). 2. They can execute several instructions at a time. 3. There is no limitation of speeds. We now have supercomputers working at Mega Flops (MFLOPS) and Tera Flops (TFLOPS) range. 4. It is less expensive if we use larger number of fast commodity processors to achieve the better performance. 5. They are not commonly found. Some centres like CDAC in NOIDA have PARAM Supercomputers which has been developed in INDIA. CRAY-1, CRAY-YMP, CRAY-XMP are some other examples of Supercomputers that have been developed by USA.

1.2.2 Limitations of Sequential Machines

1. Only single instructions execution at a time makes these computers slower.
2. The speed of electronic devices used in PE has already reached to a saturation level. The limiting factor being the speed of the light.
3. It is very expensive to make a single processor of a sequential machine faster.
4. Its speed is directly proportional to the speed at which the data can move through hardware.
5. The speed of this computer is also limited by the speed at which PE can retrieve data and instructions from memory.

1.3 COMPUTER GENERATIONS

The state of improvement in the development of a product is known as generation. Different technologies have been used for manufacturing the computer hardware. Various Computer generations are tabulated next :

Generations Architecture	Technology & Applications	Software and Examples	Popular
First (1945-54)	Vacuum tubes & relay memories, CPU driven by PC and Accumulator, fixed-point arithmetic.	Machine/Assembly languages, single user, no subroutine linkage, programmed I/O using CPU.	ENIAC, Princeton IAS, Mark-I, EDVAC, IBM 650/701.

Second (1955-64)	Transistors, core memories, I/O processor, floating-point calculations.	High-level language (HLL) used with compilers, subroutine libraries, batch processing monitor.	ATLAS, B-5000, PDP-1, IBM-7090, CDC-1604.
Third (1965-74)	Integrated circuits (SSI/MSI), microprogramming, pipelining, cache and lookahead processors.	Multiprogramming and time-sharing OS, multiuser applications.	IBM 360/370, CDC 6600, TI-ASC, HP-2100, PDP-8.
Fourth (1975-90)	LSI/VLSI and Semiconductor memory, multiprocessors, vector supercomputers, multi computers.	Multiprocessor OS, languages, compilers and environments for parallel processing.	VAX 9000, Intel 8080, Cray XMP, IBM 3090.
Fifth (1991-Present)	ULSI/VHSIC processors, memory and switches, high density packaging, scalable architectures.	Massively parallel processing, grand challenge applications, heterogeneous processing.	IBM RS - 16000, Fujitsu VPP 500, Intel Paragon.

Please note that here with each new generation, the circuitry becomes smaller and more advanced. The speed, power and memory of computers is proportionately increased due to miniaturization. That is, due to reduction in sizes of circuitry and hardware devices. Each generation is characterized by a major technological development. Also note that the latest generation computers inherited all good features and eliminated bad features in previous generations.

We shall now explain each of these generations in detail.

1. First Generation (1945–1954) : This generation used vacuum tubes and relay memories interconnected by insulated wires. The first generation computers had a single CPU (uniprocessor) that was performing serial fixed point arithmetic using program counter (PC), branch predictions and an accumulator. The CPU was involved in all memory access and Input/output operations. Machine and assembly languages were used.

Examples :

- (a) ENIAC. Electronic Numerical Integrator and Calculator. Built at Moore School of University of Pennsylvania in 1950.
- (b) IBM 70. The first electronic stored program commercial computer built by IBM in 1953.

2. Second Generations (1955–1964) : This generation used the discrete transistors, diodes and magnetic ferrite cores interconnected by printed circuits. Index registers, floating point arithmetic, multiplexed memory and I/O processor were introduced. High level languages (HLL) like FORTRAN, ALGOL and COBOL were introduced with compilers, subroutine libraries and batch processing monitors. In 1957, Irving Reed developed Register Transfer Language (RTL) for the systematic design of digital computers.

Examples :

- (a) IBM 7090 : The stretch computer, built in 1962 : Composed of instruction look ahead and error correcting memories.

(b) **Univac LARC** : Livermore Atomic Research Computer.

3. Third Generations (1965-74) : This generation used integrated circuits (ICs) both in logic and memory in SSI and MSI.

Multilayered printed circuits were used. Microprogrammed control became popular to close up the speed gap between CPU and main memory. Pipelining and cache memory were introduced. Multiprogramming and time sharing were implemented.

Examples :

1. IBM 360/370 Series.
2. CDC 6600/7600 Series.
3. Texas Instruments ASC.
4. PDP-8 Series.

4. Fourth Generation (1975-1990) : It used LSI and VLSI. Core memory was replaced by semiconductors memory. Parallel computers appeared in various architectures. Shared or distributed memory or optional vector hardware were used. Multiprocessing operating system, special languages and compilers were developed.

Examples :

1. VAX 9000
2. Cray XMP
3. IBM-3090

5. Fifth Generation (1991-Present) : This generation is marked by the use of high density and high speed processor and memory chips. VLSI technology is now more improved. For example, 64-bit 150 MHz microprocessors are now available on a single chip with over one million transistors.

These machines emphasize massively parallel processing (MPP). Scalable and latency tolerant architectures are being adopted using VLSI, Silicon, Gas technologies, high density packaging and optical technologies.

The performance of Fifth generation computers will be crossing even Tera Flops (10^{12}) range i.e., 10^{12} floating point operations per second. By using heterogeneous processing and thus a network of heterogeneous computers along with the shared memory can solve large scale problems now.

Examples :

1. MPP — Cray Research
2. CMS — Thinking Machines Corporation
3. Paragon — Intel Super Computer

1.4 THE STATE OF COMPUTING

There are some grand challenge problems in the field of super computers. Some of them are as follows :

- Global weather forecasting.
- Environmental modeling.
- Cosmology.
- Astrophysics.
- Aerodynamics.
- Artificial Intelligence.
- Superconductor modelling.
- Computer vision.
- Quantum mechanics.

And so on. All these applications are computationally intensive. So, we now need very fast computers. These are, naturally, the Supercomputers.

As per the special issue of IEEE on Heterogeneous computing – June 1993, the editor Cherry Pancake described the requirements of **Quantum Chromo-Dynamics**. It is the study of subatomic structure. The experiment involved in this study demands a CRAY Supercomputer to run for 24 hours/day for 1500 years to get the required atomic details.

1.5 EVOLUTION OF PARALLEL PROCESSORS

1.5.1 Parallel Processing Basic Terminologies

1. **Parallel processing.** It is defined as the process of information processing that emphasizes the concurrent manipulation of data elements belonging to one or more processes solving a single problem.
2. **Parallel Computer.** It is defined as a multiple processor computer capable of parallel processing.
3. **Supercomputer.** It is defined as a general purpose computer capable of solving individual problems at extremely high computational speeds, compared with other computers built during the same time.
4. **Through put.** The throughput of a device is the number of results it produces per unit time. There are many ways to improve the throughput of a device. The speed at which the devices operates can be increased or the concurrency can be increased. Please note that by 'Concurrency', we mean the number of operations that are being performed at any one time.
5. **Pipelining.** A pipelined computation is divided into a number of steps called segments/stages. The output of one segment is input of the next segment. This forms a pipe. Please note here that the work rate of the pipeline is equal to the sum of work rates of each of the segment. Also note that, it is possible to achieve pipelining with and within CPU.
6. **Data Parallelism.** It is the use of multiple functional units to apply the same operation simultaneously to the elements of a data set. Please note that in this type of parallelism, the data set is broken up but not the algorithms. For instance, Engineering and scientific programs use this type of parallelism.
7. **Speed Up.** It is defined as the ratio between the time needed for the most efficient sequential algorithm to perform a computation (t_s) and that of the time needed to perform the same computation on a machine incorporating pipelining and /or parallelism (t_p) Mathematically,
$$S(n) = t_s/t_p$$
8. **Degree of Parallelism.** It is defined as the total number of processors required to execute a program in each such time period. An efficient parallel program has a high degree of parallelism.
Parallelism is of two types :
(a) **Hardware Parallelism** — It is built into the machine's architecture.
(b) **Software Parallelism** — It is exploited by the concurrent execution of machine language instructions in a program.

1.5.2 Features of Parallel Processors

There are some interesting features of Parallel Processing/Computers. They are discussed next :

1. **Better quality of solution.** When arithmetic operations are distributed to many computers, each one does a smaller number of arithmetic operations. Thus, the rounding-off errors are lower when parallel computers are used.
2. **Better algorithms.** The availability of many computers which can work simultaneously leads to different algorithms which are *not relevant* for purely sequential computers. It is possible to explore different facets of a solution simultaneously using several processors and these give better insight to solutions of several physical problems.
3. **Better Storage distribution.** Certain types of parallel computing systems provide much larger storage which is distributed. Access to the storage is faster in each computer. This feature is of special interest in many applications such as information retrieval and computer aided design.
4. **Greater reliability.** In principle, a parallel Computer will work even if a processor fails. We can build a parallel computer's hardware and software for better fault tolerance.

1.5.3 Evolution

We are now in a position to review some historical milestones in the computer development to assess the state-of-the-art computing.

Milestones of Computer development are as follows —

There are two major stages in the development of computers :

1. Mechanical/Electromechanical.
2. Electronic.

Till 1945, computers were built with mechanical or electromechanical parts.

Abacus was the earliest mechanical computer used in China in 500 B.C. It is manually operated and used to perform decimal arithmetic with carry propagation digit by digit.

After the Abacus, the following are the major milestones in the history of computing as follows :

1. In 1940, Blaise Pascal built a mechanical adder/subtractor in France.
2. In 1827, Charles Babbage designed a difference engine in England for polynomial evaluation.
3. In 1941, Konrad Zuse built first binary mechanical computer in Germany. This computer was designed for general purpose computations.
4. In 1944, Howard Aiken proposed very first electromechanical decimal computer, for general purpose computations. IBM also built Harvard Mark I.

Please note that the speed and the reliability of these computers was greatly limited due to use of moving mechanical parts. With the introduction of electronics, the moving parts in mechanical computers were replaced by high mobility electrons and computers became electronic computers. Now, electric signals were used for information transmission. The speed of electric signal was almost equal to the speed of light. (Remember: $C=3\times10^8\text{ m/s}$).

1.5.4 Future Trends

During the past 10 years, the trends indicated by ever faster networks, distributed systems and multiprocessor computer architectures suggest that **parallelism is the future of computing**.

From the application point of view, the mainstream usage of computers is experiencing a trend of four ascending levels of sophistication —

- Data Processing.
- Information Processing.
- Knowledge Processing.
- Intelligence Processing.

The relationship between these four levels is demonstrated by a Venn-diagram as shown below in Fig. 1.2.

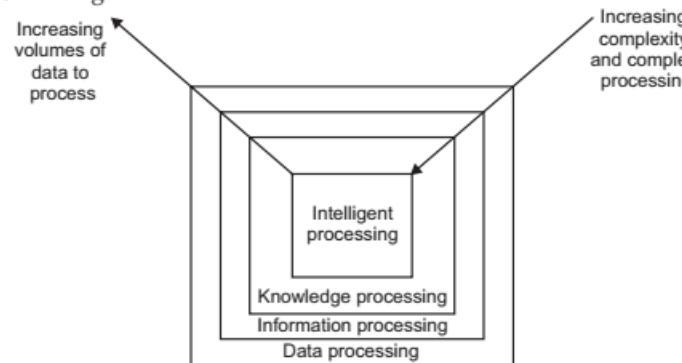


Fig. 1.2. Space diagram for 4-levels of sophistication.

From the Fig. 1.2, it is clear that the space for data processing is largest. Herein, you have increased volume of data to process. At the same time, there is less complexity at this level. However, at the next level *i.e.*, Information processing level, more degree of parallelism has been found which is higher than that at data processing level. Similarly, knowledge processing has more complexity than Information processing. And finally, Intelligent processing has maximum complexity. It is because intelligent processing involves the use of Artificial intelligence (AI) languages. AI is the branch of computer science that deals with nonalgorithmic approach of problem solving. So, these tasks are quite complex. Hottest area of AI is Neural Networks. It is successful in number of disciplines like voice recognition and natural language processing. Quantum computations, molecular and nanotechnology will drastically change the trends of future computing.

1.6 PARALLELISM IN UNIPROCESSOR SYSTEM

1.6.1 Uniprocessor Architecture

The system with a single central processor or PE is called as a Uniprocessor system. A typical uniprocessor computer has 3 main parts :

- (a) The Main Memory (MM),

- (b) The Central Processing Unit (CPU),
- (c) The Input-Output (I/O) Subsystem.

Example System :

1. Mainframe Computer IBM system 370/Model 168 uniprocessor.
2. Super minicomputer VAX-11/780 by DEC.

However, we will study the system architecture of mainframe IBM system 370/ Model 168 uniprocessor. Its architecture is shown below :

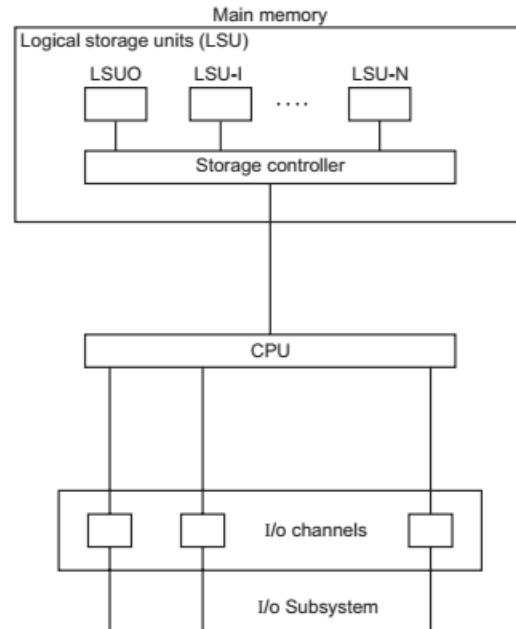


Fig. 1.3. Architecture of a typical uniprocessor

As it is clear from the above Fig. 1.3 that IBM-370 mainframe uniprocessor has three main parts :

1. **Main Memory (MM).** This uniprocessor has four units called as Logical Storage Units (LSUs). So, $N = 4$ for IBM-370 machine. These four LSUs are four way interleaving. We shall be explaining interleaving in subsequent chapters. The Storage controller provides the multiport connections between the CPU and the four LSUs.
2. **CPU.** The CPU contains the instruction decoding and execution units as well as a cache.
3. **I/O Subsystem.** The peripherals are connected to the system via high speed I/O channels which operate asynchronously with the CPU.

1.6.2 Parallel Processing Mechanisms for Uniprocessors

To achieve parallelism in a uniprocessor system, a number of parallel processing mechanisms have been developed. They are as follows :

- (a) Multiple functional units.

- (b) Parallelism and pipelining within the CPU.
- (c) Overlapped CPU and I/O operations.
- (d) Use of hierarchical memory system.
- (e) Balancing of subsystem bandwidths.
- (f) Multiprogramming and time sharing.

Now, we shall explain each of these techniques of introducing parallelism in uniprocessor systems one by one.

1. Multiple functional Units : Von Neumann's (or conventional) computers consist of only one ALU (within CPU). Also, the ALU was capable of performing only one function at a time. So, execution of a long sequence of arithmetic and logic instructions would have been very slow. Therefore, if these functions of ALU are distributed to multiple and specialized functional units working in parallel, then the execution would be faster.

In such a case, the Control Units (CU) become more complex but a higher throughput is achievable. This technique of using multiple functional units in a single processor is called as a 'Superscalar' architecture. By this, we mean that multiple instructions of a single program are simultaneously started and processed, in parallel, in different functional units.

For example : CDC 6600, has multiple functional units like one fixed point adder, two multipliers, one divider, two incrementers, one floating point adder, one shifter, one logical unit, one branch unit. Also, CPU has 24 useful registers i.e., 8 Index Registers (IR), 8 operand address registers and 8 floating point registers.

Note that the CDC-6600 is also known as a **network computer**. This is because it has one CPU which consists of 10 functional units and 10 peripherals and control processor.

The program does not see the functional units. It is the job of CU to take care of instruction's routing to functional units and taking care of conflicts. Up to 10 instructions can be issued in the CPU, as long as there are no conflicts.

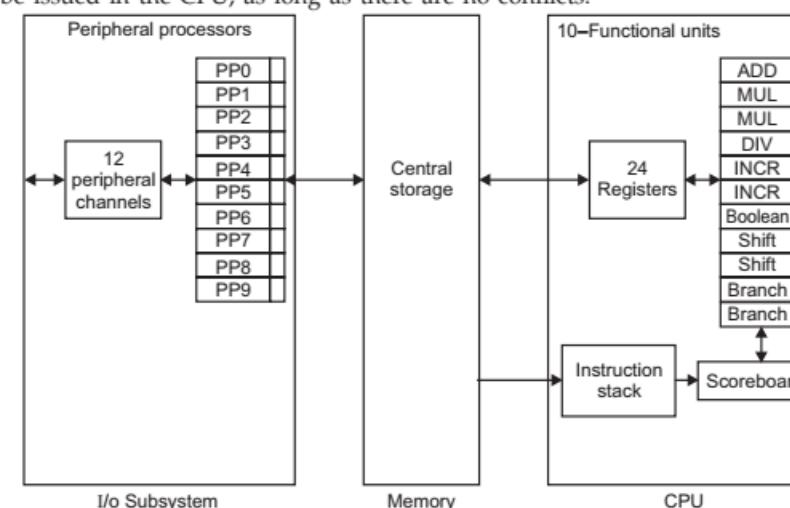


Fig 1.4. CDC-6600 uniprocessor with 10-functional units.

CDC-6600 was designed in year 1964. It's 10 functional units can operate in parallel.

There is a **Scoreboard** used to check the availability of functional units and registers required.

The central processor consists up of an **instruction stack** that can store eight words of 60 bits. The instructions are 16-bit and 30-bit long.

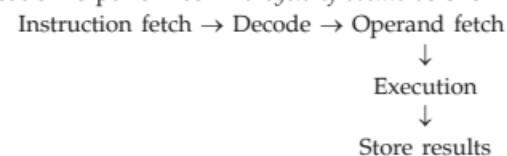
How concurrency is achieved in CDC-6600 uniprocessor?

- CDC-6600 has an instruction prefetch facility. So, 8 words of 60 bits each can be prefetched.
- CDC-6600 has 10 functional units with a Scoreboard that provides a queue and a reservation scheme.
- 32-way memory interleaving.
- 10 peripheral processors with own memory for I/O programs.

2. Parallelism and Pipelining within the CPU : We must understand that the use of multiple functional units is a form of parallelism within the CPU. In first generation, old machines we used to have bit serial address. But now, almost all CPUs make use of parallel adders - like carry-look ahead adder and carry-save adders.

High speed **multiplier recoding** and **convergence division** are the techniques for exploring parallelism and the sharing of hardware resources for the functions of multiply and divide.

Instruction execution is performed in a *cycle of events* as shown below:



All these phases are pipelined. Instructions are prefetched and data buffering techniques have been developed in order to overlap instruction execution through pipe.

3. Overlapped CPU and I/O operations : Separate I/O controllers or I/O processors or channels are used so that I/O operations are performed simultaneously while CPU is doing calculations. The Direct - Memory - Access (DMA) channel can be used to provide direct information transfer between the I/O devices and the main memory (MM). DMA works on a cycle stealing basis which is apparent to the CPU.

For Example 1. In CDC-6600, I/O multiprocessing has been achieved by the use of 10 I/O processors that can increase the speed of data transfers between the CPU and the outside world.

Example 2. Back-end database machine can also be used to manage large databases stored on disks.

4. Use of Hierarchical Memory System : Actually there is a very large speed gap between CPU and memory. Note that the memory is about 1000 times slower than CPU. To close up this speed gap, a hierarchical memory system is used. Computer memory hierarchy is conceptually shown in Fig 1.5 below.

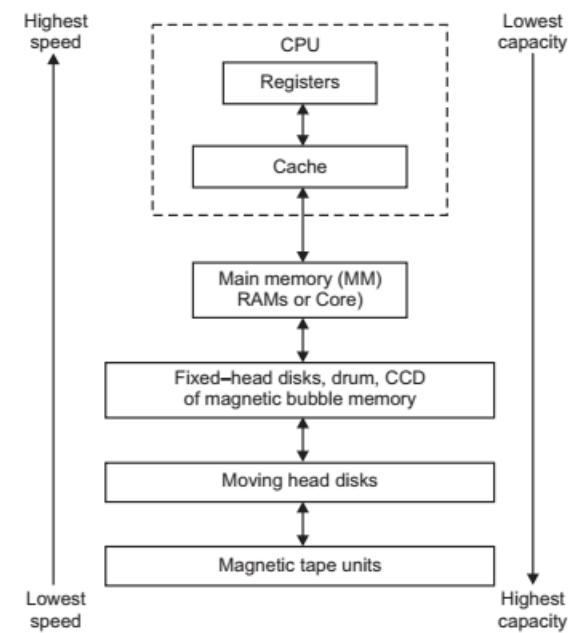


Fig.1.5. Classical memory hierarchy

As shown in Fig. 1.5, Register files is the innermost level of memory. It is directly addressable by the ALU. Next layer is cache memory. It acts as a buffer between CPU and the main memory. Main memory can be accessed in blocks by using multiway interleaving across parallel memory modules. At the outer levels of hierarchy, disks and tape units are used to establish a virtual memory space.

Please note that the hierarchy illustrated above is used to broaden the memory bandwidth so that it matches the bandwidth of CPU.

5. Balancing of Subsystem Bandwidths : As discussed earlier also, that as per Von-Neumann's architecture, a computer has three parts, namely :

- (a) CPU
- (b) Main Memory (MM) and
- (c) I/O devices.

Also, we know that :

$$\boxed{\text{ALU} + \text{CU} = \text{CPU (or PE)}}$$

However, out of these three subsystems, **CPU is the fastest unit and I/O devices are the slowest devices**. The main memory is between the two devices, i.e., CPU and I/O. The performance of these subsystems is measured in terms of their bandwidths. We shall now define the term 'bandwidth'. It is defined below :

"Bandwidth is defined as the number of operations performed per unit time."

or

"Bandwidth is defined as the maximum rate in millions of bits per second (MBPS) at which information can be transferred to or from a memory unit".

Bandwidths may be related to the main memories, processors, I/O or even the utilization Bandwidths. We shall now define these one by one.

- (a) **Main Memory bandwidth (B_m)** : It is defined as the number of memory words that can be accessed (either fetch or store) per unit time. To avoid bottleneck between the subsystems, their processing bandwidths should be matched.

Let W be number of words delivered per memory cycle and t_m be memory cycle time in seconds, then memory bandwidth B_m is equal to.

$$B_m = \frac{W}{t_m} \text{ (words/second or bytes/second).}$$

- (b) **Bandwidth of processor (B_p)** : It is the maximum CPU computation rate. e.g.

Cray - 1 processor bandwidth (B_p) = 160 megaflops (MFLOPS)

IBM 370/168 processor bandwidth = 12.5 (million instructions per second) MIPS
 B_p – Denotes processor bandwidth.

- (c) **I/O Bandwidth (B_d)** : It is defined as average data transfer rate. e.g. Modern disc drive has $B_d = 1$ magabyte/second.

- (d) **Utilization Bandwidths** : In practice, utilization bandwidths of these subsystems are lower than actual bandwidths.

1. Utilization bandwidth of memory B_m^u .

Rough measure of B_m^u is defined as follows.

$$B_m^u = \frac{B_m}{\sqrt{M}}$$

M is the number of interleaved memory modules in the memory system.

e.g. for IBM 3033 uniprocessor; $M = 8$

$$B_m^u = \frac{B_m}{\sqrt{8}}$$

It can read eight double words (8 byte each) from an eight way interleaved memory system per each memory cycle $t_m = 456$ ns.

$$\therefore B_m = \frac{8 * 8 \text{ bytes}}{456 \text{ ns}}$$

$$= 140 \text{ megabytes/sec.}$$

$$\therefore B_m^u = \frac{140}{\sqrt{8}}$$

$$= 49.5 \text{ megabytes/sec.}$$

- (e) **Utilization Bandwidth of CPU (B_p^u)** : It is decided by measuring number of output results (in words) per second.

Let R_w be number of words results and T_p be the total CPU time required to generate R_w results, then

$$B_p^u = \frac{R_w}{T_p} \text{ (words/sec.)}$$

- (f) **Utilization Bandwidth of I/O devices (B_d^u)** : It is lower than their actual Bandwidth B_d . In practice, relationship between the bandwidth's of major subsystems in a high performance uniprocessor is observed as follows :

$$B_m \geq B_m^u \geq B_p \geq B_p^u > B_d$$

Hence it is clear that main memory has highest bandwidth.

Also if t_p is processor cycle time, t_m is memory cycle time and t_d is average access time of I/O devices then relation between t_p , t_m and t_d is observed as follows :

$$t_d > t_m > t_p$$

The above relationship implies that there is unbalance in the speeds of major subsystems and the processing power of these subsystems is required to be matched to obtain better performance.

The two major techniques used for balancing subsystem bandwidths are as follows :

- Bandwidth Balancing between CPU and Memory** : The difference in the speed of CPU and the main memory can be covered up by using fast cache memory between them as shown in Fig.1.6.



Fig 1.6. Cache Memory for Balancing Bandwidth of CPU and Memory

Access time of cache (t_c) should be equal to speed of CPU i.e. $t_c = t_p$.

The part of memory words are first loaded into cache memory. Hence immediate instructions and data will be available from the cache to the CPU. Cache acts as a data/instruction buffer. However cache memories are costly and hence it is not economical to use them in bulk.

2. Bandwidth balancing between memory and I/O (Input-Output) devices : To balance the speed of I/O devices and memory, input-output channels with different speeds are used between the slow I/O devices and the main memory. The function of these I/O channels is to act as a buffer and also provide multiplexing transfer the data from multiple I/O devices into main memory by stealing CPU cycles. In addition to this, the filtering of the irrelevant data before transferring to main memory can provide a faster and more effective data transfer rate.

Ideally for a totally balanced system, entire memory bandwidth should be watched with the bandwidth sum of the processor and I/O devices.

$$\text{i.e., } B_p^u + B_d^u = B_m^u$$

$$\text{where } B_p^u = B_p$$

$$\text{and } B_m^u = B_m$$

Tremendous hardware and software support is required to achieve this total balance.

1.6.3 Multiprogramming and Time Sharing

In addition to above hardware approaches for increasing speed of uniprocessor system, following software approaches are also used to achieve concurrency.

1. Multiprogramming and
2. Time sharing

1. Multiprogramming : It is possible to achieve a high degree of resource sharing among many user programs even in presence of single CPU.

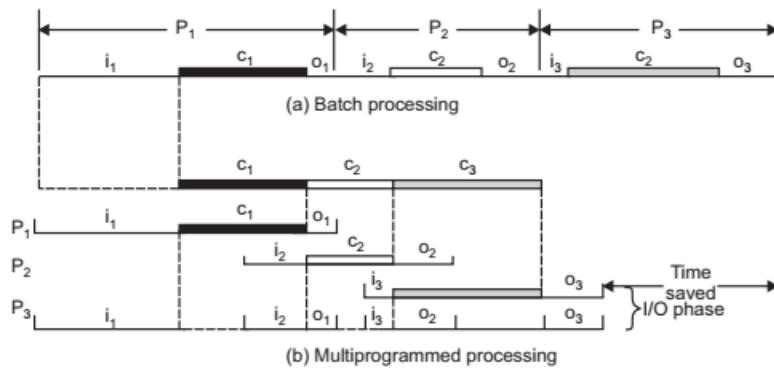


Fig. 1.7.

In Fig 1.7 (a) conventional batch processing is illustrated by sequential execution. Following notations are used for operations.

- i → input
- C → compute
- O → output

There may be more than one process active in a computer in the same time interval. Some of these processes might be competing for memory, some will be I/O processors and others might demanding CPU. Some computer programs are CPU - bound (Computation intensive) and some are I/O bound (input output intensive). Execution of various types of such programs can be mixed to balance the bandwidths among various functional units. There will be better resource utilization by means of program interleaving through overlapping I/O and CPU operations. **This interleaving of CPU and I/O operations among several programs is called multiprogramming.** Refer Fig. 1.7 (b).

Whenever I/O operations in processors P_1 is going on, system scheduler can switch CPU to process P_2 . So P_1 and P_2 (more than one) processes will be executing simultaneously in the system. When P_2 is over, the CPU can be switched to P_3 . It can be noted that there is overlapped CPU and I/O operation and CPU wait time is greatly reduced. The programs can be mixed across the boundary of users tasks and systems processes. Multiprogramming reduce the total execution time. The process P_1 , P_2 and so on, may belong to the same or different programs.

2. Time sharing : In multiprogramming, CPU is shared by many program. Sometimes, certain program occupies CPU for very long time and hence other programs has to wait. This problem can be solved by using a time sharing operating system. **The basic concept is to allot fixed or variable time slices to multiple programs.** Hence time sharing O.S. gives equal opportunities to all programs demanding CPU time.

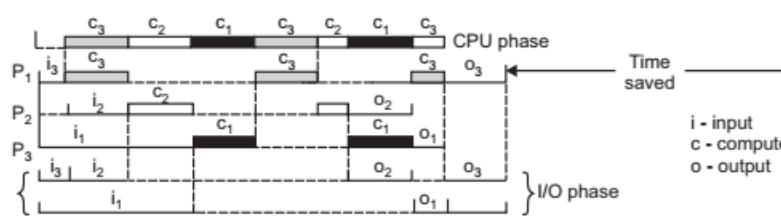


Fig. 1.8. Time Shared Processing

Time sharing may reduce execution time more than batch processing multiprogramming approaches. The concept of time sharing was first developed for a uniprocessor system and then extended to multiprocessor systems. The performance of both system depends heavily on the capability of operating system.

1.7 MULTIPROCESSORS AND MULTICOMPUTERS

There are **two main categories** of the parallel computers. Before categorizing them let us see their definitions. A **multiprocessor** is a **single computer** in which **multiple processors** exist. The complete system comprises of many (multiple) processors. The processors are interconnected through an **Interconnection Network**. They share **common resources** like memory and I/O devices. They have *just one O.S.* Even though many computer exist, it behaves as a **single computer only**. Whereas in a **multicomputer system**, **multiple autonomous computers** are interconnected by the network. Also, the communication among them is *not mandatory*. So, we can say that the **distributed memory MIMD** (Multiple Instruction Multiple Data) architectures are called as multicomputers. While **Shared memory MIMD** architectures are called as multiprocessors.

We are now in a position to tabulate the differences between the two. They are given below :

Multiprocessors	Multicomputer
<ol style="list-style-type: none"> It is a single computer in which multiple processors exist. Although many PEs are used but these PEs do not have their own individual memories. Infact, sharing of memory and I/O resources is done. So, they are also known as shared memory multiprocessors. The communication among PEs is must. They use dynamic networks wherein the communication links can be reconfigured by setting the active switching unit of the system. They are also called as tightly-coupled systems due to the high degree of resource sharing. <i>E.g.</i> The Sequent symmetry S-81 	<ol style="list-style-type: none"> It has multiple (many) autonomous computers. Here also many PEs (or CPUs) exist but each PE has its own memory and I/o resources. No sharing occurs in this case. So, they are also known as Distributed memory multicomputers. The communication among PEs is not mandatory. They use static network wherein the connection of switching units is fixed and is typically, realized as direct or point to-point connections. These networks are also known as direct networks. They are also called as loosely-coupled as there is no resource sharing. <i>E.g.</i> Message Passing Multicomputer.

Next we shall describe the two main categories of parallel computers.

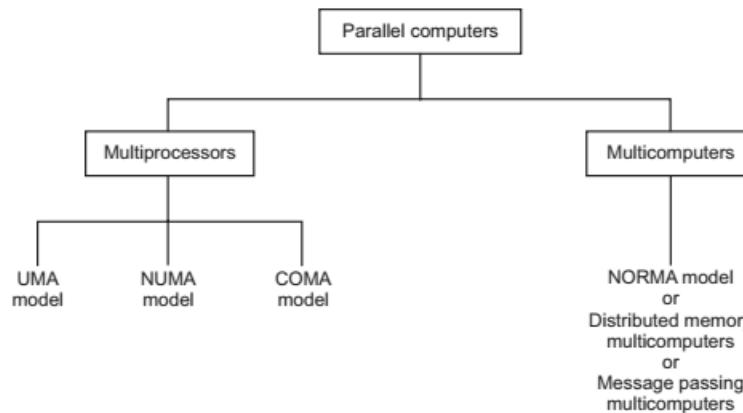


Fig.1.9. Categories of Parallel computers—multiprocessors and multicomputers.

We shall now describe each of these models one by one.

I. The UMA Model

As shown in Fig. 1.10. in this model, the physical memory is uniformly shared by all the processors. All processors ($PE_1 \dots PE_n$) have equal access time to all memory words. This is the reason as to why they are called as uniform memory access computers. Please note that in this architecture, each processor (CPU/PE) may have its own private cache memory. The peripherals (I/O) devices are also shared. Since there is a high degree of resource (memory and I/O) sharing, so these systems are also known as tightly coupled systems. At the centre of Fig 1.10, we can also see a system Interconnection Network which can be a common bus architecture, a crossbar switch or a multistage network (These will be discussed in chapter 3.)

When all processors (or PEs) have equal access to all peripheral devices then the system is called as a **symmetric multiprocessor**. So, now all the processors are equally capable of running the executive programs like OS Kernel or I/O service routines. Whereas in an **asymmetric multiprocessor**, only one or a subset of processors are executive-capable. It is the job of master processor (or an executive) to execute the O.S. and the handle I/O. The remaining processors have no I/O capability. So, they are called as **attached processors**. These attached processors execute user programs under the supervision of the master processor. However, both symmetric and asymmetric multiprocessors do resource sharing.

We shall try to solve one problem now.

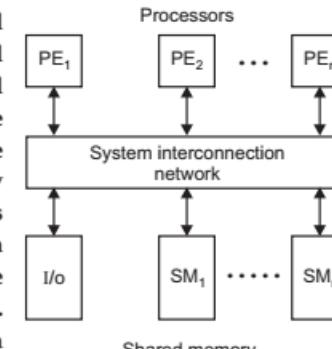


Fig.1.10. UMA Model

Example 1. Consider the following C-program.

```
for (i = 0; i < n; i++) {
    a[i] = b[i] + c[i];
}
sum = 0;
for (j = 0; j < n; j++) {
    sum = sum + a[j];
}
```

This program is to be executed firstly on a uniprocessor or sequential (or conventional or Von Neumann) machine. How many machine cycles are required. Now, if this program is run on a M-processor system, how many cycles are required? Compare your results. Assume that :

$N = 2^{20}$ elements in the array

Average no. of cycles required = 200 cycles

$M = 256$ (PEs | CPU | Processors)

How much is the efficiency?

Solution 1. Let us first of all number the lines of the code in given program :

for (i = 0; i < n; i++) {	L1
a[i] = b[i] + c[i];	L2
}	L3
Sum = 0;	L4
for (i = 0; j < n; j++) {	L5
Sum = Sum + a[j];	L6
}	L7

Suppose now that line no. 2, 4 and 6 take 1 machine cycle to execute. For simplicity, let us ignore line no. 1, 3, 5 and 7. Further assume that k-cycles are needed for each interprocessor communication using a shared memory. We also assume that the arrays are already loaded in the main memory. We ignore bus contentions also.

Now, we have 2 cases :

Case 1 : If the program is executed on the conventional or sequential machine then we find that the i - for loop requires N cycles and also j -loop requires N cycles (Note : j loop contains N -recursive iterations). So, sequential execution of above program requires $2N$ cycles (totally).

Case 2 : If the program is executed on a multiprocessor with M-processors then the total number of cycles required are :

$$2 N/M + (K + 1) \log_2 M \text{ cycles} = 2 N/M + (K+1)l$$

where $l = \log_2 M$ and $(K + 1)l$ is adder time.

Why ? This is because addition of each pair of partial sums requires K -cycles through shared memory. To merge all partial sums, an l -level binary adder is required, where $l = \log_2 M$. This adder tree takes $l(K + 1)$ cycles to merge the M partial sums sequentially from the leaves to the root of the tree. Hence, the result.

Now, we are given that :

$$N = 2^{20}, M = 256 = 2^8, K = 200$$

\therefore Sequential execution takes = $2N$ (cycles)

$$\begin{aligned}
 &= 2^1 \times 2^{20} \text{ cycles} \\
 &= 2^{21} \text{ machine cycles} \quad (\text{Ans})
 \end{aligned}$$

$$\begin{aligned}
 \text{Parallel execution takes} &= \frac{2N}{M} + (k+1) \log_2 M \text{ (cycles)} \\
 &= 2 \times \frac{2^{20}}{256} + (200+1) \log_2 256 \\
 &= 2^1 \times \frac{2^{20}}{2^8} + 201 \log_2 256 \\
 &= 2^{21-8} + 201 \log_2 2^8 \\
 &= 2^{13} + 201 \times 8 \log_2 2 \\
 &= 2^{13} + 1608 \times 1 \\
 &= 2^{13} + 1608 \\
 &= 9800 \text{ cycles} \quad (\text{Ans}) \\
 \therefore \text{Speed-up} &= \frac{2^{21}}{9800} \approx 256
 \end{aligned}$$

Whereas actual speed-up obtained is 214 only. So,

$$\begin{aligned}
 \text{Efficiency } (\eta) &= \frac{214}{256} \times 100\% \\
 &= 83.6\%
 \end{aligned}$$

\therefore Efficiency of 83.6% is achieved by parallel processing.

Applications of UMA Model :

1. It is suitable for general purpose and time-sharing applications by multiple users.
2. It can be used to speed up the execution of a single large program in time-critical applications.

II. The NUMA Model

It is a sort of shared memory multiprocessor only. In these computers, the access time varies with the location of the memory word. There are two possibilities of NUMA model. One is to have the shared memory that is physically distributed to all PEs. This memory is called as local memory (LM). This is shown in Fig 1.11 below :

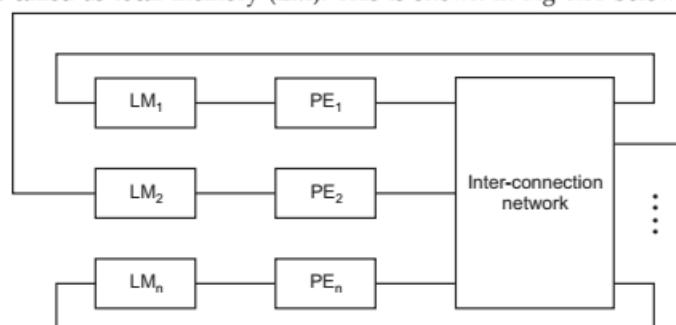


Fig 1.11. Shared Memory NUMA Model.

Please note that here the collection of all local memories forms a global address space (GAS) that is accessible by all processors. The BBN Butterfly system is its example.

Another model is known as a **hierarchical cluster model**. In this model, instead of distributed memories, a global shared memory is added to a multiprocessor system.

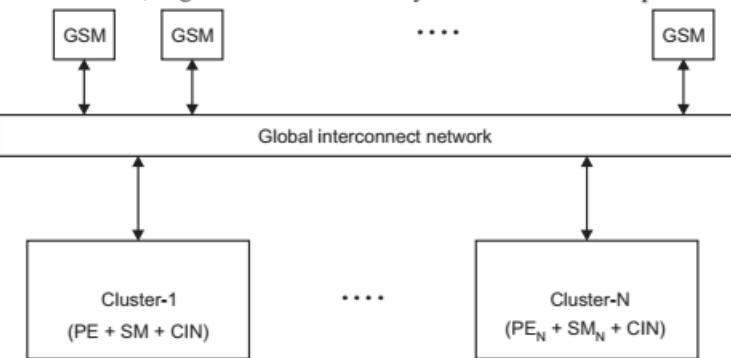


Fig 1.12. Globally shared memory multiprocessor.

A **Cluster** comprises of many PEs, shared memory of each PE and a cluster interconnection network. So, we have three types of memory accesses. They are as follows:

- (a) Local memory access (within cluster)
- (b) Global memory access (GSM)
- (c) Remote memory access.

Please note that the local memory access is fastest. Then comes the global memory access. Then, finally, we have the slowest one, that is, the remote memory access. Also note that, each cluster is itself an UMA or NUMA multiprocessor. The clusters are connected to GSM. This entire system is collectively known as a NUMA multiprocessor. There are two important points to be kept in mind. They are :

- (a) All processors belonging to the same cluster are allowed to *uniformly* access the cluster shared-memory.
- (b) All clusters have *equal access* to global memory.

Note : Speed of local memory (LM) > Speed of cluster Shared memory > Speed of GSM

Example System : The Cedar Multiprocessor by University of Illinois.

III. The COMA Model

We can easily understand this model if we understand this equation :

$\text{Multiprocessor} + \text{Cache memory} = \text{COMA Model}$

Examples of COMA machines :

- Data Diffusion Machine (DDM) by Swedish Institute of Computer Science.
- KSR-1 machine by Kendall Square Research.

A COMA model is shown below in Fig. 1.13.

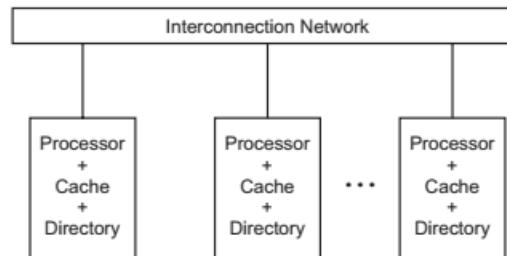


Fig 1.13. COMA Model

So, now distributed memories have been converted to caches. There is no memory hierarchy now. All these cache form a global address space. If cache access is remote (far-off) then distributed cache Directories help.

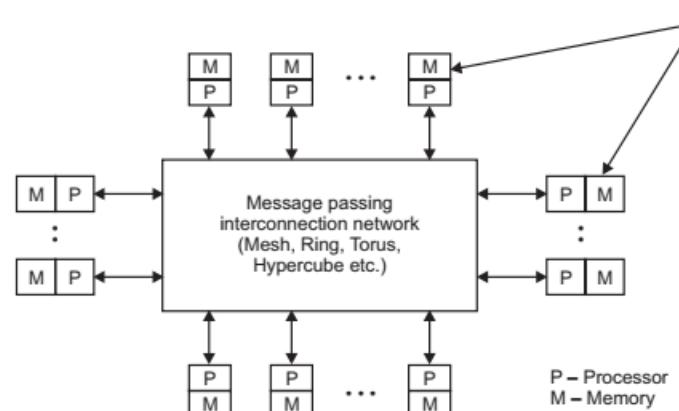
Other variants of COMA also exist. For instance, cache coherent non-uniform memory access (CC-NUMA) model. For eg., Stanford Dash is a CC-NUMA model. Also note that machines like SS-581 is known as a **minisupercomputer**. IBM-390 model are also known as **high-end mainframes or near-supercomputers**.

Applications. General purpose multiuser applications need multiprocessors.

We shall describe multicomputers now (Category-II).

II NORMA machines /Message passing multicomputer/Distributed-memory multicomputers

These machines consist up of multiple computer. These multiple computers are known as **nodes**. A node is an autonomous computer consisting of a processor, local memory, attached disks or I/O Peripherals. These nodes are also interconnected by a *message passing network*. This network of CEs may be in Mesh, Ring, Torus, Hypercube etc. forms. Such a computer is shown below in Fig. 1.14.



The interconnection network provides point-to-point static connections among the nodes. All local memories are private and are accessible only by local processors. That is why these multicomputers are also known as no-remote-memory-access i.e., NORMA machines. Please note that internode communications is carried out by passing messages through static connection network. This network can be a router also. Each node is attached to a router. Certain message-passing protocols are used here.

These multicomputers have also gone through two generations of development. They are given below in the table :

Generation	Technology and Architecture	Example Systems
First (1983–87)	<ul style="list-style-type: none"> • Processor board technology • Hypercube architecture 	<ul style="list-style-type: none"> • Intel iPSC/1 • Caltech cosmic
Second (1988–1992)	<ul style="list-style-type: none"> • Hardware message routing • Mesh architecture 	<ul style="list-style-type: none"> • Paragon by Intel. • Parsys Super Node 1000
Third (1993–1997)	<ul style="list-style-type: none"> • VLSI • Fine-grain multicomputers 	<ul style="list-style-type: none"> • MIT J-machine • Caltech Mosaic

Table—Generations of Multicomputers.

1.8 MULTIVECTOR AND SIMD COMPUTERS

Now, we discuss about vector supercomputers first.

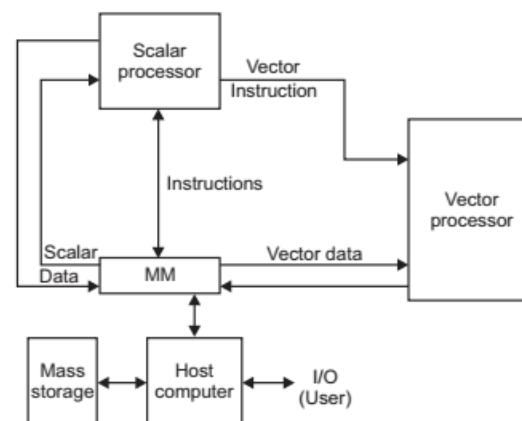


Fig 1.15. A vector supercomputer

It is a register-to-register architecture. The scalar processor has scalar functional pipelines while the vector processor has vector functional pipelines. The vector processor has vector registers which are programmable. The **working** is very complex. If the decoded instruction is a **scalar operation** then it is directly executed by scalar processor using scalar functional pipelines. Else, if the instruction is decoded as a **vector operation**, it is sent to vector control unit. It supervises flow of vector data between MM and vector functional pipes. As shown in Fig. 1.15, the program and data from the host computer are loaded into the Main Memory (MM). Both are decoded by the scalar unit first.

Another **variant** of vector supercomputers is based on memory-to-memory architecture. It uses a vector stream unit to replace vector registers. Retrieval of operands (vector) and results is done directly from the main memory in superwords (like 512 bits).

Example system : Cyber-205, Cray-YMP. Let us tabulate the differences between the two architectures now.

Memory-to-Memory	Register-to-Register
1. Source operands, intermediate and final results are retrieved directly from main memory.	1. Here, operands and results are retrieved indirectly through a large number of vector registers.
2. They have slow speeds.	2. They are faster systems.
3. Architectural design is simple .	3. Architectural design is complex .
4. Spatial parallelism is possible with multiple PEs.	4. Temporal parallelism restricts size of vector to be processed.
5. Parallel algorithms are easy to solve.	5. Parallel algorithms are difficult to solve.

Note : Number of vector registers and functional pipelines in a vector processor are fixed.

Next, we discuss about SIMD computers in short. They will be discussed in detail in subsequent chapters.

SIMD Model. A single Instruction Multiple Data (SIMD) can be specified by 5-parameters :

$$\text{SIMD (M)} = \{\text{N}, \text{C}, \text{I}, \text{M}, \text{R}\}$$

Where

'N' - represents the number of PEs in the machine.

'C' - represents a set of instructions.

'I' - represents a set of instructions that are broadcast by the CU to all PEs.

'M' - represents a set of masking schemes for masking or demasking certain PEs.

'R' - represents a set of data-routing functions.

For example : Master Computer (MP-1 family) which has processors (or PEs) ranging from 1024 to 16,384 PEs with 1.3 GFlops rate. Each of this PE has 16 kB local memory. It supports Fortran-77, UNIX OS, debuggers and so on.

We shall be describing SIMD computers in detail again in subsequent chapters.

1.9 PARALLEL ARCHITECTURAL CLASSIFICATION SCHEMES

There are three computer architectural classifications based on the **multiplicity of instruction** and **data stream**. They are given below :

1. Flynn's classification (1966)
2. Feng's classification (1972)
3. Handler's classification (1977)

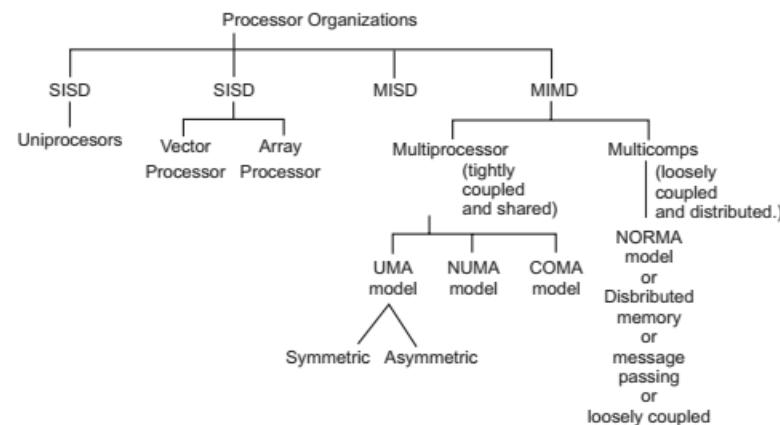


Fig 1.16. Processor types.

We shall describe each of these one by one now.

1.9.1 Flynn's Classification

Michael J Flynn introduced a scheme for classifying computer organizations. He uses the concept of data stream and instructions. A **data stream** is defined as a sequence of data including input, partial or temporary results which are called for by the instruction stream. An instruction stream is a sequence of instructions as executed by the machine.

Flynn's four machine organizations are given and discussed below :

1. SISD Computer (Single Instruction, Single Data)

These computers execute single instruction and single data at a time. They are shown in Fig. 1.17. below :



Fig 1.17. SISD Computer

For example : Von Neumann's sequential computer (or uniprocessors) fall under this category, IBM 701, VAX11, CDC6600, mainframes.

2. SIMD Computer (Single Instruction, Multiple Data)

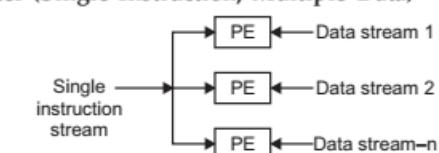


Fig 1.18. SIMD computer

So, in these computers, a single instruction is broadcasted which operates on many data streams. Hence, similar type of operation is performed on all data streams.

For example : Array Processors, Illiac-IV, MPP, CM-2, Maspar MP-1, IBM 9000, Cray C90.

3. MIMD Computer (Multiple Instruction, Multiple Data)

In these computers, individual PEs operate on programs having their own instructions and data. So, we have many (multiple) instructions working or operating on many data streams.

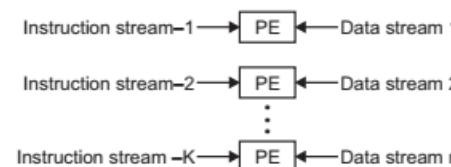


Fig 1.19. MIMD Computer

For example : Multiprocessors are put under this category, IBM 370, Cray-2, UNIVAC 1100/80, ENCORE, MULTIMAX, INTER iPSC, NCUBE/7 etc.

4. MISD Computer (Multiple Instruction and Single Data)

In this category, multiple instructions operate on a single data stream. That is,

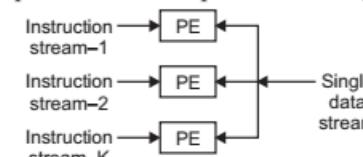


Fig 1.20. MISD Computers

For example : Systolic array

1.9.2 Feng's Classification

Tse-Yun Feng has suggested the use of the degree of Parallelism to classify various computer architectures. First of all we must define what is meant by **maximum degree of Parallelism**. It is defined as the **maximum number of bits that can be processed within a unit time by a computer system**. It is represented by P .

Let P_i be the number of bits that can be processed within the i^{th} processor cycle (or i^{th} clock period). Consider T processor cycles indexed by $i=1, 2, \dots, T$. Then, the average parallelism degree p_a is defined by :

$$P_a = \frac{\sum_{i=1}^T P_i}{T}$$

Generally, $P_i < P$.

∴ We define the utilization rate (μ) of a computer system within T cycles by :

$$\mu = \frac{P_a}{P} = \frac{\sum_{i=1}^T P_i}{T.P}$$

If the computing power of the processor is fully utilized i.e., the parallelism is fully exploited, then we have $P_i = P$ for all i and $\mu = 1$ for 100% utilization. The utilization rate depends on the applications program being executed.

Thus, computer can be classified based on their maximum parallelism degree. Fig. 1.20 demonstrates this classification.

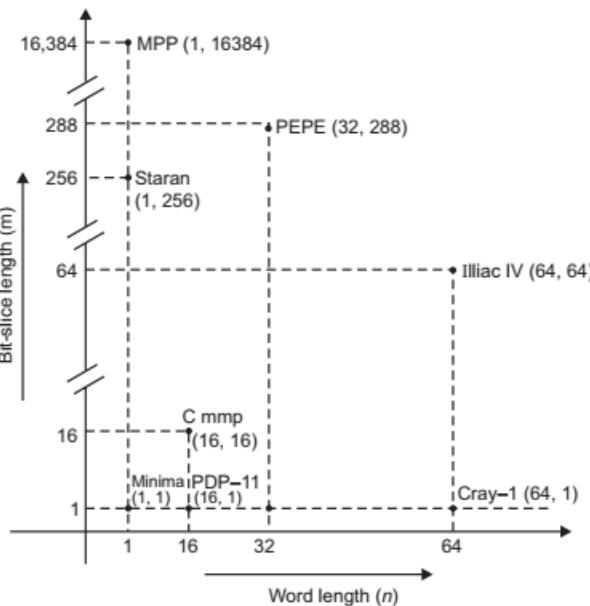


Fig 1.20. Feng's classification

As shown in Fig. 1.20, the horizontal axis shows the word length, n . The vertical axis corresponds to the bit slice length, m . Please note here that both length measures are in terms of the number of bits contained in a word or in a bit slice. A bit slice is a string of bits, one from each of the words at the same vertical bit position.

Now, the maximum parallelism degree, $P(c)$ of a given computer system C is represented by the product of the word length, w , and the bit slice length, m , that is

$$P(c) = n.m$$

Here, the pair (n, m) corresponds to a point in a computer space corresponding to a particular computer, shown by the coordinate system in the graph of Fig. 1.20. Please note that $P(c)$ is equal to the area of the rectangle defined by the integers n and m .

There are *four types* of processing methods that can be seen from Fig. 1.20 :

1. Word-serial and bit-serial (WSBS).
2. Word-parallel and bit-serial (WPBS).
3. Word-serial and bit-parallel (WSBP).
4. Word-Parallel and bit-parallel (WPBP).

Fig 1.21 below shows the word slice and bit slice.

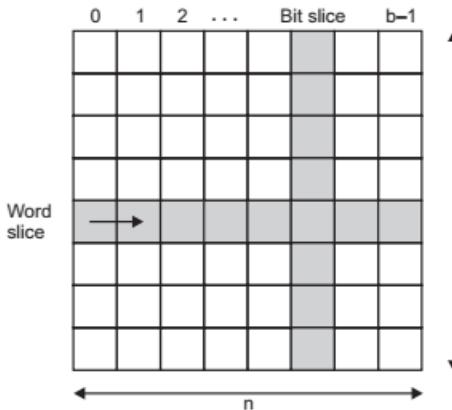


Fig 1.21. Word slice and bit slice.

Let us explain these four types of processing methods with respect to Fig. 1.21.

Method-I : WSBS processing method (where $n = m = 1$):

It is also known as **bit serial processing** because one bit is processed at a time. This method was only used in first generation computers.

For example : The "MINIMA".

Method-II : WPBS processing method (where $n = 1, m > 1$) :

It is also known as **bit-slice or bis-processing**. In this method, an m -bit-slice is processed at a time.

For example : STARAN and MPP (by Goodyear Aerospace).

Method-III : WSBP processing method (where $n > 1$ and $m = 1$) :

It is also known as **word-slice processing** as it processes one word on n -bits at a time. This is found in most existing computers.

For example : IBM-370, CDC 6600

Method IV : WPBP processing method (where $n > 1$ and $m > 1$) ;

This is known as **fully parallel processing** or simply **parallel processing**. Here an array of $(n \times m)$ bits is processed simultaneously.

Please note that this is the fastest processing mode out of the four models.

For example : ILLIAC IV and PEPE .

We summarize Feng's classification in a tabular form now. It is given below :

Mode	Computer model (manufacturer)	Degree of Parallelism (n,m)
1. WSBS ($n=m=1$)	• The "MINIMA" (unknown)	(1, 1)
2. WPBS ($n=1, m>1$) (bit-slice processing)	• STARAN (Goodyear Aerospace) • MPP (Goodyear Aerospace) • DAP (ICL, England)	(1, 256) (1, 16, 384) (1, 4096)
3. WSBP ($n>1, m=1$)	• IBM 370 • CDC 6600	(64, 1) (60, 1)

(word-slice processing)	<ul style="list-style-type: none"> • Burrough 7700 • VAX 11 (DEC) 	(48, 1) (16, 1)
4. WPBP (n>1, m>1) (fully parallel processing)	<ul style="list-style-type: none"> • Illiac-IV (Burroughs) • TI-ASC(TI) • C.mmp(CMU) • S-I (LCNL) 	(64, 64) (64, 32) (16, 16) (36, 16)

1.9.3 Handler's Classification

Handler's classification is based on the **degree of parallelism** versus **degree of pipelining** that is built into the *hardware structures* of a computer system. Here, the **parallel-pipeline processing** is considered at three subsystem levels :

1. Processor Control Unit (PCU) *i.e.*, CPU.
2. Arithmetic Logic Unit (ALU) *i.e.*, PE.
3. Bit Level Circuit (BLC) *i.e.*, combinational logic circuitry needed to perform 1-bit operations in the ALU.

A computer system, X, can be characterized by a triplet containing six independent entities as follows :

$$T(X) = \langle K \times K', D \times D', W \times W' \rangle$$

Where

K = the number of processors (PCUs) within the Computer.

D = the number of ALUs or PEs under the control of one PCU.

W = the word length of an ALU or of a PE.

W' = the number of pipeline stages in all ALUs or in a PE

D' = the number of ALUs that can be pipelined.

K' = the number of PCUs that can be pipelined.

For example :

Texas Instrument's Advanced Scientific Computer (TI- ASC) has following specifications:

$$\begin{aligned} T(ASC) &= \langle 1 \times 1, 4 \times 1, 64 \times 8 \rangle \\ &= \langle 1, 4, 64 \times 8 \rangle \end{aligned}$$

It has one controller controlling four arithmetic pipelines and each has 64-bit word length and eight stages.

Please note that whenever the second entity *i.e.*, K', D' or W' equals to 1, we drop it, since pipelining of one stage or of one unit is meaningless.

Also note that, in many computers, the number of stages in different functional units are variable. In such cases, the range of pipeline stages is indicated within parenthesis. For example, Cray-1 Supercomputer contains a variable number of pipeline stages in all ALUs. So, a variable range of 1 to 14 is used. That is,

$$T(Cray-1) = \langle 1, 12 \times 8, 64 \times (1 \sim 14) \rangle$$

For cray-1, the pipeline chaining degree is a variable with a maximum value equal to 8.

Let us consider another example. CDC 6600, has a CPU with an ALU that has 10 specialized hardware functions, each of word length of 60-bits. Furthermore, CDC-6600 has 10 peripheral I/O processor which can operate in parallel. Each I/O processor has one ALU with a word length of 12-bits. So, we say that :

$$\begin{aligned} T(CDC) &= T(\text{CPU}) * T(\text{I/O Processor}) \\ &= <1, 1 * 10, 60> * <10, 1, 12> \end{aligned}$$

Yet, in another example of Illiac -IV, we state it's system specification as follows-

$$T(\text{Illiac-IV}) = <1, 64, 64>$$

Note here that K', D' and W' are being omitted as they are all 1.

1.9.4 Classification based on Coupling between processing elements (PEs)

Coupling refers to the way in which PEs cooperate with one another. A system may be:

1. Loosely Coupled : The degree of coupling between the PEs is less.

Example. A parallel computer consisting of workstations connected together by LAN (like Ethernet) is **loosely coupled**. In this case, each one of the workstations work independently. If they want to cooperate, they will exchange messages. Thus, logically they are autonomous and physically they do not share any memory and communicate via I/O channels.

2. Tightly coupled : The degree of coupling between the PEs is more. A tightly coupled parallel computer shares a common main memory (MM). So, the communication among PEs is very fast. The cooperation may be at the level of instructions also. It is carried out by each PE as they share a common memory.

We have already tabulated the differences earlier in chapter - 1, Section 1.7.

1.9.5 Classification based on Modes of Access Memory

This has already been discussed in section 1.7 of this chapter - 1.

1.9.6 Classification based on Grain Size

We shall now describe this type of classification. What is a grain size? We define it as follows :

"The quantum of work done by a PE before it communicates with another processor is called as the grain size."

It determines the frequency of communication between PEs during the solution of a problem. This will correspond to the number of instructions carried out by PE before it sends the results to a cooperating PE. The grain sizes are classified as :

- (a) Very fine grain.
- (b) Fine grain.
- (c) Medium grain.
- (d) Coarse grain.

We shall explain each one by one.

Very fine grain parallelism : Instruction level parallelism is exploited by having multiple functional units and overlapping steps during instruction execution. Thus, ILP exploits very fine grain parallelism and VLIW processor uses it effectively.

Fine grain parallelism : It will correspond to a thread. A thread may be the instructions belonging to one iteration of a loop, typically 100 machine instructions. Tightly coupled parallel computers can exploit fine grain parallelism effectively, as the processors cooperate via a shared memory and time to access a shared memory is small (a few machine cycles).

Medium grain parallelism : It will correspond to a procedure (or subroutine). This grain is formed of 100 machine instructions.

Coarse grain parallelism : It will correspond to a complete program.

Please note that loosely coupled parallel computers can exploit only medium or coarse grain parallelism as the time to communicate results using a communication network will be a few 100 clock cycles and the quantum of work done by processors must be larger than this.

1.10 INSTRUCTION LEVEL PARALLELISM AND THREAD LEVEL PARALLELISM

Pipelining can overlap the execution of instructions when they are independent of one another. This potential overlap among instructions is called as instruction level parallelism (ILP). It is so called since then the instructions can be evaluated in parallel.

The amount of parallelism available within a basic block (a straight line code sequence with no branches) is quite small. For a typical MIPS program, the average dynamic branch frequency is between 15% and 25%. It means that between four and seven instructions execute a pair of branches. Since, these instructions are likely to depend upon one another, the amount of overlap we can exploit within a basic block is likely to be much less than the average basic block size. To obtain substantial performance enhancement, we must exploit ILP across multiple basic blocks.

Actually, processing of instructions in parallel requires three major tasks. They are-

Task-1 : Checking dependencies between instructions to determine which instructions can be grouped together for parallel execution. Parallelism among instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in the pipeline. This increases pipeline utilization (*i.e.*, it is now full).

Task-2 : Dynamic scheduling is used where the hardware rearranges instruction execution to reduce the hazards.

Task-3 : Scoreboarding is the process of allowing instructions to execute out of order when there are sufficient resources and no data dependencies.

Thread level parallelism (TLP) means use of multiple threads of execution that are inherently parallel. A **thread** is a separate process with its own instructions and data. A thread may be a process that is part of a parallel program or it may represent an independent program on its own. A thread has all states for it to execute.

TLP is an important alternative to ILP because it could be more cost-effective to exploit than ILP.

1.11 PERFORMANCE OF PARALLEL PROCESSORS – METRICS & MEASURES

The program behaviour and the machine capability need to be matched for an ideal performance from a machine. It is not easy to achieve this perfect match. Better hardware technology, innovative architectural features and better resource management can enhance machine capability. However, it is difficult to predict program behaviour. This is because it depends totally on the application and run-time conditions. Other factors like algorithm design, data structures, language efficiency, programmer skill and technology, also affect the program behaviour.

Also note that the machine performance may change from program to program. So, it is difficult to achieve peak performance in real-life applications.

Next, we shall discuss the fundamental factors that affect the performance of a computer. They are as follows-

1. **Parallel Run Time, ($T(n)$)** : It is defined as the time required to run a program on an n -processor computer.

If $n = 1$, $T(1)$ denotes the run time of a program on a single processor (Sequential).

2. **Speed-up, ($S(n)$)** : It is defined as the ratio of the time taken to run a program on a single processor ($T(1)$) to the time taken to run it on a parallel computer with identical processors ($T(n)$).

$$\therefore S(n) = \frac{T(1)}{T(n)} \quad \dots(1)$$

Speed-up metric is a quantitative measure of performance. It denotes the relative benefit of running a program in parallel.

The speed up achievable by a parallel computer with n -identical processors working concurrently on a single problem is at most n -times faster than a single processor. Please understand that practically speaking, the speed-up is much less because some PEs are idle at a given time due to conflicts over memory access or communication paths, inefficient algorithms for exploiting the natural concurrency in given problem. Fig. 1.22 shows the various estimates of the actual speed-up, ranging from a lower-bound, $\log_2 n$ to an upper bound $n/\ln n$. Here, the lower bound of $\log_2 n$ is known as the **Minsky's conjecture**. Most of the commercial multiprocessor systems have no. of PEs (n) from 2 to 4. Exploratory research multiprocessors have challenged $n = 16$ (PEs) in C.mmp and S-1 systems. Also note that using Minsky's conjecture, only a speed-up of 2 to 4 can be expected from existing multiprocessors with 4 to 16 processors.

This seems to be rather pessimistic, (not sure). A more optimistic (sure) speed up estimate is upper bounded by $n/\ln n$. Let us prove it.

Proof : Consider a computing problem which can be executed by a uniprocessor in unit time, $T_1 = 1$. Let f_i be the probability of assigning the same problem to i -processors working equally with an average load, $d_i = 1/i$ per processor. Also assume equal probability of each operating mode using i -processor i.e.,

$$f_i = \frac{1}{n}, \text{ for } n \text{ operating modes : } i = 1, 2, \dots, n.$$

\therefore Avg. time required to solve the problem on an n -processor system,

$$\begin{aligned} T_n &= \sum_{i=1}^n f_i d_i \\ &= \sum_{i=1}^n \frac{1}{i} \\ &= \frac{n}{\ln n} \end{aligned} \quad \dots(2)$$

\therefore The average speed-up (s) is obtained by

Substituting (2) in (1) as follows

$$S = \frac{T_1}{T_n} = \frac{n}{\sum_{i=1}^n i} \leq \frac{n}{\ln n} \quad \dots(3)$$

Hence, proved.

Note : For a given multiprocessor system with 2, 4, 8, or 16 processors, the respective average speed ups are 1.33, 1.92, 3.08 and 6.93 (from equation (3) above). The speed-up (s) as derived in equation-1 can be approximated by $n/\ln n$ for large n .

$$\text{For e.g., } S = \frac{1000}{\ln 1000} = 144.72 \text{ for } n = 1000 \text{ PEs.}$$

This can be shown in a graph form also.

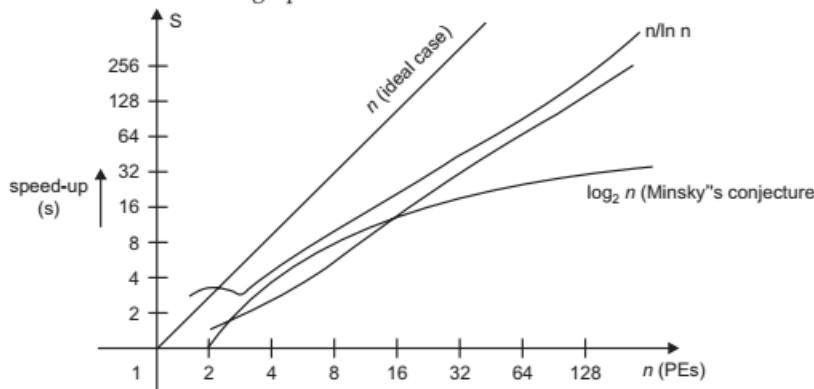


Fig. 1.22. Various estimates of speed up of an n -processor system over a single processor.

This is the reason as to why a typical commercial multiprocessor system consists of only 2 to 4 PEs only. According to Dr. John Worlton of USA Los Alamos Scientific Lab. — "The designers of supercomputers will do better at exploiting concurrency in the computing problems if they use a small number of fast processor instead of a large number of slower processors." This conclusion coincides with the proofs given above.

Other measurement parameters

1. **Pipelined uniprocessor systems** are still market dominating. Pipelined computers cost less and their OS are well developed to achieve better resource utilization and higher performance.

2. **Array processors** are usually custom designed for some specific applications.

Their performance/cost ratio is low. Also note that programming in an array processor is much more difficult due to its rigid structure.

3. **Parallelism Profile in Programs** : Degree of parallelism is the measure of the extent to which software Parallelism and hardware parallelism match. There are 2 things that we usually talk about –

1. Degree Of Parallelism (DOP)
2. Average Parallelism.

The execution of a program on a parallel computer may use different number of processors at different time periods during the execution cycle. For each time period, the number of processors used to execute a program is defined as the degree of parallelism (DOP). Please note that DOP is a discrete time function, assuming only non-negative i.e., positive integer values. The plot of DOP as a function of time is called as the parallelism profile of a given program.

Average Parallelism is, however, different. Consider a parallel computer consisting of ' n ' homogeneous processors. The maximum parallelism in a profile is ' m '. In the *ideal case*, $n \gg m$. The computing capacity (Δ) of a single processor is approximated by the execution rate, such as MIPS or MFLOPS, without considering the penalties from memory access, communication latency or system overhead. When ' i ' processors are busy during an observation period, we have $DOP = i$.

So, the total amount of work 'W' (instructions or computations) performed is proportional to the area under the profile curve.

$$\therefore W = \Delta \int_{t_1}^{t_2} DOP(t) dt \quad \dots(1)$$

$$= \Delta \sum_{i=1}^m i \cdot t_i \quad \dots(2)$$

Where

t_i = total amount of time that $DOP = i$ and

$$\sum_{i=1}^m t_i = (t_2 - t_1) \text{ is the total elapsed time.}$$

Then, the average parallelism is given by -

$$A = \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} DOP(t) dt \quad \dots(3)$$

For example : Consider the figure 1.22 below :

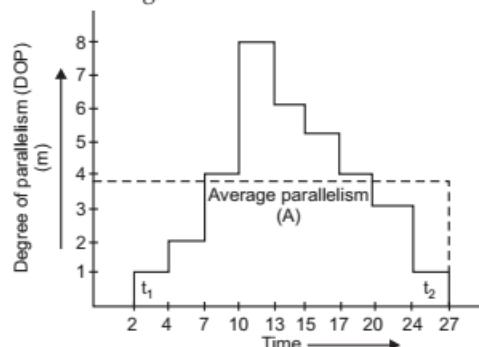


Fig 1.23. Parallelism profile of a divide-and-conquer algorithm.

As it is clear from Fig. 1.22 that the parallelism profile ' m ' increases from 1 to 8 and then decreases to 0 during the time period (t_1, t_2) . So, the average parallelism, A , is computed.

4. Clock rate and CPI : It is the clock that drives the CPU. It has a constant cycle time, τ . τ is measured in nanoseconds. Clock rate is defined as the inverse of cycle time.

\therefore

$$f = \frac{1}{\tau}$$

Where ' f ' is the frequency (or clock rate).

The program size is decided by its **instruction count** (I_c) i.e., the number of machine instructions to be executed in the program. Different machine instructions require different number of clock cycles to execute. Therefore, **cycles per instruction** (CPI) becomes an important parameter to measure the time taken to execute each instruction.

∴

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count } (I_c)}$$

But CPU Time = (CPU clock cycles for a program) × (clock cycle time) ... (1)

$$\text{But } \tau = \frac{1}{f} \quad \dots(2)$$

$$\therefore \text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{clock rate}} \quad \dots(3)$$

$$\text{But } \text{CPU time} = \text{Instruction count} * \text{CPI}$$

(from equation - 1)

$$\therefore = \frac{\text{Instruction count} * \text{clock cycle time}}{\text{Clock Rate}} \quad \dots(4)$$

∴

$$\text{CPU time } (T) = I_c \times \text{CPI} \times \tau \quad \dots(5)$$

$$\text{or } T = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

Where I_c = Instruction count in a program

and τ = clock period.

5. MIPS Rate and MFLOPS : Let C be the total number of clock cycles required to execute a given program. Then, the CPU time (T) is given by :

$$T = C \times \tau = \frac{C}{f} \quad \dots(6)$$

∴

$$\text{CPI} = \frac{C}{I_c} \quad \dots(7)$$

and

$$\begin{aligned} T &= I_c \times \text{CPI} \times \tau \\ &= I_c \times \frac{\text{CPI}}{f} \end{aligned} \quad \dots(8)$$

The speed of the processor is usually measured in terms of **million instructions per second** (MIPS).

$$\text{MIPS rate} = \frac{I_c}{T \times 10^6} = \frac{f}{\text{CPI} \times 10^6}$$

$$\boxed{\text{MIPS} = \frac{f \times I_c}{C \times 10^6}} \quad \dots(9)$$

MIPS rate varies with the clock rate (f), instruction count (I_c) and CPI of a given machine.

MFLOPS – Million floating point operations per second is emerging as a performance measure. MIPS and MFLOPS ratings are not convertible because they measure different ranges of operations. Both are used to describe instruction execution rate and floating point capability of a parallel computer.

(6) **Efficiency ($E(n)$) or η** : Efficiency of a program, on n processors is defined as the ratio of speed-up achieved and the number of processors used to achieve it.

$$\begin{aligned}\therefore \eta = E(n) &= \frac{s(n)}{n} = \frac{\text{Actual speed-up}}{\text{Ideal speed-up}} \\ &= \frac{T(1)}{n[T(n)]}\end{aligned}$$

Where

$T(1)$ = Time taken to execute a program on sequential or uniprocessor system.

$T(n)$ = Time taken to execute same program with ' n ' number of processors by parallel system.

Please note that $E(n)$ or η is the measure of the fraction of time for which a processor, is usefully employed.

$$\begin{aligned}\therefore 1 &\leq S(n) \leq n \\ \text{and} \quad \frac{1}{n} &\leq \eta \leq 1.\end{aligned}$$

Also note that the maximum efficiency of 1 is never achieved due to parallel overheads. Consider Fig. 1.23 below. It shows three different curves :

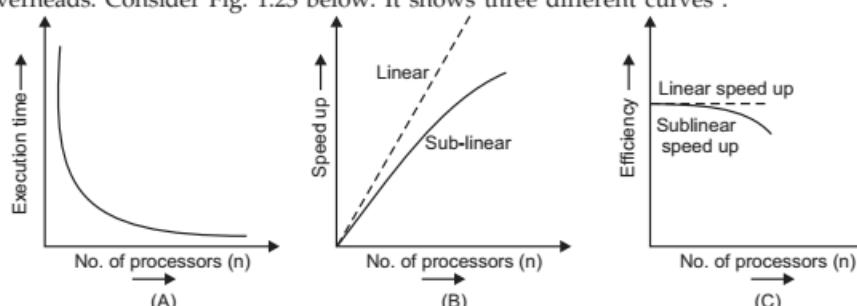


Fig. 1.23

Ideally, the speed-up should be linear. But generally it decreases due to parallel overhead. Parallel computers do not achieve linear speed-up or an efficiency of one because of parallel overheads due to Inter process communications, load imbalance, intertask synchronization etc. All the above sources are application or problem dependent. Efficiency indicates the actual speed-up achieved as compared with maximum value. Efficiency is lowest when the entire program is executed sequentially on a single processor and it is maximum when all ' n ' processors are fully utilized during complete execution period.

7. **Redundancy, $R(n)$** : If $O(n)$ represents the total number of unit operations performed by a n -processor system then Redundancy in a parallel computation is defined as :

$$R(n) = \frac{O(n)}{O(1)} = \frac{\text{Unit operations on } n\text{-processor system}}{\text{Unit operations on 1-processor system}}$$

It indicates the extent of matching between software parallelism and the hardware parallelism. It is obvious that :

$$1 \leq R(n) \leq n$$

8. Utilization, $U(n)$: System utilization signifies the percentage of resources like processors, memories etc. that are busy during the execution of parallel program. It is defined as :

$$U(n) = R(n). E(n) = \frac{O(n)}{nT(n)}$$

9. Quality of Parallelism, $Q(n)$: $Q(n)$ is directly proportional to speed-up, $S(n)$, and efficiency, $E(n)$ and inversely related to redundancy, $R(n)$.

$$\therefore Q(n) = \frac{S(n)E(n)}{R(n)}$$

$Q(n)$ is always upper-bounded by speed-up, $S(n)$ as $E(n) < 1$ and $R(n) < n$.

10. Throughput Rate, W_s : A system throughput is defined as the number of programs executed per unit time on the system. Mathematically,

$$W_s = \frac{\text{Programs}}{\text{Second}} \text{ (system throughput).}$$

11. CPU Throughput, W_p : It is given by the number of instructions executed per second. Mathematically,

$$W_p = \frac{f}{I_c \times CPI}$$

Where

f = clock rate

I_c = Instruction Count

CPI = cycles/instructions.

$$\therefore W_p = \frac{MIPS \times 10^6}{I_c} \text{ (from equation 9)}$$

But $W_s < W_p$ due to additional overheads of the system because of I/O, compiler and OS, when multiple programs are interleaved for CPU execution by multiprogramming or time sharing operations.

Scalability Metrics

The following metrics affect the scalability of a system-

1. Machine size (n): The number of processors employed in a parallel computer system. A large machine size means more resources and more computing power.

2. Clock rate (f): It determines the basic machine cycle. We try to build a machine with components like CPUs, Memory, bus, network etc, driven by a clock which can scale up with better technology.

3. Problem size (s): The amount of computational workload or the number of data points used to solve a given problem. The problem size(s) is directly proportional to the sequential execution time $T(s, 1)$ for a uniprocessor system because each data point may demand one or more operations.

4. CPU time (T): The actual CPU time (in secs) elapsed in executing a given program on a parallel machine with n processors collectively. This is parallel execution time, denoted as $T(s, n)$ and is a function of both s and n .

5. **I/O demand (d):** The I/O demand in moving the program, data and results associated with a given application run. The I/O operations may overlap with the CPU operations in a multiprogrammed environment.

6. **Memory Capacity (m):** The amount of main memory (in bytes or words) used in a program execution. Please note that the problem size, the program size, the algorithms affect the memory demand and the data structures used. Also note that the memory demand varies dynamically during program execution. Here, we refer to the maximum number of memory words demanded virtual memory is almost unlimited with a 64 bit address space. It is the physical memory, which is limited in capacity.

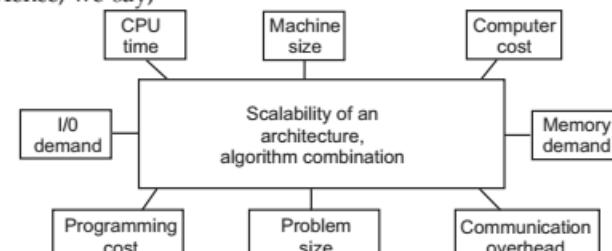
7. **Communication overhead (h):** The amount of time spent for interprocessor communication, synchronization, remote memory access etc. This overhead includes all non-computing operations, which do not involve the CPUs or I/O devices. This overhead $h(s, n)$ is a function of s and n and is not the part of $T(s, n)$. For a uniprocessor system, the overhead $h(s, 1) = 0$.

8. **Computer Cost (c):** The total cost of hardware and software resources required to carry out the execution of a program.

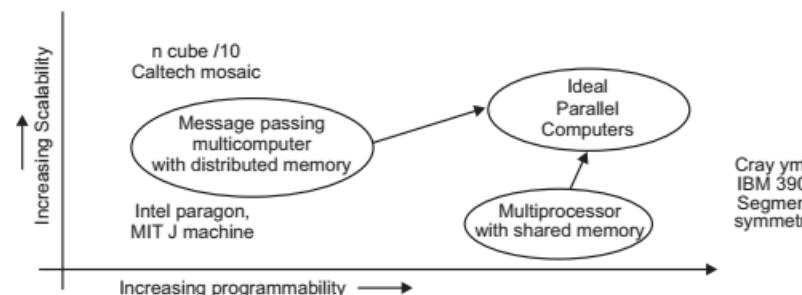
9. **Programming overhead (p):** The development overhead associated with an application program . Programming overhead (p) may slow down software productivity and thus requires a high cost. Unless stated, both computer cost and programming cost are ignored in our scalability analysis.

Note: Depending on the computational objectives and resource constraints imposed, one can fix some of the above parameters and optimize the remaining ones to achieve the highest performance with the lowest cost.

Hence, we say,



We can compare programmability with scalability



1.12 DISTRIBUTED PROCESSING

It means, basically, a distributed shared memory concept. Each processor has access to the whole memory using a single memory address space. If a PE has to access

a location not in its local memory, message passing must occur to pass data from the processor to the location or from the location to the processor. However, this is done in an automated way that hides the fact that the memory is distributed. This is actually called as a **shared virtual memory**. It gives an illusion of shared memory even when it is distributed.

Li was the first person to develop shared virtual memory concept. A unique way of achieving shared memory with a system in which the memory is physically distributed is to provide only cache memory local to each processor and have no main memory at all. The data is copied between caches when needed. As an example, remember that KSR1 multiprocessor uses this technique. So, we draw a shared memory (distributed) multiprocessor in Fig. 1.24.

1.13 PRINCIPLES OF SCALABLE PERFORMANCE

Increasing the number of PEs decreases efficiency and increasing the amount of computation per processor increases efficiency. Our objective is to keep the efficiency fixed by increasing both the size of the problem and the number of processors simultaneously. A parallel computing system (*i.e.*, parallel algorithms + parallel machine) is said to be scalable if its efficiency can be fixed by simultaneously increasing the machine size and the problem size. The scalability of a parallel system is a measure of its capacity to increase speed-up in proportion to the machine size. As the number of PEs increases, in a supercomputer, the fixed load is distributed to more processors for parallel execution. Our primary goal is that we must achieve minimum turnaround time. The speed-up obtained for time critical applications is called **fixed-load speed-up**.

1.14 SPEED-UP PERFORMANCE LAWS

Now, we are in a position to discuss some laws related to the speed-up of supercomputer or parallel computers.

1.14.1 Amdahl's Law – for fixed workload

Assuming there will be some parts that are only executed on one processor, the ideal situation would be for all available processors to operate simultaneously for the other times. If the fraction of computation that cannot be divided into concurrent tasks is, f and no overhead incurs when the computation is divided into concurrent parts, the time to perform the computation with n processors is given by.

$$ft_s + (1 - f)t_s/n.$$

So, we draw now.

Illustrated is the case with a single serial part at the beginning of computation, but the serial part could be distributed throughout the computation. Hence the speed-up factor is given by :

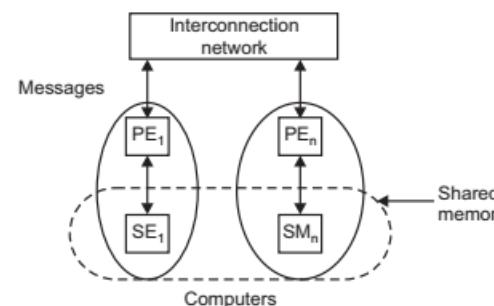
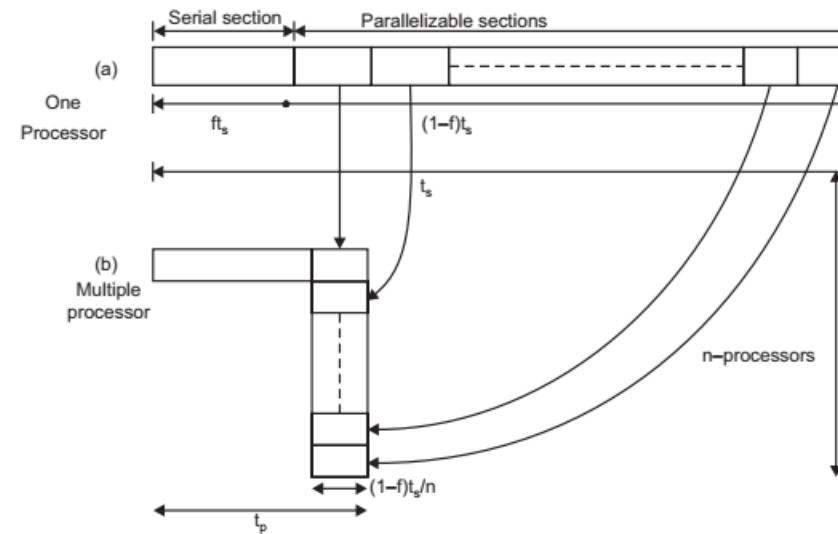


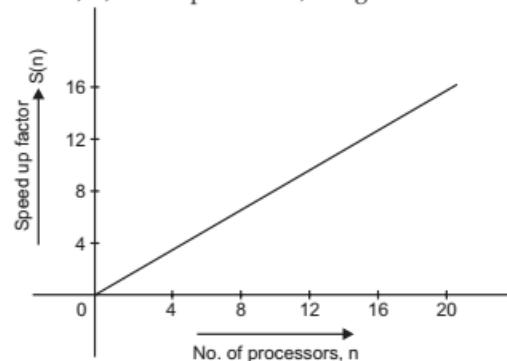
Fig. 1.24. A distributed (shared) memory processor

**Fig. 1.25.** Amdahl's law

$$S(n) = \frac{t_s}{f t_s + (1-f)t_s / n} = \frac{n}{1 + (n-1)f}$$

This equation is known as Amdahl's law.

If we plot $S(n)$ versus, n , no. of processors, we get :

**Fig 1.26.** Speed-up versus n

We see that indeed a speed improvement is indicated. Even with an infinite number of processors, the maximum speed up is limited to $1/f$ i.e.,

$$\text{Lt. } S(n) = \frac{1}{f} \quad n \rightarrow \infty$$

Actually Gene Amdahl law (1967), assumes that the addition of processor cores is perfectly scalable. This law shows that the maximum benefit a program can expect from parallelizing some portion of the code is limited by the serial portion of the code For e.g., As per Amdahl's law, if 30% of your application is spent in serial code, the maximum speed up that can be obtained is 30x, regardless of the number of processor cores.

Please note that endlessly increasing the processor cores only affects the parallel portion of the denominator. Hence, if a program is only 5% parallelized than the maximum theoretical benefit is that the program can run in 95% of the sequential time.

Corollary: "Decreasing the serialized portion by increasing the parallelized portion is of greater importance than adding more processor cores."

For e.g., If you have a program that is 30% parallelized running on a dual-core system, doubling the number of processor cores reduces run time from 85% of the serial time to 77.5%. On the other hand, doubling the amount of parallelized code reduces run-time from 85% to 70%.

Also note that when a program is mostly parallelized, adding more processors helps more than parallelizing the remaining code.

Shortcomings of Amdahl's Law :

1. The work load cannot scale to match the available computing power as the machine size increases. Hence, the fixed load prevents scalability in performance.
2. The sequential bottleneck is a serious problem.

1.14.2 Gustafson's Law-for Scaled Problem

In 1988, John Gustafson gave a law which overcomes the drawback of Amdahl's law. This law states that any sufficiently large problem can be efficiently parallelized; the sequential operations will no longer be a bottleneck if the number of parallel operations in the problem is scaled-up sufficiently. Gustafson's law addresses the shortcomings of Amdahl's law that cannot scale to match availability of computing power as the machine size increases. Gustafson suggested a solution over sequential bottleneck of Amdahl's law by relaxing the restriction of fixed size of the problem and by the notion proposing of fixed execution time. In critical accuracy application, it is always desirable to solve the largest problem size possible on a larger machine rather than solving a smaller problem on a smaller machine with almost the same execution time. In practice, the problem size is not independent of the number of processors. Rather than assuming that the problem size is fixed, we should assume that the parallel execution time is fixed. In increasing the problem size, Gustafson also makes the case that the serial section of the code does not increase as the problem size. So, with the constraint of constant parallel execution time, the resulting speed-up factor will be numerically different from Amdahl's speed-up factor and is called as scaled speed-up factor.

In Amdahl's law, we fix the total execution time on a single processor as a constant. But, in Gustafson's law, the parallel execution time is constant rather than the serial execution time. Let us now derive the Gustafson's equation.

Let the execution time of the parallel computer be a serial part and a parallel part, $S + p$. For convenience, let

$$S + p = 1 \text{ (say)} \quad \dots(1)$$

So, the execution time on a single computer would be $(S + pn)$ as the n parallel parts must be executed sequentially.

\therefore Scaled speed-up factor $S'(n)$

$$= \frac{S + np}{S + p} = \frac{S + np}{1} \quad (\text{from (1)})$$

$$S'(n) = n + (1 - n)s$$

This equation is called as **Gustafson's Law**. A plot of $s'(n)$ and f , the sequential fraction of the program is shown in Fig. 1.27 below.

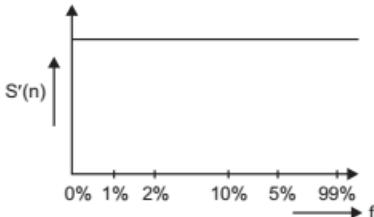


Fig. 1.27. Effect of Gustafson's law

Please note that the slope of the curve of Fig. 1.27 is much more flatter than in case of $S(n)$ in Amdahl's law.

Suppose that we had a serial section of 5% and 20 PEs, then as per the definitions of Gustafson's law, the scaled speed up factor is :

$$\begin{aligned} S'(n) &= 0.05 + 0.95(20) \\ &= 19.05 \end{aligned}$$

Whereas it is only 10.26 according to the Amdahl's law. Speed-ups of 1021, 1020 and 1016 have been achieved in practice with 1024 PEs. Thus, we can conclude that there will not be a problem of sequential bottleneck when the problem can scale to match the available computing power.

1.14.3 Memory bounded Speed-up Model – Sun and Ni's Law

In 1993, Xian-He Sun and Lionel Ni proposed this law. They developed a memory-bounded speed-up model. The idea was to solve the largest possible problem, limited only by the available memory capacity. This may result in some increase in execution time to achieve scalable performance. This law is a generalisation of Amdahl's law and Gustafson's law. The model maximizes the use of both CPU and memory capacity.

Let us derive its equation now.

Say, M = represents the memory requirement of a given problem.
and W = represent the computational workload.

Now, M and W are related to each other depending on the address space and the architectural constraints.

$$\begin{aligned} \text{Let } W &= g(M) \\ \text{or } M &= g^{-1}(W) \end{aligned}$$

In a multicomputer, the total memory capacity increases linearly with the number of nodes available. The execution time, T , will be a function of number of processors, n and the workload, W and will also be related to memory requirement, M .

$$\begin{aligned} \therefore T &= f(M) \\ \text{or } M &= f^{-1}(T) \end{aligned}$$

The enhanced memory can be related to the execution time of the scaled workload as follows :

$$T^* = f^*(nM)$$

Where nM is the increased memory capacity of n processor machine.

$$\begin{aligned} \text{Now, } f^*(nM) &= F(n) \cdot F(M) \\ &= F(n) \cdot T(n, W) \end{aligned}$$

Where $T(n, W) = F(M)$

and f^* is a homogeneous function.

$F(n)$ reflects the increase in the execution time as memory increases n -times.

So, a fixed-memory speed-up is given by :

$$\begin{aligned} S^*(n) &= \frac{T_s + F(n) T_p(n, W)}{T_s + \frac{F(n)}{n} (T_p(n, W))} \\ &= \frac{\alpha + F(n) (1 - \alpha)}{\alpha + \frac{F(n)}{n} (1 - \alpha)} \end{aligned}$$

Where

$$T = T_s + T_p$$

T = Total execution time

T_s = Execution time for sequential load

T_p = Execution time for parallel load

α = Sequential fraction of a program

This equation of $S^*(n)$ for fixed-memory speed-up has been derived above on the basis of two assumptions :

1. All the individual memory spaces forms a global address space. It means that there is a distributed shared memory space.
2. Full memory capacity is used up for solving scaled problem.

Now, three special cases arise.

Case 1 : When $F(n) = 1$

This is when the problem size is fixed. In this case, the **fixed memory speedup becomes equivalent to Amdahl's law**, when fixed workload is given.

Case 2 : When $F(n) = n$

This is when workload increases n -times, when memory is increased n -times.

So, the fixed memory speed up becomes equivalent to Gustafson's law, when execution time is fixed.

Case 3 : When $F(n) \geq n$

This is the case in which the computational workload increases faster than the memory requirement. Thus, memory bound or fixed memory model may give a higher speed up than the fixed execution time (Gustafson's) model. So, we conclude 2 things :

1. Amdahl's law and Gustafson's law are special cases of Sun and Ni's law.
2. When computation grows faster than the memory requirement, the memory bound model may yield a higher speed up and better resource utilization.

∴

$$S_n^* \geq S'_n \geq S_n$$

1.15 CASE STUDY OF INTEL ITANIUM PROCESSOR

The first implementation of IA-64 architecture is *Itanium processor*. It was jointly developed by HP and Intel. It has a blend of superscalar features.

The block diagram of Itanium processor is shown in Fig. 1.28. From the Fig. 1.28 it is clear that Itanium architecture includes :

- (a) Upto 6 instructions of 64-bit per clock cycle can be executed.
- (b) It has 128 General Purpose Registers (GPRs)
- (c) It has 3 level cache.
- (d) It has large address space : 50 bit physical/64-bit virtual.
- (e) Very few transistors in each core.
- (f) It has nine functional units. All of them are pipelined.
- (g) Instructions are fetched from L1 cache.
- (h) Decoupling buffer is used to hold upto 8 bundles of instructions. It is fed by L1 instruction cache.

The organization of Itanium CPU is simpler than a conventional superscalar organization. No reservation stations (to study later on), no reorder buffers and no memory ordering buffers are observed. There is no register dependency-detection logic. It is replaced by explicit parallelism.

There are three levels of cache L1, L2 and L3. L1 cache is divided into a 16 KB instruction cache and a 16 KB data cache.

Each of these are 4 way set associative with 32 byte line size. L2 cache is of 96 KB and is 6-way set associative with 64 byte line size. L1 and L2 caches are on chip with CPU. L3 cache is off-chip but on the same package as the processor.

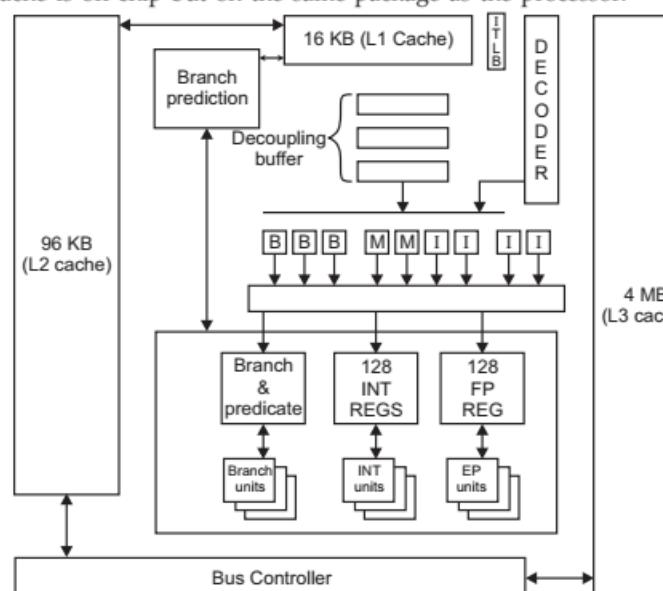


Fig. 1.28. Block diagram of Itanium CPU

SUMMARY

The widespread demand for high performance computers has stimulated a renaissance in computer design. Parallel processing means use of multiple processors to solve a given problem. It is an efficient form of information processing. It emphasizes on concurrency. Concurrency implies parallelism, simultaneity and pipelining. Various classifications of parallel computers have been discussed. ILP and TLP have been discussed. Finally, a case study is given.

MULTIPLE CHOICE QUESTIONS (MCQS) WITH ANSWERS

1. It is possible to achieve parallelism :
(a) With and within the CPU (b) With many CPUs only
(c) Without CPUs (d) All of the above.
2. A CPU is also called as a :
(a) PE (b) Parallel computer
(c) ALU (d) None of the above.
3. 100 MFLOPs corresponds to :
(a) 1/100 million floating point operations/second.
(b) $1/10^{10}$ million floating point operations/second.
(c) 100 million floating point operations/second.
(d) None of the above.
4. Second Generation computers belong to :
(a) 1945–54 (b) 1955–64
(c) 1965–74 (d) None of the above.
5. Third generation computers uses :
(a) Multiprogramming and time-sharing.
(b) Multiprogramming
(c) Multithreading
(d) All of the above.
6. With each new generation, the circuiting size becomes :
(a) Bigger (b) Smaller
(c) Smaller and advanced (d) None of the above.
7. Which of the following computers belong to 5th generation :
(a) Cray-XMP (b) Intel Paragon
(c) VPP 500 (d) Both b and c.
8. ENIAC stands for :
(a) Electronic Numerical Integrator and Calculator.
(b) Electronic Number Integration and Calculation.
(c) Enhanced Number Integrator and Calculator.
(d) None of the above.
9. RTL stands for :
(a) Register Transfer Language (b) Remote Transfer Language
(c) RISC Transfer Language (d) None of the above.

21. The process of allowing instructions to execute out of order when resources are sufficient is called as :

 - (a) Scaffolding
 - (b) Pipelining
 - (c) Score boarding
 - (d) None of the above.

22. The speed-up, $S(n)$, is given by :

 - (a) $S(n) = T(1) * T(n)$
 - (b) $S(n) = T(n)/T(1)$
 - (c) $S(n) = T(1)/(n)$
 - (d) None of the above.

23. The number of machine instructions to be executed in the program is called as the:

 - (a) Cycle
 - (b) Time period
 - (c) Instruction Count
 - (d) None of the above.

24. The equation for Amadahl's law is :

 - (a) $S(n) = \frac{1}{f}$
 - (b) $S(n) = f$
 - (c) $S(n) = \frac{1}{T}$
 - (d) None of the above.

25. In temporal parallelism, 'temporal' refers to :

 - (a) Time
 - (b) Space
 - (c) PEs count
 - (d) None of the above.

26. Performance (P) and execution time (T) of CPU are related by:

 - (a) $P \propto T$
 - (b) $P \propto \frac{1}{T}$
 - (c) $P = T$
 - (d) $P + T$

27. David Kuck of university of Illininos adopted :

 - (a) Implicit Parallelism
 - (b) Explicit Parallelism
 - (c) Both a and b
 - (d) None of the above.

ANSWERS

- | | | | | |
|---------|---------|---------|---------|---------|
| 1. (a) | 2. (a) | 3. (c) | 4. (b) | 5. (a) |
| 6. (c) | 7. (d) | 8. (a) | 9. (a) | 10. (c) |
| 11. (a) | 12. (b) | 13. (b) | 14. (a) | 15. (c) |
| 16. (a) | 17. (a) | 18. (b) | 19. (b) | 20. (a) |
| 21. (c) | 22. (c) | 23. (c) | 24. (a) | 25. (a) |
| 26. (b) | 27. (a) | | | |

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

1. What are the different types of parallelism ?

Ans. Parallelism is of two types :

(a) **Hardware parallelism**. It is built into the machine's architecture.

(b) **Software parallelism.** It is exploited by the concurrent execution of machine language instructions in a program as coded by a programmer.

2. Distinguish between temporal and Data Parallel processing ?

Ans. The following table summarizes the differences between the two :

Temporal Parallelism	Data Parallelism
<ul style="list-style-type: none"> 1. In this parallelism, the job is divided into a set of independent tasks and tasks are assigned for processing. 2. Tasks should take equal time. 3. Pipeline stages should be synchronized. 4. Bubbles in jobs may lead to idling of PEs. 5. PEs are specialized to perform tasks. 	<ul style="list-style-type: none"> 1. Full jobs are assigned for processing. 2. Jobs may take different times. 3. No need to synchronize beginning of jobs. 4. Bubbles in jobs do not cause idling of processors. 5. PEs are general purpose.

3. Give some examples of India's supercomputers.

Ans. Some examples of India's Supercomputers are :

1. **PARAM 10000** (by CDAC ; in 1998). It uses Sun Solaris OS for parallel programming. The peak speed is 10G flops.
2. **PACE +** by DRDO ; in 1997). It was ANUPAM programming Environment with FORTRAN and C languages.
3. **ANUPAM Model-2** (by BARC ; in 1997). It uses DEC Alpha processors with DEC UNIX environment.

4. How is bit serial memory accessing different from bit parallel accessing ?

Ans. The first electronic digital computers used a bit serial MM. Each bit of a word was read individually from memory. *For example* : EDSAC, Pilot ACE, EDVAC and UNIVAC systems (all of them had bit serial memories).

Bit-parallel arithmetic became possible once bit-parallel memory was available. *For example* : IBM-701 machine used bit parallel arithmetic.

5. What do you mean by Inter leaved memory ? Distinguish between low-order and high-order inter leaving.

Ans. An interleaved memory is a memory unit divided into a number of modules or banks that can be accessed simultaneously. Each memory bank has its own addressing circuitry. **Instruction and data addresses** are interleaved to take advantage of the **parallel fetch capability**. It can be done in two ways.

(a) **Low-order interleaving.** This method uses low-order bits of an address to determine the memory bank containing the address : For example

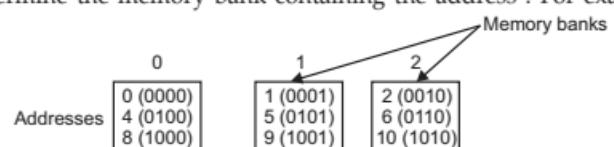


Fig (a). Low-order interleaving lets the low-order address determine the memory bank

(b) **High-order interleaving** : This method uses the high-order bits of an address to determine the memory bank.

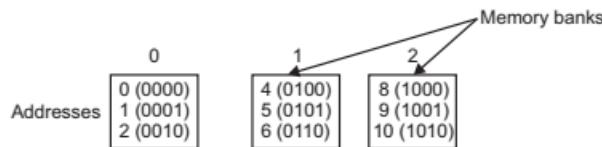


Fig (b) : High-order interleaving lets the high-order address determine the memory bank.

Note :

1. During computer designing, it is very important to match the speed of the various components.
2. IBM STRETCH computer was the first computer to have an interleaved memory.
3. The advantage of having two memory banks was that the maximum data transfer rate to and from the memory was increased by a factor of 2 (2-times)
6. A program is run on a 40MHZ CPU with the instruction mix and corresponding clock cycle count as given in the table below. Determine
 - (a) Effective CPI.
 - (b) Execution time.
 - (c) MIPS rate for the program. [UPTU, B. Tech (CSE) 7th Sem.; 2003-04] and [GGSIPU, M.Tech (IT)-1st sem., Jan-2011]

The table is shown below :

Instruction Type	Clock cycle Count	Instruction Count
1. Integer Arithmetic	1	45,000
2. Floating point	2	32,000
3. Data transfer	2	15,000
4. Control Transfer	2	8,000

Ans. As we know that :

$$\text{CPI} = \frac{\text{Total clock cycles required for given } I_c}{\text{Frequency of clock}}$$

Corresponding to the values in the table given above, we find that :

1. Clock cycles required for Integer Arithmetic instructions are :
 $= 45,000 * 1 = 45,000$
2. Clock cycles required for floating-point instructions are :
 $= 32,000 * 2 = 64,000$
3. Clock cycles for Data transfer instructions are :
 $= 15,000 * 2 = 30,000$
4. Clock cycles for control transfer instruction are :
 $= 8,000 * 2 = 16,000$

$$\therefore \text{Total clock cycles} = 1,55,000$$

Also, total number of instructions

$$\begin{aligned}
 (I_c) &= (45,000 + 32,000 + 15,000 + 8,000) \\
 &= 1,00,000
 \end{aligned}$$

$$\therefore I_c = 1,00,000$$

(a) Effective CPI = $\frac{1,55,000}{1,00,000}$
 $= 1.55 \text{ cycles/instruction}$

(b) Execution time for the program

$$\begin{aligned} &= \frac{\text{Total clock cycle}}{\text{clock period}} \\ &= \frac{1,55,000}{40 \times 10^6} \\ &= 3.87 \text{ msec.} \end{aligned}$$

(c) MIPS = $\frac{\text{Clock Rate}}{\text{CPI} \times 10^6}$

$$\begin{aligned} &= \frac{40 \times 10^6}{1.55 \times 10^6} \\ &= 25.8 \text{ MIPS} \end{aligned}$$

7. Calculate the average CPI when the following conditions are given :

Instruction Types	Cycles	Relative Frequency
I	2	30%
II	3	25%
III	5	40%
IV	10	5%

Ans. As already explained, in this chapter that —

$$\begin{aligned} \text{CPI} &= \text{No. of cycles/instruction} \\ &= \text{cycles} * \text{Relative Frequency} \\ \therefore \quad \text{CPI}_I &= 2 \times 0.30 = 0.60 & [\text{From table above}] \\ \text{CPI}_{II} &= 3 \times 0.25 = 0.75 \\ \text{CPI}_{III} &= 5 \times 0.40 = 2.0 \\ \text{CPI}_{IV} &= 10 \times 0.05 = 0.5 \\ \therefore \text{Average} \quad \text{CPI} &= \frac{\text{CPI}_I + \text{CPI}_{II} + \text{CPI}_{III} + \text{CPI}_{IV}}{4} \\ &= \frac{(0.60 + 0.75 + 2.0 + 0.50)}{4} \\ &= \frac{3.85}{4} \\ &= 0.9625 \end{aligned}$$

8. Suppose that the same program is executed on two different machines—A and B. Compilers and data set on both the machines are identical. Given that :

Machine-A : Average CPI = 3.5

cycle time = 1.0 ns

Machine-B : Average CPI = 1.2

cycle time = 0.5 ns

Which out of these two machines is slower ?

[GGSIPU, B.Tech.(CSE) -7th sem-, 2009]

Ans. Machine-A : CPI * cycle time
 = 3.5 * 1.0
 = 3.5

Machine-B : CPI * cycle time
 = 1.2 * 0.5
 = 0.6

∴ Machine-B is faster as it has a lower CPI.

9. How are performance and execution time related to each other ?

Ans. Performance $\propto \frac{1}{\text{Execution time of CPU}}$

So, a computer that executes a program in half the time taken by another machine has twice the performance. Please note that the performance and execution time vary in opposite directions. Also note that doubling processor performance does not mean replacing an xGHz CPU with a 2x GHz CPU.

10. Distinguish between Implicit and Explicit parallelism ?

[GGSIPU, B.Tech (CSE) -7th sem., 1st minors, 2012]

Ans.

Implicit Parallelism	Explicit Parallelism
<ul style="list-style-type: none"> 1. To achieve implicit parallelism, the source code is written in sequential languages like C, FORTRAN 77, LISP or PASCAL. 2. We need a parallelizing compiler. 3. A parallel object code is generated. 4. This approach has been used in the programming of shared-memory multiprocessors. 5. The compiler must be able to detect parallelism. 	<ul style="list-style-type: none"> 1. To achieve explicit parallelism, the source code is written in the concurrent dialects of C, FORTRAN, LISP or PASCAL. 2. Here, we need a concurrency preserving compiler. 3. A concurrent object code is generated. 4. This approach has been used in multi-computer development. 5. The compiler need not detect parallelism as it is explicitly specified in the user programs.

Examples :

- (a) David Kuck of University of Illinois adopted implicit-parallelism approach.
- (b) Ken Kennedy of Rice University used this approach.

11. Consider the execution of an object code with 200,000 instructions on a 40 MHz processor. The program consists of four major types of instructions. The instruction mix and the number of cycles (CPI) needed for each instruction type are given below based on the result of a program trace experiment :

Instruction Type	CPI	Instruction mix
1. Arithmetic and logic	1	60%
2. Load/store with cache hit	2	18%
3. Branch	4	12%
4. Memory reference with cache miss	8	10%

(a) Calculate the average CPI when the program is executed on a uniprocessor.

(b) Also calculate the corresponding MIPS rate ?

[GGSIPU, B.Tech, (CSE) -7th Sem., 2009]

and [GGSIPU, M.Tech (IT) -7th Sem., 2007]

$$(b) \text{ MIPS rate} = \frac{40}{2.24} = 17.86 \text{ MIPS}$$

12. A CPU renders a graphic image in 100 ms with graphics card-A and 125 ms with graphics card-B. What is the speed up?

Ans. Speed-up (s) = $\frac{125}{100} = 1.25$
 $= 25\%$

13. Explain I_C and CPI. Why we calculate average value of CPI. Give an example to illustrate.

Ans. IC - Instruction Count : is defined as the total number of instructions in a program.

CPI - Clock cycles per Instruction: It shows, on average, how many cycles are needed for one instruction.

We calculate an average value of CPI as CPI is not a constant value. It varies from instruction to instruction.

For example

- (a) Memory access takes more time than register access.
 - (b) Floating point operations take more time than integer operations.
 - (c) Multiplication is more time consuming than addition.

Say, if a RISC machine performs 9 instructions per 10 clock cycles then its average CPI is 1.11 clock periods per instruction.

On the other hand, a CISC machine that performs 4 instructions per 10 clock cycles has an average CPI of 2.5 clock periods per instruction.

Formula to calculate CPI—

$$\begin{aligned} \text{Total clock cycles} &= \Sigma CPI_C * (\# \text{ Instructions})_C \\ &= \left[\begin{array}{l} \text{Cycles per instruction} \\ \text{for instruction class } C \end{array} \right] * \left[\begin{array}{l} \# \text{ instructions of} \\ \text{instruction class } C \end{array} \right] \end{aligned}$$

and Execution Time = $(A \text{ instruction} * X \text{ cycles}) +$
 $(B \text{ instruction} * Y \text{ cycles}) +$
 $(C \text{ instruction} * Z \text{ cycles})$

where A , B and C are number of different types of instructions and x , y , z are no. of cycles needed for each type of instruction.

14. Consider the following data for 2-machines A and B:

	# k Register Instructions (1 cycle each)	# k Memory Access Instructions (2 cycles each)	# k floating point Instructions (3 cycles each)
Machine-A	2	1	2
Machine-B	4	1	1

Compare A with B in terms of CPI and clock cycles.

Ans. Machine-A

$$\begin{aligned}\text{Clock-Cycles (A)} &= 2 * 1 + 1 * 2 + 2 * 3 \\ &= 2 + 2 + 6 \\ &= 10 \text{ cycles}\end{aligned}$$

$$\therefore \text{CPI (A)} = \frac{10 \text{ cycles}}{5 \text{ instructions}} = 2$$

Machine-B

$$\begin{aligned}\text{Clock-Cycles (B)} &= 4 * 1 + 1 * 2 + 1 * 3 \\ &= 4 + 2 + 3 \\ &= 9 \text{ cycles}\end{aligned}$$

$$\therefore \text{CPI (B)} = \frac{9 \text{ cycles}}{6 \text{ instructions}} = 1.5$$

Conclusions

Program-A requires fewer instructions but more cycles.

⇒ Higher CPI

and Program-B requires more instructions but fewer cycles.

⇒ Lower CPI.

15. What happens when the clock period changes per computer?

Ans. Say, if clock period = 700 MHz

$$\begin{aligned}\text{Then} \quad \text{Cycle Time} &= \frac{1}{\text{clock-rate}} \\ &= \frac{1}{700 \text{ MHz}} \\ &= \frac{1}{700,000,000 \text{ clock periods per second}} \\ &= 1.4 \text{ ns}\end{aligned}$$

And if clock period = 450 MHz

$$\begin{aligned}\text{Then} \quad \text{Cycle Time} &= \frac{1}{450 \text{ MHz}} = \frac{1}{450,000,000 \text{ clock periods per second}} \\ &= 0.2 \text{ ns.}\end{aligned}$$

But please remember that if CPI and I_C are same then the clock rate determines which computer is faster.

$$\therefore \text{Execution Time} = I_C * \text{CPI} * t$$

Also note that clock rate = cycles /second

and 1 Hertz = 1 cycle/sec

\therefore 1 Mega Hertz (MHz) = $1 * 10^6$ cycles/second.

16. A hardware manufacturer company increases the frequency from 700 MHz to 1.2 GHz. What is the potential speed-up achieved?

Ans. We know that

$$\text{Execution Time} = I_C * \text{CPI} * t$$

$$\text{Given: } t_{\text{old}} = \frac{1}{700} \quad t_{\text{new}} = \frac{1}{1200}$$

$$\therefore \text{Speed-up} = \frac{t_{\text{old}}}{t_{\text{new}}}$$

$$= \frac{I_C * \text{CPI} * \frac{1}{700}}{I_C * \text{CPI} * \frac{1}{1200}} = \frac{1}{700} \times 1200$$

$$= 1.71428$$

$$\therefore \text{Speed-up (\%)} = 71\%.$$

17. Machine-A has a clock cycle time of 1 ns and an average CPI = 2. Machine-B has a clock cycle time of 2 ns and an average CPI = 1.2. Instruction count (I_C) is same which of the two machines is faster?

$$\begin{aligned} \text{Ans. Execution Time (A)} &= I_C * 2 * 1 \text{ ns} \\ &= 2 I_C \text{ ns} \end{aligned}$$

$$\begin{aligned} \text{Execution Time (B)} &= I_C * 1.2 * 2 \text{ ns} \\ &= 2.4 I_C \text{ ns} \end{aligned}$$

$$\therefore \frac{\text{Performance-A}}{\text{Performance-B}} = \frac{\text{Execution Time B}}{\text{Execution Time A}}$$

$$= \frac{2.4 I_C \text{ (ns)}}{2 I_C \text{ (ns)}}$$

$$= 1.2$$

\therefore Machine-A is 1.2 times faster than B.

18. Consider the following data:

Which of the two computers appear faster if it is used to run program-x and program-y:

	Time on Comp-A	Time on Comp-B
Program-x	1	10
Program-y	1000	100

(a) 50% of the time.

(b) 90% and 10% of the time resp.

When would computer-A appear faster than computer-B?

Ans. (a) If the given two programs x and y are used 50% of the time then the usage is:

$$\begin{aligned}\text{Comp}_A &= (0.5) * 1 + (0.5) * 1000 \\ &= 500.5 \\ \text{Comp}_B &= (0.5) * 10 + (0.5) * 100 \\ &= 55\end{aligned}$$

\therefore Computer-B appears faster than Computer-A.

(b) If the two programs are used 90% and 10% of the time then, usage is:

$$\begin{aligned}\text{Comp}_A &= (0.9) * 1 + (0.1) * 1000 \\ &= 100.9 \\ \text{Comp}_B &= (0.9) * 10 + (0.1) * 100 \\ &= 19\end{aligned}$$

\therefore Computer-B is still faster.

Now, Comp_A will run faster than Comp_B if and only if:

$$\begin{aligned}1n + 1000(1-n) &< 10n + 100(1-n) \\ \text{or } (1000 + 1n - 1000n) &< (10n - 100n + 100) \\ \text{or } 900 &< 909n \\ \text{or } \frac{900}{909} &< n \\ \text{or } n &> 0.9900990099 \\ \therefore \text{Comp}_A &= 0.99(1) + 0.01(1000) = 10.99 \\ \text{Comp}_B &= 0.99(10) + 0.01(100) = 10.9.\end{aligned}$$

EXERCISE QUESTIONS

1. Write short notes on (a) Application Areas Supercomputers.
[B.E. (CSE); Pune Univ., 2001, 2002, 2005, 2006]
(b) Techniques used in B.W. matching of a uniprocessor ?
2. "Finer the grain size, the higher the potential for parallelism, but higher the communication and scheduling overhead." Justify this statement.
[B.E. (CSE); Pune Univ. 2001]
3. What are the fundamental issues in parallel processing ?
[B.E. (CSE) ; Pune Univ. 2004]
4. Explain Feng's classification. [UPTU, B. Tech (CSE) 8th Sem. 2004-05 & 2005-06]
& [B.E. (CSE) ; Pune Univ. 2006]
5. Indicate whether each of the following is true or false. Give reasons :
(a) The CPU computations and I/O operations cannot be overlapped in a multiprogrammed computer.
(b) Synchronization of all PEs in an SIMD computer is done by hardware rather than by software as is often done in most MIMD computers.
(c) As far as programmability is concerned, shared-memory multiprocessors offer simpler inter processor communication support than that offered by a message-passing multicomputer.

- (d) In an MIMD computer, all processors must execute the same instruction at the same time synchronously.
 - (e) As far as scalability is concerned, multicomputers with distributed memory are more scalable than shared-memory multiprocessors.

[GGSIPU, M.Tech (IT), Dec. 2003]

Hint :

- (a) **False.** The fundamental idea of multiprogramming is to overlap the computations of some programs with the I/O operations of other programs.
 - (b) **True.** In an SIMD machine, all processors execute the same instruction at the same time. Hence, it is easy to implement synchronization in hardware. In an MIMD machine, different processors may execute different instructions at the same time. Hence, it is difficult to support synchronisation in hardware.
 - (c) **True.** Inter processor communication is facilitated by sharing variables on a multiprocessor and by passing messages among nodes of a multicomputer. The multicomputer approach is usually more difficult to program since the programmer must pay attention to the actual distribution of data among the processors.
 - (d) **False.** In general, an MIMD machine executes different instruction streams on different processors.
 - (e) **True.** The contention among processors to access the shared memory may create hot spots, making multiprocessors less scalable than multicomputers.

6. Give five performance factors and explain four system attributes ?

[GGSIPU, B.Tech, 7th sem (CSE), Dec. 2004-07]

7. Explain the architectural operations of SIMD and MIMD computers. distinguish between multiprocessors and multicomputers based on their structures, resource sharing and IPC [IGGSIU B.Tech. 7th sem. CSE Dec. 2007]

8. With the help of block diagrams, explain Flynn's classification of computer architectures. [GGSIUPU, M.Tech (IT) 1st sem ; Dec. 2004 & IPUU, B. Tech (CS) 7th Sem : 2005-06 & 2008-09]

9. (a) Define the following terms :

(b) What is a supercomputer ? Explain the need for supercomputers.

[UPTU] B. Tech (CSE) 8th Sem.: 2005-06 & KUD, B.E. (CSE) 8th sem : 1997]

10. (a) Why do you need high speed computing ? List the points and explain in brief.

(b) List and explain the interesting features of parallel computers.

[KUD, B.E. (CSE) : 8th sem : 1997]

11. How does a multiprocessor machine differ from a multicomputer machine ? Give two examples of each.

[Delhi Univ.-ME (CS) : DCE, 2006]

12. (a) Which is the best classification for parallel system – Flynn's or Handler's. Discuss their limitations, if any. [Delhi Univ.-ME (CS) ; DCE. 2006]
(b) How does a multiprocessor machine differ from a multicomputer machine.
(c) What are the limitations of speed-up ? Can we have superlinear speed-up ? Justify your answer. [DU-ME (CSE) DCE. 2006]
13. (a) What are the different methods increasing the speed of computers.
(b) Compare temporal and Data parallel processing methods.

[Delhi Univ., ME (CS), DCE. 2002]

14. Describe a place of the following computer system in the Flynn's classification scheme :
 1. Illiac-IV
 2. nCUBE
 3. Cray T90 series
 4. CM-5
 5. PARAM 10000

[Delhi Univ. ; ME (CS) ; DCE. 2003]

Hint :

1. SIMD
2. MIMD
3. SISD
4. SIMD
5. MIMD

15. Suppose a multiprocessor is built out of individual processors capable of sustaining 50 MFLOPS. What is the largest fraction of a program's execution time that could be devoted to sequential operations if the parallel computer is to exceed the performance of a supercomputer capable of sustaining 1GFLOPS.

[Delhi Univ. ME (CS) ; DCE. 2002]

16. Explain how the degree of paralessim (DOP) and number of processors effect the performance of a parallel computing system. Give Amdahl's law and find the expression for fixed load speed-up. [UPTU, B. Tech (CSE) 8th Sem.; 2005-06]

17. (a) State Amdahl's law and prove it.
(b) Describe with block diagram, shared memory architecture. Enumerate its features.

18. Explain how instruction set, compiler technology, CPU implementation and control and memory hierarchy affect the CPU performance and justify the efforts in terms of program length, clock rate and effective CPI.

[UPTU, B. Tech (CSE) 8th Sem.; 2007-08]

19. (a) Define parallel computing. What are the fundamental using in parallel processing? Discuss various applications of parallel computing?
(b) Explain how DOP and number of processors affect the performance of a parallel computing system. Discuss various speedup performance laws.

[UPTU, B. Tech (CSE) 8th Sem.; 2008-09]

CHAPTER

2

PROGRAM AND NETWORK PROPERTIES

2.0 INTRODUCTION

The aim of a parallel programmer is to exploit parallelism in a given problem. And the ability to execute several program segments in parallel requires each segment to be independent of the other segments. This independence may be data independence, control independence and resource independence. But it is important to understand that all modules/programs need not be independent, *i.e.*, they are dependent. We need to identify such critical statements and to parallelize them.

2.1 CONDITIONS OF PARALLELISM (BERNSTEIN'S CONDITIONS)

Bernstein (in 1966), gave a set of conditions on the basis of which two (2) processes can execute in Parallel.

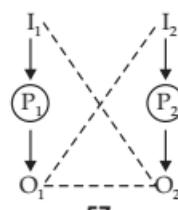
According to Bernstein

- (a) **Process (P_i)** : It is a software entity.
- (b) **Input (I_i)** : It is a set of all input variables (operands) needed to execute the process. These operands can be fetched from **memory or registers**.
- (c) **Output (O_i)** : It is a set of all output variable generated after execution of the process, P_i . These are the results to be stored in **working registers or memory locations**.

In general, we can say :



Consider now that there are *two processes*, P_1 and P_2 . Their **inputs** are I_1 and I_2 and **outputs** are O_1 and O_2 respectively. Now, note here carefully that these two processes can execute in parallel (*i.e.*, $P_1 \parallel P_2$) if they are independent and do not create confusing results. That is,



According to Bernstein, the two conditions for $P_1 \parallel P_2$ (read as P_1 parallel P_2) are given below :

Condition I : $I_1 \cap O_2 = \emptyset$
 Condition II : $I_2 \cap O_1 = \emptyset$
 Condition III : $O_1 \cap O_2 = \emptyset$

These three conditions
are known as Bernstein
conditions.

In general, a set of processes, P_1, P_2, \dots, P_k can execute in parallel if Bernstein's conditions are satisfied on a pairwise basis. This means that :

$P_1 \parallel P_2 \parallel P_3 \parallel \dots \parallel P_k$ if and only if $P_i \parallel P_j$ for all $i \neq j$.

Note. This is true if the output of one process will not be used as input to the other process.

Properties of Parallelism Relation (\parallel)

The following are the properties of parallelism relation in general :

1. **Commutativity.** The parallelism relation (\parallel) is commutative, i.e.,

$P_i \parallel P_j$ implies $P_j \parallel P_i$

2. **Non-transitivity.** The parallelism relation (\parallel) is not transitive i.e.,

if $P_i \parallel P_j$ and $P_j \parallel P_k$

then $P_i \neq P_k$ (i.e., P_i does not parallelize with P_k).

3. **Associativity.** The parallelism relation is associative. That is ,

$(P_i \parallel P_j) \parallel P_k = P_i \parallel (P_j \parallel P_k)$

Please note that the condition $I_i \cap I_j \neq \emptyset$ does not prevent parallelism between two processes. However, violation of any of the three Bernstein's conditions prohibits parallelism between two processes. There are many program constructs that may prohibit parallelism.

Any statements or processes which depend on run-time conditions are not transformed to parallel form. *For example,* If statements or conditional branches. Recursions also prohibit parallelism.

In general, data dependence, control dependence and resource dependence, all will prevent parallelism from being exploited.

Our aim is to identify these dependencies so that parallelization of such segments can be done.

Before discussing these dependencies, we must solve a problem now.

Example 2.1 Consider the following five instructions P_1 to P_5 for some program where each instruction represents one process. Assume that each statement requires one step to execute. There is no pipelining. Show how sequential execution and parallel executions can be done.

$P_1 : C = D \times E$

$P_2 : M = G + C$

$P_3 : A = B + C$

$P_4 : C = L + M$

$P_5 : F = G \div E$

Solution. If these five processes, $P_1 - P_5$ are executed sequentially then we find that 5 steps are needed. That is,

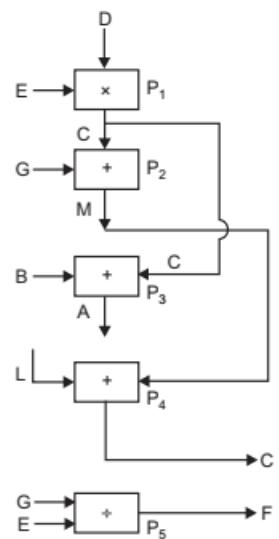


Fig. 2.1. Sequential Executive of $P_1 - P_5$

. \therefore Sequential execution requires 5-steps as shown in Fig. 2.1.

Now consider the **parallel execution** cases. If two adders are available simultaneously, the parallel execution requires only three steps. That is,

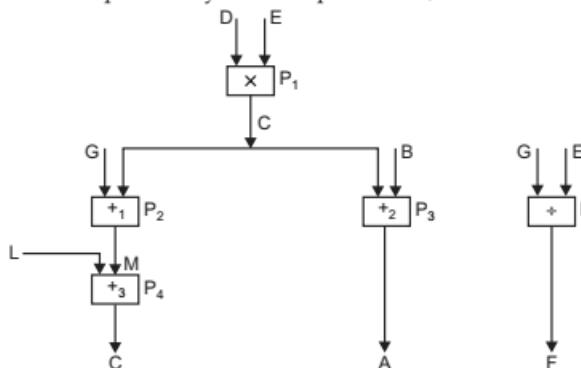


Fig. 2.2. Parallel execution of $P_1 - P_5$

. \therefore Parallel Execution requires 3 steps only. From Fig. 2.2, it is clear that :

$$\left. \begin{array}{l} P_1 \parallel P_5 \\ P_2 \parallel P_3 \\ P_2 \parallel P_5 \\ P_5 \parallel P_3 \\ P_4 \parallel P_5 \end{array} \right\} \text{ or } P_2 \parallel P_3 \parallel P_5 \text{ is thus possible.}$$

And this is possible only if no resource conflicts are there.

2.2 TYPES OF DEPENDENCIES

To describe the relations between these dependencies, we usually draw **dependence graphs**. We use the following terms here :

1. **Nodes.** The nodes of a dependence graph correspond to the program statements/instructions.
2. **Directed Edges.** The directed edges show the ordered relations among the statements.

When we analyze these dependence graphs we can find out the areas where chances of parallelization exists.

There are three (3) main types of dependencies as discussed below.

1. Data Dependence
2. Control Dependence
3. Resource Dependence.

I. Data Dependence

It represents an ordering relation between statements. There are five (5) types of data dependencies. They are defined below :

(a) Flow dependence :

A statement S_2 is **flow-dependent** on statement S_1 , if an execution path exists from S_1 to S_2 . It is denoted as follows :

$$S_1 \longrightarrow S_2$$

Here, there should be atleast one output of S_1 that feeds S_2 as an input.

(b) Antidependence :

A statement S_2 is **antidependent** on statement S_1 , if S_2 follows S_1 in program order and if the output of S_2 overlaps the input to S_1 . It is denoted as follows:

$$S_1 \dashrightarrow S_2 \quad (\text{with a crossed bar})$$

(c) Output dependence :

Two statements are **output dependent** if they produce (or write) the same output variable. It is denoted by :

$$S_1 \circ \longrightarrow S_2$$

(d) I/O dependence :

It occurs because the **same file is referenced by both I/O statements**. It is not because the same variable is involved. Note that **read and write are I/O statements**.

$$S_1 \xrightarrow{I/O} S_2$$

(e) Unknown dependence :

There are some critical situations wherein dependencies cannot be determined. This is known as **unknown dependency**.

For examples :

1. Subscript of a variable is itself subscripted e.g. $a(IJ)$.
2. Subscript does not contain the loop index variable. e.g. $a[]$.
3. A variable appears twice with subscripts having different coefficients of the loop variable.
4. Subscript is nonlinear in the loop index variable.

When any of this situation exists, we say unknown dependencies exist. We are in a position to solve some problems now.

Example 2.2 Given the following four statements S1 – S4. Find out the data dependencies by drawing its data dependence graph :

S1 : Load R1, A

S2 : Add R2, R1

S3 : Move R1, R3

S4 : Store B, R1

Solution. We find that :

(a) S2 is **flow dependent** on S1 because the variable A is passed via register, R1.

So, $S1 \longrightarrow S2$

(b) S3 is **anti dependent** on S2 because of the conflicts in register contents of R1.

So, $S2 \dashv \longrightarrow S3$

(c) S3 is **output dependent** on S1 because they both modify the same register, R1.

So, we draw its dependence graph :

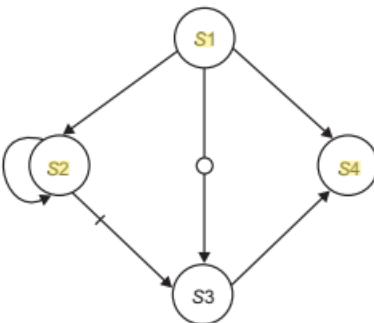


Fig. 2.3. Dependence Graph of S1 – S4

Note that it is not necessary that statements will always be related. For example, S2 and S4 in above graph of Figure 2.3 are totally independent.

Example 2.3 Consider another code snippet with some I/O operations :

S1 : Read (4), A(I)

S2 : Rewind (4)

S3 : Write (4), B(I)

S4 : Rewind (4).

Find dependencies and draw its dependence graph ?

Solution. We find that :

(a) S1 and S3 (read and write statements) are I/O dependent on each other because they both access the same file from the tape unit 4.

(b) No other dependency is found here. So, we draw its dependency graph now :

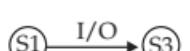


Fig. 2.4. Shows I/O dependence

II. Control Dependence

A situation where the order of execution of statements cannot be determined before run-time. For e.g., the *IF* statement of FORTRAN will not be resolved until run-time. Different branches of IF, may introduce or eliminate data dependencies among instructions.

Consider 2 Situations Now.

Situation 1 : Control independent loops	Situation 2 : Control dependent loops
<pre>Do 10 I = 1, N A (I) = C(I) If (A (I). LT . 0) A (I) = 1 10 CONTINUE</pre>	<pre>Do 20 I = 1, N If (A (I - 1). EQ . 0) A (I) = 0 20 CONTINUE</pre>

What I want to show from these two scenarios is that in situation II, wherein control dependency exists ; parallelism is not exploited. Better compilers are now needed to exploit more parallelism.

III. Resource Dependence

It means the conflicts that arise when using **shared resources** like integer units, floating-point units, registers and memory areas.

If the conflicting resource is an ALU then it is known as **ALU dependence**. If the conflicts involve workplace storage then it is called as **storage dependence**.

The transformation of a sequentially coded program into a parallel executable form is done either manually by the programmer using explicit parallelism or by compiler detecting implicit parallelism automatically. However, some programs are inherently sequential in nature. So they cannot be decomposed into parallel branches.

2.3 HARDWARE AND SOFTWARE PARALLELISM

To achieve our objective of parallelism, we need some special hardware and software support. We shall now tabulate the difference between the two.

Hardware Parallelism	Software Parallelism
<ol style="list-style-type: none"> 1. It is build into the machine's architecture and hardware multiplicity. It is also known as machine parallelism. 2. It is a function of cost and performance trade offs. 3. It displays what is the resource utilization. 	<ol style="list-style-type: none"> 1. It is exploited by the concurrent execution of machine language instructions in a program as coded by a programmer or as generated by a high-level language compiler. 2. It is a function of algorithm, programming style and compiler optimization. 3. It displays patterns of simultaneously executable operations.

2.4 PROGRAM PARTITIONING AND SCHEDULING

We need to understand the terminologies first. They are discussed below :

- (a) **Grain size or granularity.** It is a measure of the amount of computation involved in a Software Process.
- (b) **Grain.** It is a program segment that is chosen for parallel processing.
- (c) **Latency.** It is a time measure of the communication overhead incurred between machine subsystems.

For example : The time required by a PE to access the memory is known as the memory latency.

Now we shall discuss about the levels of parallelism.

2.4.1 Levels of Parallelism

Parallelism has been explored at five levels of processing. This is shown below in Figure 2.5.

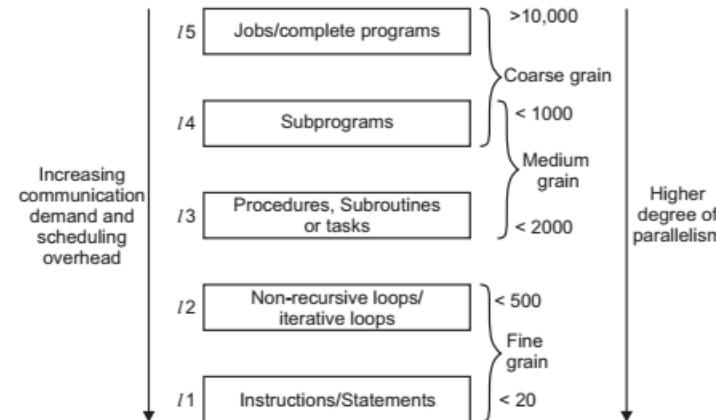


Fig. 2.5. Levels of Parallelism.

We now discuss these 5-levels one by one.

Level-1 : Instruction Level/Statement level parallelism

At this level of parallelism, a typical grain (or program's segment) consists of less than 20 instructions. So, it is called as **fine grain parallelism**. Please note that the lower the level (in Fig. 2.5), the finer will be the granularity of the software processes.

The advantage of this type of parallelism is the fact that at this level there will be huge amount of parallelism available. For doing so, we also need an **optimizing compiler**. It is a compiler which automatically detects parallelism. It converts the source code to a parallel form which can be understood by the run-time system. Note that detecting instruction level parallelism by an ordinary programmer in a source code is difficult.

Level-2 : Loop level parallelism

At the loop level, a grain size will have **less than 500 instructions**. (Whereas in level 1 above we had just 20). What we can do is that if some loop operations are non-recursive and independent in successive iterations then we can vectorize them for pipelined execution. Note that it is difficult to parallelize recursive loops. This level of parallelism is better and finer as compared to level-3 and level-5. This type of grain is very suitable for a parallel computer.

Level-3 : Procedure level parallelism

At this level, a typical grain corresponds to medium size at the task, procedural, subroutine or functional levels. The grain size is usually **less than 2000 instructions**. Please note that the detection of parallelism at this level is much more difficult than at the fine-grain levels. The programmers, however, need to workhard. They must restructure a program at this level. They need compilers also.

Level-4 : Subprogram level parallelism

At this level, a typical grain corresponds to subprograms or job steps. The grain size is **thousands of instructions**. Earlier, the parallelism at this level has been exploited by algorithm designers or programmers and not by the compilers.

Level-5 : Job/Program level parallelism

At this level, the grain corresponds to the independent jobs or programs for parallel execution. The grain size can be from **tens of thousands of instruction in a single program**. So, it is also known as **coarse grain level parallelism**. To achieve this level-5 or program level parallelism we need the program loader and an O.S. Time sharing systems explore this level of parallelism only.

Conclusions : We conclude with these points now.

1. **Fine grain parallelism** is exploited at instruction or loop levels. It requires a parallelizing or a vectorizing compiler.
2. **Medium grain parallelism** is exploited at the task or job step. It requires compilers as well as significant programmer roles.
3. **Coarse grain parallelism** is exploited at the program level. It requires effective O.S. and efficient algorithm. Message - passing multicomputers have been used for medium and coarse-grain computations.

Let us generalize this. We find that – **the finer the grain size, the higher is the potential for parallelism and the higher is the communication and scheduling overhead**. Please note that fine grain provides a higher degree of parallelism but more will be the communication overhead. This may not be true with coarse grain computations. Massive parallelism is often found at the fine grain levels like data parallelism on SIMD or MIMD computers.

What is grain packing ?

This approach was introduced by Lewis in 1988 for parallel programming. The **idea of grain packing** is to apply fine grain first in order to achieve a higher degree of parallelism. Then we combine (pack) multiple fine-grain nodes into a coarse-grain node if it can eliminate unnecessary communication delays or reduce scheduling overhead. Usually, all fine-grain operations within a single coarse-grain node are assigned to the same CPU for execution. Fine-grain partition of a program demands more inter processor communication than that required in a coarse-grain partition. Thus, grain packing offers a trade off between parallelism and communication overhead. Grain packing removes communication delays. Also note that the internal delays among fine grain operations within the same coarse-grain node are negligible. This is due to the fact that communication delay is contributed mainly by inter-PE delays and not by the delays within the same processor. Hence, the choice of optimal grain size is must. It is only then we can achieve the shortest schedule for the nodes on a parallel computer.

We are in a position to solve some problems now.

Example 1 : Consider the following program –**Begin**

1.a := 1 2.	b := 2
3.c := 3 4.	d := 4
5.e := 5 6.	f := 6
7.g := a * b	8. h := c * d
9.i := d * e	10. j := e * f
11.k := d * f	12. l := j * k

```

13.  $m := 4 * l$            14.  $n := 3 * m$ 
15.  $o := n * i$            16.  $p := o * h$ 
17.  $q := p * q$ 
End

```

Show fine grain program graph before packing. Convert it to the coarse grain program graph after packing.
[GGSIPU, M-Tech (IT) 6th sem., May 2003]

Solution. We draw the program graph. It shows the structure of a program. Each node in this graph corresponds to a computational unit in the program. The grain size is measured by the number of basic machine cycles – both processor and memory cycles that are needed to execute all the operations within the node.
How a node is represented ?

It is shown below :

where

- (n, s) denotes (node-number, grain-size)
- (x, i) denotes (input, delay)
- (z, k) denotes (output, delay)

and (y, j) and (g, h) are other inputs and outputs respectively.

So, the grain size reflects the number of computations involved in a program segment.
Fine-grain nodes have a smaller grain size and coarse-grain nodes have a larger grain size.

So, we draw its fine grain program graph first.

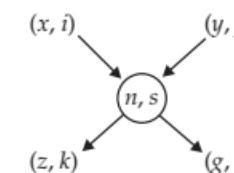
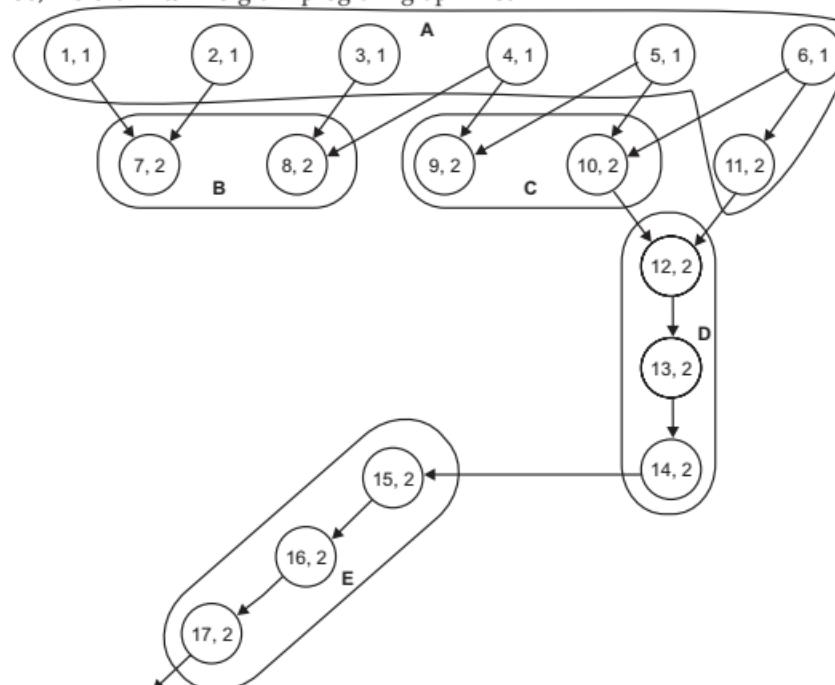


Fig. 2.6 (a). Fine grain parallelism

So, we convert it to a coarse-grain program graph now.

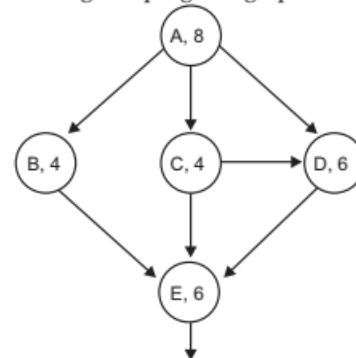


Fig. 2.6 (b). Coarse grain parallelism

In Figure – a, we find that nodes 1, 2, 3, 4, 5 and 6 are data fetch operations from/to memory that takes one cycle. All other nodes (7 to 17) are CPU operations, each requiring 2 cycles to complete.

In Figure – b, we do grain packing. This forms coarse grain nodes with larger grain sizes (from 4 to 8 as shown in Fig. b). Please note that the node (A, 8) in Fig. b, is obtained by combining the nodes (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1) and (11, 2) of Fig. a. And the grain size of node – A is the summation of all grain sizes of all these nodes i.e., $(1 + 1 + 1 + 1 + 1 + 1 + 2 = 8)$. So, we get (A, 8). Similarly, other packings are done.

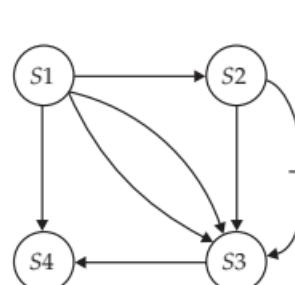
Example 2. Draw dependence graphs for the following programs :

- | | |
|--|---|
| (a) $S1 : A = B + D$
$S2 : C = A * 3$
$S3 : A = A + C$
$S4 : E = A/2$ | (b) $S1 : X = \sin(Y)$
$S2 : Z = X + W$
$S3 : Y = -2.5 * W$
$S4 : X = \cos(Z)$ |
|--|---|

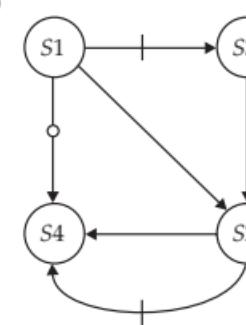
[GGSIPU, B-Tech (CSE) – 7th sem., Dec. 2008]

Solution.

(a)



(b)



Example 3. Consider the following FORTRAN code :

```

DO 10 I = 1, N
S1 : A(I + 1) = B(I - 1) + C(I)
S2 : B(I) = A(I) * K

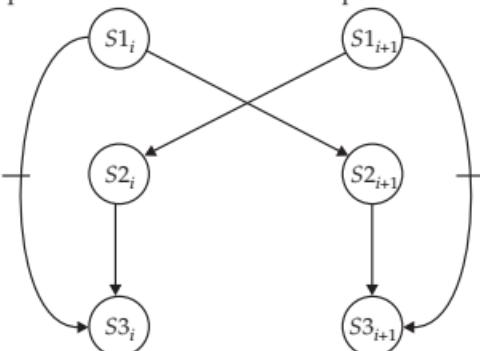
```

$$S \ 3 : C(I) = B(I) - I$$

10 continue

Determine the data dependencies in this program ?

Solution. The graph is shown below. The subscript is used to indicate the iteration:



Example 4. Analyze the data dependencies among the following statements in a given program:

$S1 : Load R1, 1024$

$S2 : Load R2, M(10)$

$S3 : Add R1, R2$

$S4 : Store M(1024), R1$

$S5 : Store M((R2)), 1024$

Where (R_i) means the context of register R_i and Memory (10) contains 64 initially.

- (a) Draw a dependence graph to show all the dependencies.
- (b) Are there any resource dependencies if only one copy of each functional unit is available in the CPU ?
- (c) Repeat the above for the following program statements :

$S1 : Load R1, M(100)$

$S2 : Move R2, R1$

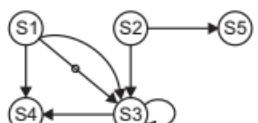
$S3 : Inc R1$

$S4 : Add R2, R1$

$S5 : Store M(100), R1$

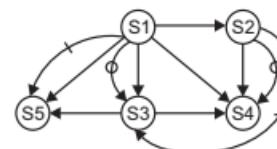
[UPTU, B. Tech (TT) 8th Sem.; 2007-08]

Solution. (a)



- (a) Yes, there are storage dependencies between the instruction pairs – $(S2, S5)$ and $(S4, S5)$. There is a resource dependence between $S1$ and $S2$ on the load unit and another between $S4$ and $S5$ on the store unit.

- (b) There is an ALU dependence between $S3$ and $S4$ and a storage dependence between $S1$ and $S5$.



2.5 PROGRAM FLOW MECHANISMS

Many parallel computers are built using the **conventional Von Neumann architecture** processors (PEs) as the building block. These parallel computers put the entire burden of parallelising a procedure on the programmer. A programmer has to understand the underlying architecture of the parallel computer to write efficient parallel programs. The systems do not attack the issue of parallelism at a fundamental level. **The data flow approach** to computing considers the issue of parallelism as its foundation. It proposes a new machine language which offers a powerful formalism for describing parallel computation at a fine grain level.

Reduction computers are based on a demand driven mechanism. It initiates an operation based on the demand for its results by other computations. We discuss each of these now.

2.6 CONTROL FLOW, DATA FLOW AND REDUCTION COMPUTERS

We have three categories of computers :

- (a) Control flow computers (or Von Neumann Computers)
- (b) Data flow Computers.
- (c) Reduction Computers.

We shall describe each of them one by one.

I. Control flow or Von Neumann's Conventional Computers

These computers use a **program counter** (PC) to sequence the execution of instructions in a program. These machines are also called as **control-driven machines** because the program flow is explicitly controlled by programmers. These computers use shared-memory to store program instructions as well as data. Instructions operate on variables in shared memory. Because the memory is shared so execution of one instruction may produce side effects on other instructions too. This is not preferable as it prohibits parallelism. These systems are uniprocessor system and are sequential machines *i.e.*, execution of instructions take place from left to right and top to bottom. However, note that these control flow machines can be made parallel by using parallel language constructs or parallel computers.

For example : 8085 CPU, 8086 CPU, Pentium-IV work on control flow mechanism.

II. Data Flow Computers/Data-driven Computers/Eager Machine

These machines are eager for data to be present. If it is present then the instruction is executed. These computers are thus known as **data driven computers or eager machines**. In a data driven program, the **instructions are not ordered**. There is **no shared memory**. Thus, data is stored inside instructions. The results or data tokens are passed directly between the instructions. Actually, a **copy of data** is passed to instructions. If an instruction has **consumed** these data token then they (tokens) are not available for reuse by other instructions. In these computers there is

□ No shared memory.

□ No Program Counter (PC)

□ No Control Sequencer.

Since there is no memory sharing so there are no side effects seen here. What these computers require is that they should be able to detect data's presence, to match the data tokens with the needy instructions and to enable the chain reaction of asynchronous instruction executions. We shall be discussing about these a bit later. Please note however, that a pure data flow computer exploits fine grain parallelism at the instruction level.

What is a data flow program graph ?

A machine language program of a data flow computer is represented by a data flow program graph. This graph is made up of operators connected by arcs. Arcs carry tokens which have values. An operator with two input arcs and one output arc is shown below in Fig. 2.6.

The operator of Fig. 2.6 is enabled only if tokens are present in both its input arcs and no token is present in its output arc. Tokens are placed on and removed from the arcs according to the firing rules. To explain the firing rule, let us say $a = 5$ and $b = 6$ in above Fig. 2.6. Then, we have :

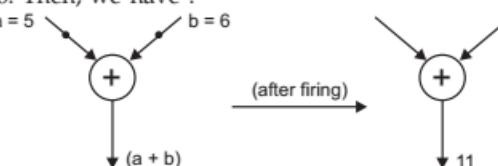


Fig. 2.7. Firing rules

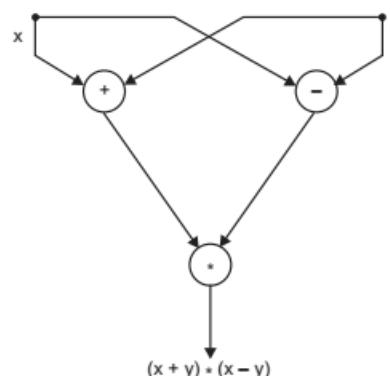
Please note that here, an operator fires, i.e., the tokens in the two input arcs are removed, the operation (+) is applied on the values carried by the tokens and a token with the result value is placed on the output arc. This is what exactly happens in this example.

We solve a problem now.

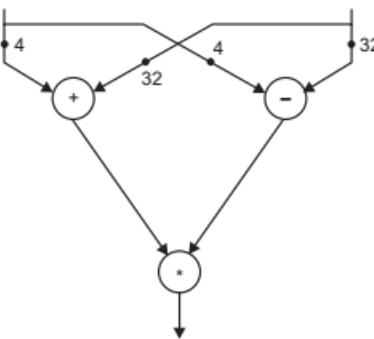
Example 1. Draw a data flow graph to compute the expression given below :

$(x + y) * (x - y)$?[GGSIPU, B-Tech (CSE) – 7th sem., First minor 2012]

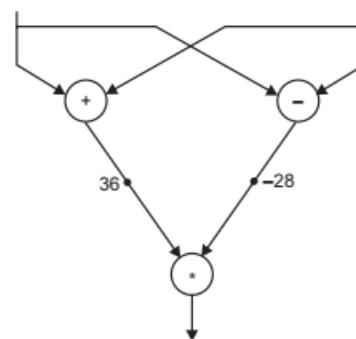
Solution. Here, x and y are two operands and there are 3 operators (+, -, *). We draw now:



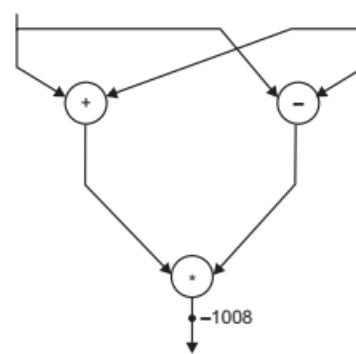
Now, say $x = 4$ and $y = 32$ appears on the input arc, then we draw :



Then, in the next stage we get :



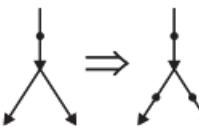
And, finally we get :



So, we find that the operators may be connected as shown above to form a data flow program graph. If the tokens with values for x and y are placed in the input arc, both operators $+$ and $-$ can fire simultaneously and place the result token on the respective output arcs. As soon as both the output tokens are generated, the $*$ operator fires giving the result (as -1008 in our example).

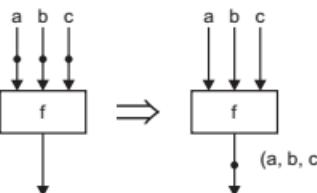
Other operators used in Data flow graph :

1. **Fork.** A fork looks like this :

**Fig. 2.8. Fork operation**

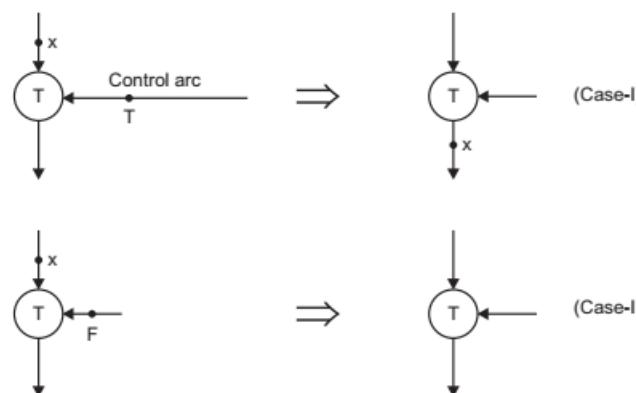
In this notation, as soon as a token arrives on the input arc it is copied in each of its output arcs; as shown in Fig. 2.8.

2. **Function.** Fig. 2.9. depicts a function.

**Fig. 2.9. Function operation**

Herein, after the tokens arrive on *all* the input arcs, this operator fires and the result token is placed on the output arc.

3. **True gate.** A true gate is shown in Fig. 2.10.

**Fig. 2.10. True gate**

Case – I. When a "true" token (T) arrives on the control arc of this graph and a token 'x' arrives on the input arc, the gate fires and 'x' is transmitted to the output arc.

Case – II. When a "false" token (F) arrives on the control arc and 'x' arrives on the input arc, then the gate fires with no token on the output arc. Both 'x' and F tokens on the input arc are destroyed.

Thus, we can draw a true gate operation table as follows :

Input arc	Control arc	Status of the operator	Output arc	Remarks
Nil	Nil	Not fired	Nil	Waiting for inputs
Nil	T	Not fired	Nil	Waiting for input
Nil	F	Not fired	Nil	Waiting for input
x	nil	Not fired	Nil	Waiting for control
x	T	fired	x	x and T consumed
x	F	fired	Nil	x and F consumed

4. *False gate.* A false gate is shown in Fig. 2.11.

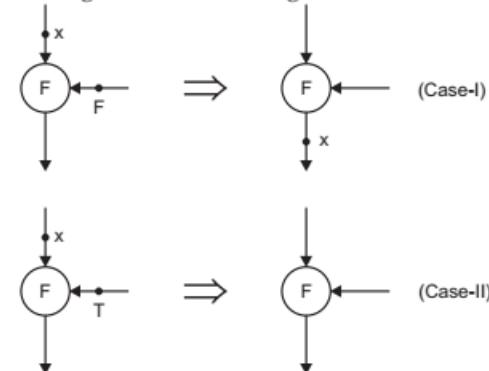


Fig. 2.11. False gate

A false gate shown above in Fig. 2. 11, works in a similar fashion. It fires when an input token and a control token both arrive.

Case – I. If the control token is F then the input token is transmitted to the output.

Case – II. If the control token is T then both input and control tokens are destroyed and no output token appears.

5. *Merge.* A merge operator is shown below in Fig. 2.12.

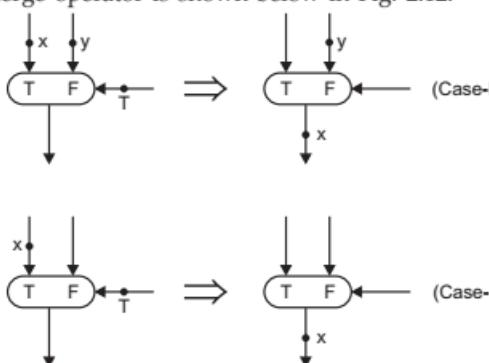


Fig. 2.12. Merge operator

It is seen in Fig. 2.12. that depending on whether the token on the control arc is T or F, the input token from the appropriate arc is placed on the output arc. The token on the input arc which is not selected is not destroyed. The firing rules of the merge-operator as listed below :

Input T arc	Input F arc	Control arc	Status of operation	Output arc	Remarks
Nil	Nil	Nil	Not fired	Nil	
Nil	Nil	T	Not fired	Nil	
Nil	Nil	F	Not fired	Nil	
Nil	y	T	Not fired	Nil	
Nil	y	F	fired	y	Both x and y are not destroyed
Nil	y	Nil	Not fired	Nil	
x	Nil	T	fired	x	
x	Nil	F	Not fired	Nil	
x	y	T	fired	x.	
x	y	F	fired	y	
x	Nil	Nil	Not fired	Nil	
x	y	Nil	Not fired	Nil	

6. *Predicate*. A predicate operator is shown below in Fig. 2.13. The values of the token in the two input arcs are compared by the predicate operator.

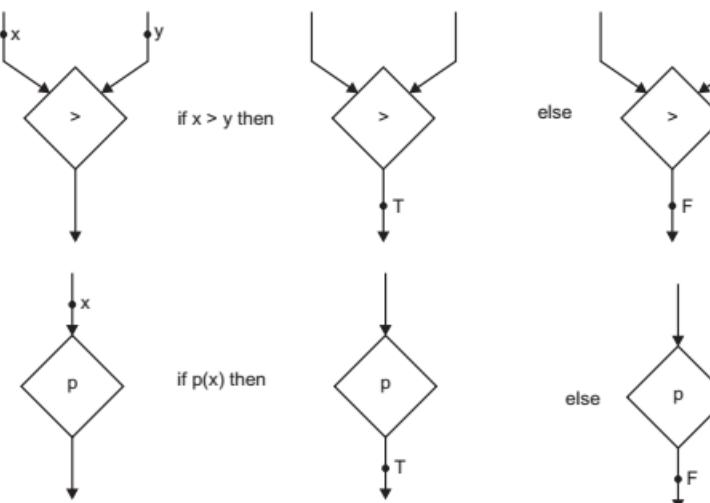


Fig. 2.13. Predicate operator

A T(true) or F(false) token is placed on the output arc based on whether the predicate succeeds or not.

7. **Switch.** A switch is not a new operator. It is a combination of T and F gates as shown below in Fig. 2.14.

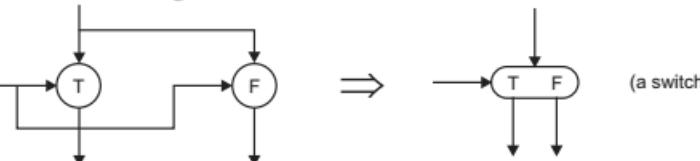
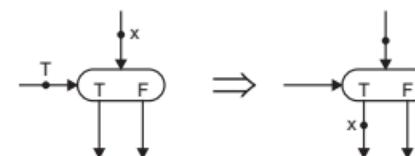


Fig. 2.14. Switch operator

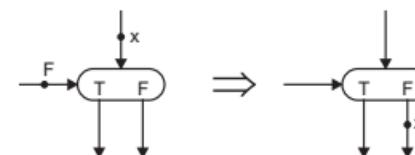
The notation in concise is shown as a switch. How it works ?

Case – I.



When x appears on the input arc and the control arc receives T then x appears on the T branch of the switch as an output.

Case – II.



Similarly, when x appears on the input arc and F appears on the control arc then x appears on an output arc.

Note here that switch is not a new operator. It is actually a combination of T and F gates.

We are now in a position to solve some problems now.

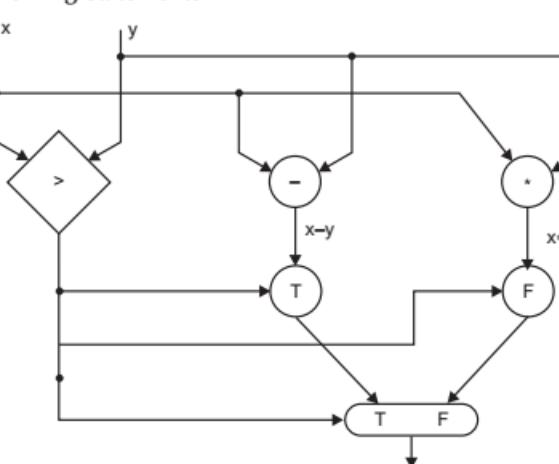
Example 1. Consider the following statements :

```
if x > y then (x - y)
else (x * y)
endif
```

Draw its data flow graph ?

Solution.

Herein, we have used predicate, merge and switch operators to implement the given statement.

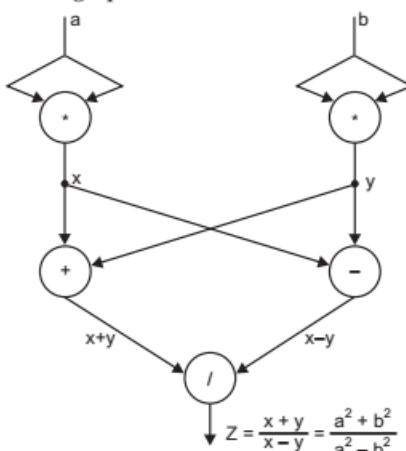


Example 2. Draw a data flow graph to compute the following expression :
 $(a^2 + b^2) / (a^2 - b^2)$?

Solution. Say,

$$\begin{aligned}x &= a * a \\y &= b * b \\ \text{Then } z &= \frac{x+y}{x-y} = \frac{a^2+b^2}{a^2-b^2}\end{aligned}$$

So, we draw its data flow graph now :



Note that the first two instructions can be carried out in parallel. That is why, we have drawn such a data flow graph.

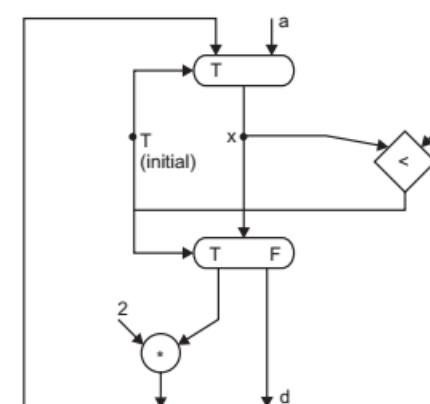
Example 3. Consider the following while loop :

```

x = a
while (x < y)
{
    x = 2 * x;
} // while ends
  
```

Draw its data flow graph. Assume that the control input has T (initially).

Solution. In this example, we need a merge operator and that it is given that the control input has T initially. So, we draw:

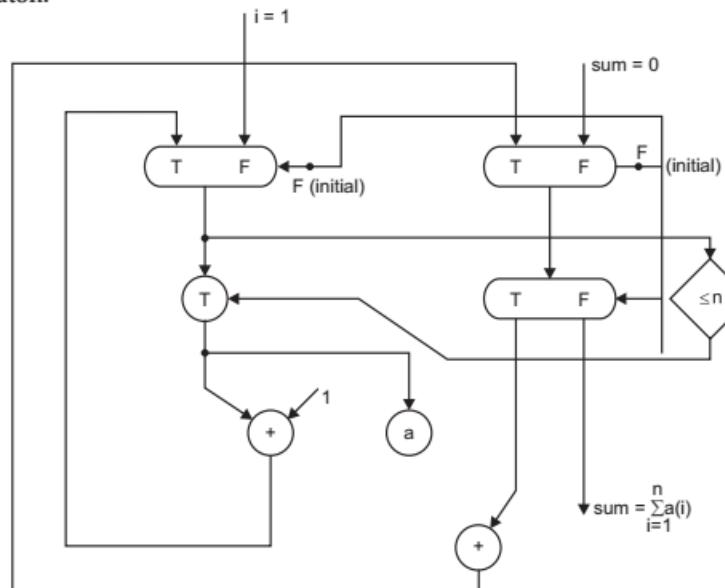


Example 4. It is desired to store the sum of all array elements, $a[i]$ for $i \leftarrow 1$ to n . That is,

$$\text{Sum} = \sum_{i=1}^n a(i)$$

Draw its data flow graph. Also explain your results.

Soluton.



In this flow graph, an initial token, F is supplied to the two merge gates. The initial values of i and sum enter the loop via the F input of the merge gates. If the predicate evaluates to T then i and sum are routed to the loop body. The values circulate in the loop till the predicate evaluates to F . When this happens the result comes out of the loop.

Please note that the possibility of parallel execution will be exploited by the way in which the hardware is designed to execute the data flow graph.

What is a data flow computer architecture ?

A data flow computer's computing element (CE) is shown below in Fig. 2.15.

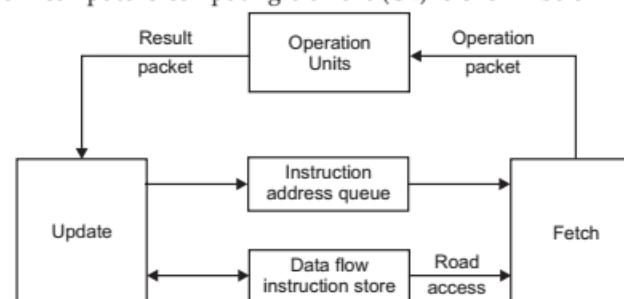


Fig. 2.15. A data flow computer's computing element (CE)

How it works ? The instruction store in this computer stores the machine instructions corresponding to the data flow graph. When an instruction is ready for execution, its address is sent to the instruction address queue. The fetch unit (as shown) fetches the address of the first instruction waiting in this queue. Using this address it retrieves a data flow instruction from the instruction store.

The operation code, the operands and the addresses where the results are to be sent are formed into an **operation packet** and then sent to the **operation execution units**. There can be multiple units also that can work together. The results so obtained after execution are formed into a result packet.

Note : Result packet = result + address where
the results
should go
+
addresses where
acknowledgements
should go.

The result packets are sent to the update unit. This unit enters these results in the appropriate operand slots in the instructions in the instruction store. Note that the update unit will also forward acknowledgements as specified by the result packet. An instruction is ready for its execution when the count of input operands and acknowledgements received by it match the number specified in the instruction.

Note : When the program is first invoked, all arcs are assumed to be empty. Thus, no acknowledgements are needed. This is implemented by initialising the number of acknowledgements to equal the number of destination addresses.

The update unit finds the instructions ready for execution in the instruction store and puts their addresses in the instruction queue. Please note that during a data flow program execution, the number of instruction addresses waiting in the instruction queue is a measure of parallelism present in the program. Besides this, the fetch unit, the operation unit and the update unit can all be simultaneously working. This mechanism is a circular pipeline with different packets flowing in different parts. The **degree of concurrency** is limited by the number of operation units in the ring and the degree of pipelining in fetch and update units. However, the **level of concurrency** is limited by the parallelism within the program and the capacity of the data paths connecting the units in the ring.

A Data flow multi computing System

Many data flow computing elements (DFCE or CE) can be used to built a data flow multi computing system. It is shown below in Fig. 2.16. Each DFCE is connected to a communication system.

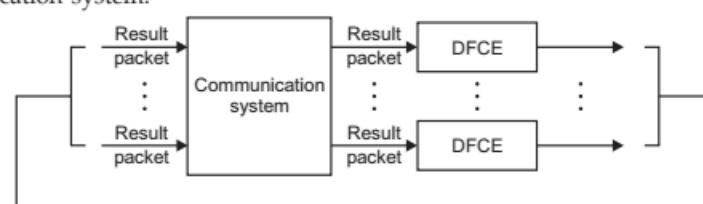


Fig. 2.16. Data flow multi computer

What is actually done ? A data flow program is divided into many parts. Each part goes to one DFCE i.e., only 1 part/DFCE. The instruction stores of DFCEs collectively realise a single large global address space. The destination address field of an instruction may select any of the instruction stores in the system. If the destination address of the result is in the local instruction store then it is locally forwarded within the DFCE. Else it is routed by the communication system to the DFCE whose instruction store has that address. Such a computer is called as a **static data flow computer**. As given by Dennis, this computer allows only one token in an arc at a time. Also, no new tokens can be accepted by an operator until the result is consumed by the receiving operators and acknowledgements to this effect are received by the operator. If this is not ensured then the results could be incorrect.

There is another type of data flow multi computer. It is known as a **tagged token data flow computer or dynamic computer**. This model was given by Arvind and Gurd. In this model, more than one token is allowed in an arc. This will exploit more parallelism. Each token on an arc is given a *tag* which maintains the logical order in which tokens appear on an arc. The i^{th} token to flow along an arc carries a tag, i . An operator fires when all its input arcs have tokens whose tags are identical. No acknowledgement is required as each token carries its own unique tag. A token has this form :

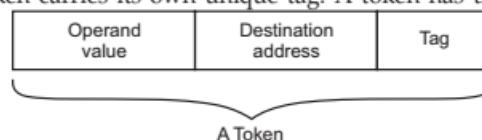


Fig. 2.17. Token format

Please note that the tagged data flow model allows concurrent computation of many iterations of a loop. This is so because the iteration number appears as a tag. Each iteration of a loop creates a new result which has an independent existence. A tagged token DFC is shown below in Fig. 2. 18.

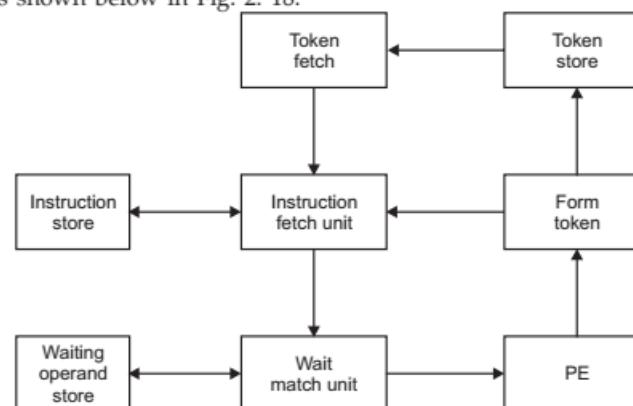


Fig. 2.18. Tagged token DFC

This model works without acknowledgement part in the instruction. When a token arrives for an instruction which requires a single input operand, the instruction can be executed immediately. When the first token containing one of the operands arrives for an instruction requiring 2 input operands, it must wait until its correct partner arrives. So, a waiting operand store is provided. This shows working of a single DFCE. We have

many DFCEs connected together using a communication medium. A large data flow program is partitioned and stored in individual instruction memories of the DFCEs. There is a global addressing of instructions as well as data.

The statements like $x \leftarrow x * 10$ are not meaningful here. There is no sense in updating a variable with a new value if it is a DFC. This creates a problem when operations on some parts of an arrays are to be performed by many DFCEs, in which case, arrays have to be duplicated. For this reason, a synchronization mechanism called as **I-structure** is provided within each PE. Herein, arrays are **centrally stored**. Only one write operation is permitted in the structure. It can be read and written independently. It is a **tagged memory unit** (2-bit tag) for overlapped usage of a data structure by both producer and consumer processes. Note that the I-structure provides the hardware aid to assist multiple computers to independently read/write into a common store without causing read/write races.

Please note that a data flow graph is the machine language of a DFC. A HLL is needed for this machine which will use the implicit parallelism in the algorithm. The programmer is thus free to not to use statements like fork, join, send and receive. It should be independent of the machine specifications such as number of PEs, memory hierarchy etc.

Some languages which have been designed with clear rules for syntax and semantics allow a user to write programs easily. For e.g. *Id* and *VAL* languages. These languages have compilers to translate a program to a data flow graph and thus utilizing parallelism to its maximum. It 'unfolds' loops and enables multiple simultaneous execution of loops. Such languages are known as **single assignment functional programming languages**.

III. Demand – Driven Computers/Lazy Computers/Reduction Machines

In these machines, if there is a demand for the result then only the computation is triggered. They use top down approach as compared to the bottom-up approach as used in previously described machines. It is also known as a **lazy machine** because the operations are executed only when their results are required by another instruction. In these machines, only required instructions are executed. There is a high degree of parallelism.

2.7 COMPARISON OF CONTROL FLOW, DATA FLOW AND DEMAND DRIVEN COMPUTERS

We now compare the three approaches discussed above in a tabular form. This also gives us an idea of what are the advantages, disadvantages and the applications of such computers. We also know that the control-flow machines are control-driven, data flow machines are data driven and Reduction machines are demand driven. So, we draw a table :

Control driven or Control Flow	Data driven or Data Flow	Demand driven or Reduction
1. These are basically conventional Von Neumann's machines computations i.e., token of control indicates when a statement should be executed.	1. They involve Eager evaluation i.e., statements are executed when all operands are available.	1. They involve lazy evaluation i.e., statements are executed only when their result is required for another computation.

2. Programmer has full control – an advantage.	2. It provides very high potential for parallelism – an advantage.	2. Only required instructions are executed – an advantage.
3. Complex data and control structure are easily implemented.	3. High through put is seen here.	3. High degree of parallelism.
4. Less efficient machines.	4. Time lost waiting for un-needed arguments.	4. No sharing of objects.
5. Programming is not easy.	5. High control overhead.	5. Time is needed to propagate demand tokens.

SUMMARY

The Bernstein conditions as studied in this chapter are completely general. They can be used to identify instruction level parallelism or coarser parallelism where a set of routines is being considered for concurrent operation. In that case, the inputs are the parameters to the routines and the outputs are the variables/values returned. Also in this chapter, we find that an interesting feature of DFC is the absence of PC to sequence instruction. There are quite few experimental data flow computer projects. Arvind and his associates at MIT have developed a tagged-token DFC.

MULTIPLE CHOICE QUESTIONS (MCQS) WITH ANSWERS

8. Pentium-IV works on :
- Control flow mechanism
 - Data flow mechanism
 - Demand driven mechanism
 - All of the above.
9. Static data flow computer (SDFC) was given by :
- Dennis
 - Bernstein
 - Albrecht
 - None of the above.

:: ANSWERS ::

- | | | | | |
|--------|--------|--------|--------|--------|
| 1. (b) | 2. (a) | 3. (b) | 4. (a) | 5. (a) |
| 6. (c) | 7. (c) | 8. (a) | 9. (a) | |

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

1. Consider the following C statements :

$$\begin{aligned} a &= x + y ; \\ b &= x + z ; \end{aligned}$$

Do they satisfy Bernstein conditions ? Furthermore, if $a = x + y$; and $b = a + b$; then will they satisfy the same ?

Ans. Consider the first two statements :

$$\begin{aligned} a &= x + y ; \\ b &= x + z ; \\ \text{Now, } &I_1 = (x, y) \\ &I_2 = (x, z) \\ &O_1 = (a) \\ &O_2 = (b) \\ \therefore &\left. \begin{aligned} I_1 \cap O_2 &= \emptyset \\ I_2 \cap O_1 &= \emptyset \\ O_1 \cap O_2 &= \emptyset \end{aligned} \right\} \end{aligned}$$

are satisfied. Hence, these two statements can be executed in parallel.

$$\begin{aligned} \text{Now, say } &a = x + y ; \\ &b = a + b ; \\ \therefore &I_2 \cap O_1 \neq \emptyset \end{aligned}$$

\therefore These two statements cannot be executed concurrently.

2. What is a communication latency ?

Ans. In general, n tasks communicating with each other require $n(n - 1)/2$ communication links among them. So, the complexity grows quadratically. This puts an upper bound (limit) on the number of PEs that can be used in a computer.

3. What is the difference between the machine instruction of a conventional Von Neumann's machine to that of a data flow computer (DFC) ?

Ans. The machine instruction format of a Von Neumann computer is as follows :

Operation to perform	Address in main memory of operands	Address in the main memory of the result	Address of the next instruction to be carried out (PC)
----------------------	------------------------------------	--	--

Whereas the instruction format of a DFC is as follows :

Operation to perform	Slot for operand from left arc	Slot of operand from right arc	Address of the machine instruction and address of slots in which the result is to be stored.

We see that the operands in data flow machines are directly forwarded to the instructions which need them rather than being stored in memory as is done in Von Neumann computers. There is no program counter (PC) to give the address of the next instruction to be executed as in Von-Neumann computer. A data flow instruction is activated as soon as all its input operand slots are filled and can fire if the input operand slots of the instruction (s) to which the result is to be sent are empty. The operand slots of the receiving instruction will be free if it has used up whatever operands came to it earlier. Thus, as soon as a receiving instruction consumes its input operands, it sends an acknowledgement to all the sending instructions. An enabled instruction fires after it receives acknowledgements from all of the result destination addresses.

4. Consider the following sequential code in FORTRAN language :

```

S1 :      A = B + C
S2 :      C = B * D
S3 :      S = 0
S4 : DO I = A, 100
          S = S + X (I)
      END DO
S5 :      IF (S . GT . 1000) C = C * 2

```

Will this satisfy Bernstein's conditions for parallelism ? Restructure the program in order to maximize parallelism? [GGSIPU, M-Tech (IT) – 1st sem., Jan. 2011]

Ans. Firstly, we form the input and output sets for the instructions given :

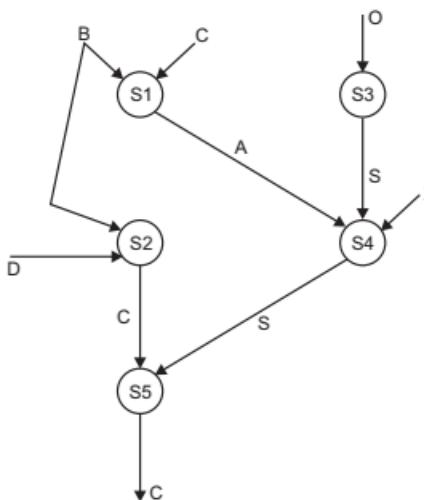
$$\begin{array}{ll}
 I_1 = \{B, C\} & O_1 = \{A\} \\
 I_2 = \{B, D\} & O_2 = \{C\} \\
 I_3 = \emptyset & O_3 = \{S\} \\
 I_4 = \{S, A, X (I)\} & O_4 = \{S\} \\
 I_5 = \{S, C\} & O_5 = \{C\}
 \end{array}$$

Using Bernstein's conditions, we find that :

- (a) $I_1 \cap O_3 = \emptyset$ $I_3 \cap O_1 = \emptyset$
 and $O_1 \cap O_3 = \emptyset$
 $\Rightarrow S1$ and $S3$ can be executed concurrently.
- (b) $I_2 \cap O_3 = \emptyset$ $I_3 \cap O_2 = \emptyset$
 and $O_2 \cap O_3 = \emptyset$
 $\Rightarrow S2$ and $S3$ can be executed concurrently.

- (c) $I_2 \cap O_4 = \emptyset$ $I_4 \cap O_2 = \emptyset$
 and $O_2 \cap O_4 = \emptyset$
 $\Rightarrow S2$ and $S4$ can be executed concurrently.
- (d) $I_1 \cap O_5 = \{C\}$
 $\Rightarrow S1$ and $S5$ can not be executed concurrently.
- (e) $I_1 \cap O_2 = \{C\}$
 $\Rightarrow S1$ and $S5$ can not be executed concurrently.
- (f) $I_4 \cap O_1 = \{A\}$
 $\Rightarrow S1$ and $S4$ can not be executed concurrently.
- (g) $I_5 \cap O_2 = O_5 \cap O_2 = \{C\}$
 $\Rightarrow S2$ and $S5$ can not be executed concurrently.
- (h) $I_4 \cap O_3 = O_4 \cap O_3 = \{S\}$
 $\Rightarrow S3$ and $S4$ can not be executed concurrently.
- (i) $I_5 \cap O_3 = \{S\}$
 $\Rightarrow S3$ and $S5$ can not be executed concurrently.
- (j) $I_5 \cap I_4 = I_5 \cap O_4 = \{S\}$
 $\Rightarrow S4$ and $S5$ can not be executed concurrently.

We can reconstruct the program to draw the flow graph as shown below :



5. Consider the execution of the following code segment consisting of 7 statements. Use Bernstein's conditions to detect the maximum parallelism embedded in this code. Justify the portions that can be executed in parallel and the remaining portions that must be executed sequentially. Rewrite the code using parallel constructs such as co-begin and co-end. No variable substitution is allowed.

$S1 : A = B + C$
 $S2 : C = D + E$
 $S3 : F = G + E$
 $S4 : C = A + F$
 $S5 : M = G + C$
 $S6 : A = L + C$
 $S7 : A = E + A$

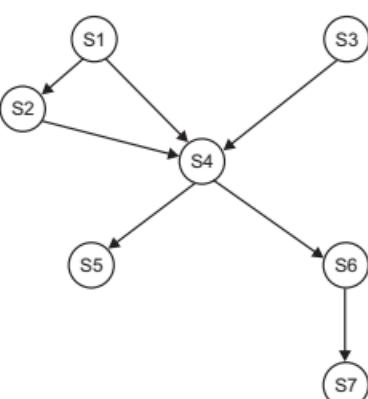
Ans. The input and output sets for the instructions are as follows :

$I_1 = \{B, C\}$	$O_1 = \{A\}$
$I_2 = \{D, E\}$	$O_2 = \{C\}$
$I_3 = \{G, E\}$	$O_3 = \{F\}$
$I_4 = \{A, F\}$	$O_4 = \{C\}$
$I_5 = \{G, C\}$	$O_5 = \{M\}$
$I_6 = \{L, C\}$	$O_6 = \{A\}$
$I_7 = \{E, A\}$	$O_7 = \{A\}$

Using Bernstein's conditions we find that

$S1 \parallel S3$	$S3 \parallel S5$
$S1 \parallel S5$	$S3 \parallel S6$
$S2 \parallel S3$	$S3 \parallel S7$
$S2 \parallel S7$	$S5 \parallel S6$
$S5 \parallel S7$	

However, Bernstein's conditions are not sufficient for this problem. The precedence relations as shown below have to be taken into account :



It is clear that $S1, S2$ and $S3$ must be executed before $S4$. Moreover, $S5, S6$ and $S7$ must be executed after $S4$. This consideration prohibits parallel execution among the two groups of statements. Thus, the statements that can be executed in parallel are : $S1 \parallel S3, S2 \parallel S3, S5 \parallel S6$ and $S5 \parallel S7$.

The parallel code is given below :

```
Co begin  
    S1, S3  
Co end  
    S2  
    S4  
Co begin  
    S5, S6  
Co end  
    S7
```

6. Define the following terms :

- | | |
|-------------------------------|---------------------------|
| (a) Computational granularity | (b) Communication latency |
| (c) Flow dependence | (d) Anti dependence |
| (e) Output dependence | (f) I/O dependence |
| (g) Control dependence | (h) Resource dependence |
| (i) Bernstein conditions | (j) Degree of parallelism |

- Ans.** (a) The amount of computation involved in a software process.
(b) The time measure of the communication overhead incurred between machine subsystems.
(c) A statement S_2 is flow-dependent on statement S_1 if an execution path exists from S_1 to S_2 and atleast one output of S_1 feeds in as input to S_2 .
(d) A statement S_2 is antidependent on statement S_1 if S_2 follows S_1 in program order and the ouput of S_2 overlaps the input to S_1 .
(e) Two statements are output dependent if they produce the same variable.
(f) I/O dependence occurs when the same file or I/O device is requested by two statements.
(g) This refers to the situation where the order of execution of statements cannot be determined before run-time.
(h) Resource dependence is concerning the conflicts in using shared resources such as integer units, floating-point units, registers and memory areas among the parallel events.
(i) Bernstein's condition states that two processes can execute in parallel if their input sets I_1, I_2 and the output sets O_1, O_2 satisfy the following conditions :

$$\begin{aligned}I_1 \cap O_2 &= \emptyset \\I_2 \cap O_1 &= \emptyset \\O_1 \cap O_2 &= \emptyset\end{aligned}$$

- (j) For each time period, the number of processors used to execute a parallel program is defined as the degree of parallelism.
7. (a) Compare control flow, data flow and reduction computers in terms of the program flow mechanism used.
(b) What are the differences between string reduction and graph reduction machines?

- Ans.** (a) A control flow computer use Program Counter (PC) to sequence the execution of instructions in a program. In a data flow computer, the execution of

an instruction is driven by data availability. In a **reduction computer**, the computation is triggered by the demand for an operation's result.

- (b) In string reduction, each demander gets a separate copy of the expression for its own evaluation. In graph reduction, each demander is given a pointer to the result of the reduction.

8. Consider the following code :

```
input d, e, f
C0 = 0
for i from 1 to 8 do
begin
    ai := di + ei
    bi := ai * fi
    ci := bi + Ci-1
end
output a, b, c
```

Find out how many cycles are required to run this code on :

- (a) Sequential (uniprocessor) machine.
(b) Data flow computer (DFC) ?

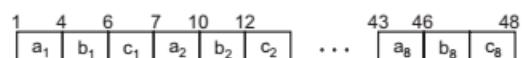
Ans. (a) On a sequential machine.

when $i = 1$, in above code then

$$\left. \begin{array}{l} a_1 = d_1 + e_1 \\ b_1 = a_1 * f_1 \\ c_1 = b_1 + c_0 \end{array} \right\}$$

and so on; till $i = 8$.

On a sequential machine the execution is as follows :



\therefore 24 instructions are there as $3 * 8 = 24$

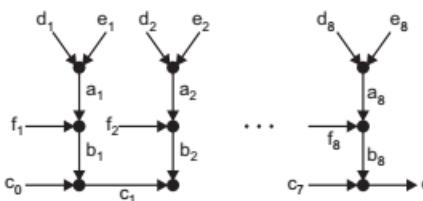
Each add, multiply and divide takes 1, 2 and 3 cycles respectively (say).

\therefore Sequential machine takes 48 cycles to execute these 24 instructions.

(b) On a DFC (data flow computer)

On a data driven computer, the scenario is different. That is ,

1	4	7	8	9	10	11	12	13	14
a ₁	a ₅	c ₁	c ₂	c ₃	c ₄	c ₅	c ₆	c ₇	c ₈
a ₂	b ₁	b ₂	b ₄	b ₆	b ₈				
a ₃	a ₆	b ₃	b ₅	b ₇					
a ₄	a ₇	a ₈							



i.e., if a_1 and a_5 are available then all c_1 to c_8 results are available.

So, a DFC will take only 14 cycles.

9. Write a note on – Role of compilers in parallel computing ?

Ans. We list the roles of compilers :

1. Compilers are used to exploit hardware features to improve performance.
2. CDC STACKLIB, Cray CFT, Rice PFC and Yale Bulldog are some of its examples.
3. The general guideline is to increase the flexibility in hardware parallelism and to exploit software parallelism.
4. Hardware and software design trade offs exist in terms of cost, complexity, expandability, compatibility and performance.
5. Multi processor's compilation is much more involved than for uniprocessors.
6. Granularity and communication latency play important roles in code optimizations.

EXERCISE QUESTIONS

- 1.** Use Bernstein's conditions to detect the parallelism embedded in this code :

$$P1 : C = D * E$$

$$P2 : M = G + C$$

$$P3 : A = B + C$$

$$P4 : C = L + M$$

$$P5 : F = G \div E$$

Also draw a dependence graph showing both data dependence and resource dependence. [GGSIPU, B.Tech. 7th sem. Dec. 2007]

- 2.** Answer the following questions related to data flow computers :

(a) Distinction between static and dynamic data flow computers.

(b) Draw a data flow graph showing the computations of the roots of quadratic equation :

$$A_i x^2 + B_i x + c_i = 0 \quad (i = 1, 2, \dots N)$$

[GGSIPU, M-Tech. (IT) ; 6th sem. May. 2003]

- 3.** What is non-Von Neumann architecture computer ? Compare Von Neumann architecture vis-a-vis data flow architecture computer. Which architecture is more suitable for parallel algorithm and why ?

[GGSIPU, M.Tech (IT) 6th sem. May. 2003]

- 4.** Explain Bernstein's conditions for parallelism.

[GGSIPU, M.Tech (IT) 1st sem. Dec. 2004]

- 5.** (a) Discuss the concept of grain packing and scheduling with the help of an example.

(b) Compare control flow and data flow computers.

[GGSIPU, M.Tech (IT) 1st sem. Dec. 2004]

- 6.** List the special features of data flow computers and differentiate between static and dynamic data flow computers. [UPTU, B. Tech (CSE) 8th Sem.; 2006-07 &

KUD, BE (CSE) 8th sem. 1996]

7. List the different operators used in data flow graphs along with their usage.
Draw data flow graphs to compute factorial of a positive integer, N .

[KUD, BE (CSE) 8th sem. 1996]

8. (a) Describe the following :
(i)Data driven computation.
(ii)Event driven computation.
(iii)Static data flow graph.
(iv)Dynamic data flow graph.
(b) Draw data flow graphs to represent the following computations :
(i)if (($a == b$) & ($c < d$)) then $c = c - a$ else $c = c + a$
(ii)for ($i = 0$; $i \geq 10$; $i++$)
{
 cout << i ;
}

[KUD, BE (CSE) 8th sem. 1997]

9. Write short notes on :
classification of parallel computers based on :
(a) Mode of Accessing Memory.
(b) Grain Size. [DU, M.E. (CSE) ; DCE 2003]
10. Analyze the data dependencies for the following program. Draw a dependency graph to show all dependencies :
Load R1, M (100)
Move R2, R1
Inc R1
Add R2, R1
Store M (100), R1 ? [Pune University ; B.E. (CSE). Dec. 2001]

11. Consider the following loopnest:
DO I = 1, N
DO J = 2, n
 A (I, J) = B (I - J) + C (1, J)
 C (I, J) = D (I, J) / 2
 E (I, J) = A (I, J - 1) ** 2 + E (1, J - 1)
END DO
END DO [UPTU, B. Tech (CSE) 8th Sem.; 2003-04]

12. Distinguish between Medium grain and fine grain multicomputer in their architecture. [UPTU, B. Tech (CSE) 8th Sem.; 2007-08]
13. Differentiate between Hardware and Software parallelism. [GGSIPU, B. Tech (CSE) 7th Sem.; Dec. 2011]

CHAPTER

3 SYSTEM INTERCONNECT ARCHITECTURES

3.0 INTRODUCTION

There are static and dynamic networks for connecting multiprocessors and multicomputers. These networks can be used for internal connection among processors, memory modules and I/O disk arrays in a centralized system or for distributed networking of the multicomputer nodes.

3.1 NETWORK PROPERTIES

Every network is measured for its effectiveness using some of its properties. These properties are :

1. **Network Size.** The number of nodes in the graph is called the network size.
2. **Graph.** A graph of a network consists of *nodes* that represent switching points and *edges* that represent communication links.
3. **Node degree (d).** The number of edges or links or channels that are incident on a node is called the node degree.

For example, for a unidirectional channel, the number of channels into a node is the **indegree** and that out of a node is **out degree**.

Then we say that :

$$\text{Node-degree } (d) = \text{indegree} + \text{outdegree}$$

' d ' reflects the number of I/O ports required per node.

Also note that ' d ' should be kept as a constant or as small as possible in order to reduce cost and to achieve modularity.

4. **Diameter (D).** The diameter of a network is the largest distance between two nodes. Please note that a low diameter is better because the diameter puts a lower bound on the complexity of parallel algorithms requiring communication between arbitrary pair of nodes.
5. **Bisection width (b).** It is defined as the minimum number of edges that must be removed in order to divide the network into two halves. Please note that the higher bisection width is better because in algorithms requiring large amounts of data movement, the size of data set divided by the bisection width (b) puts a *lower bound* on the complexity of parallel algorithm.
6. **Number of edges per node.** A node (or vertex) represents a PE and an edge represents communication paths between two PEs, So, it represents the number of communication paths per processor (or PE). Note that it should be constant as then the PE organization scales more easily to systems with large numbers of nodes.

7. **Maximum edge (path) length.** It is best if the nodes (PE) and edges (path) represent a 3D space network so that the maximum edge length is a constant independent of the network size.

3.2 ROUTING

A data-routing network is used for inter-PE data communications. There are data routing functions (DRF) that are implemented on an inter-PE routing network.

That is,

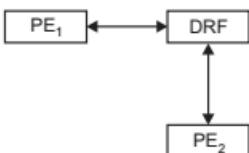


Fig. 3.1. Showing role of DRF.

Some of the common DRFs are given below :

- | | |
|---------------------------------|----------------------------------|
| 1. Permutations. | 2. Perfect Shuffle and Exchange. |
| 3. Hypercube Routing Functions. | 4. Broadcast and Multicast. |

We explain all of these one by one.

1. Permutations : If we have t objects then there are $t!$ permutations by which t objects can be reordered. We use a cycle notation to specify a permutation function.

For example : The permutation,

$\pi = (a, b, c) (d, e)$ stands for $a \rightarrow b, b \rightarrow c, c \rightarrow a, d \rightarrow e$ and $e \rightarrow d$ in a circular fashion. Also note that the cycle (a, b, c) has a period of 3 and the other cycle (d, e) has a period of 2. So, the total permutation (π) has a period of $3 * 2 = 6$. Now, if we apply the permutation (π) six times then this many times data routing can be performed.

Permutations can be implemented by using a crossbar switch (to be discussed shortly), a multistage network, shifting or broadcast operations.

2. Perfect shuffle and Exchange : In 1971, Harold Stone suggested that if we are given a binary string –as $(a_n a_{n-1} \dots a_2 a_1)$ then

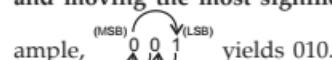
Shuffle $(a_n a_{n-1} \dots a_2 a_1) = (a_{n-1} a_{n-2} \dots a_2 a_1 a_n)$

For example :

Shuffle $(001) = 010$

Shuffle $(101) = 011$

That is, the shuffle transformation means left shifting our binary string by 1 bit and moving the most significant bit at the least significant position. Like in first example,

 yields 010.

3. Hypercube Routing Functions : We consider a 3D-binary cube as shown below in Fig. 3.2.

Say, the node address is 3 bits long (C_2, C_1, C_0) in binary. Since there are 3 bits in the node address so we can have three routing functions. These three routing functions will be on the basis of the routing of the three bits *i.e.*, either C_0 or C_1 or C_2 .

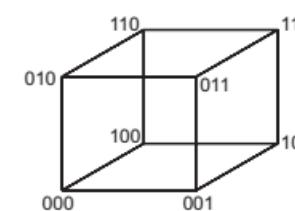


Fig 3.2. 3D binary cube

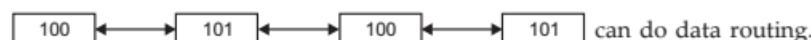
Case – I : Routing on C_0 (LSB) :

Now, the routing between any 2 nodes of Fig. 3.2 is possible only if the C_0 (LSB) bit position of these nodes differ. i.e.,



these two nodes can do routing. Also can do routing.

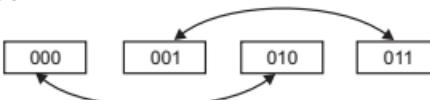
Similarly, can do routing and



Please note that 000 and 010 can not route here since their LSBs are same.

Case – II : Routing on C_1 (middle bit)

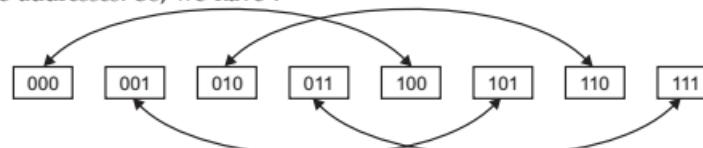
This type of routing is possible if and only if middle bit (C_1) differs on their node addresses. So, we have :



Similarly, other combination can be formed.

Case – III : Routing on C_2 (MSB – rightmost bit)

This type of routing is possible if and only if the right most/MSB bit (C_2) differs on their node addresses. So, we have :



Similarly, other combinations can be formed.

Please note that in general, an n -dimensional hypercube will have n routing functions.

IV. Broadcast and Multicast : Broadcast means one-to-all and multicast means many-to-many. Broadcasting means from one node data may be sent simultaneously to all other nodes in the network. While multicasting means from many nodes data may be sent to all other nodes in the network.

Factors affecting performance of interconnection Network

The following factors affect inter PE communications :

1. **Functionality.** It means the network's functionality. How the network supports the functions like data routing, interrupts etc.
2. **Latency.** It means the time delay for a unit message to be transferred through the network.
3. **Bandwidth.** It means the maximum data transfer rate in a network. It is measured in bytes/sec.
4. **Hardware costs.** It refers to the hardware costs like for wires, switches, connectors etc.
5. **Scalability.** It means the ability of the network to expand. If a network is scalable then it is a better network.

3.3 STATIC VERSUS DYNAMIC INTERCONNECTION NETWORKS

We tabulate the difference between static and dynamic networks.

Static Network	Dynamic Network
<ul style="list-style-type: none"> 1. They are composed of point-to-point connections. 2. The links between two processors are passive. They do not change during program execution. 3. They provide fixed connection between the nodes of a distributed system. 4. They are classified as one dimensional, 2D or 3D or as hypercube. <i>Examples :</i> Linear Array, Ring, Star, Mesh etc. 	<ul style="list-style-type: none"> 1. They are composed of switched channels. 2. The links between two processors are active or dynamic. Switches are dynamic. 3. They provide dynamic connection and are used in shared-memory multiprocessors. 4. They are classified as either single stage or as multistage dynamic networks. <i>Examples :</i> Digital Bus, Switches, cross bar network etc.

3.4 NETWORK TOPOLOGIES FOR MULTIPROCESSORS

As we know that we can have either static networks or dynamic networks. These networks use some topology for forming its network. We now discuss these topologies.

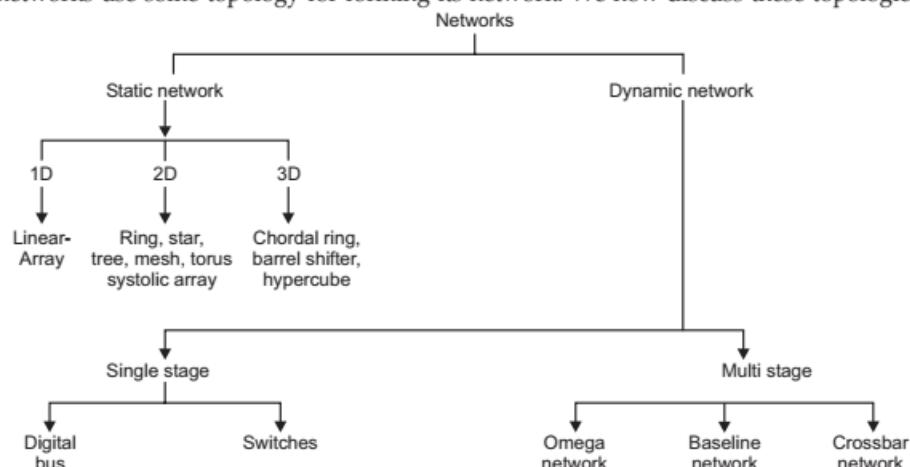


Fig. 3.3. Network topologies for multiprocessors.

We shall now describe each of these static and dynamic networks one by one.

I. Linear Array : It is the simplest connection topology. It is shown below in Fig. 3.4.

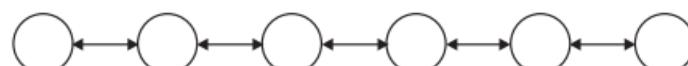


Fig. 3.4. A linear array of 6 PEs (nodes)

It is a one-dimensional network, where n nodes are connected linearly by $(n-1)$ links. The terminal nodes have a degree of 1 and the internal nodes have a degree of

2 i.e., they are connected to two other nodes. This network is not very efficient in communicating for large values of n . We summarize some of its properties :

Links : $n-1$
Diameter : $n-1$
Degree : 2 (internal nodes) and 1 (terminal nodes)
Bisection Width (b) : 1

II. Ring : A ring topology is shown below in Fig. 3.5. The type of data flow on the links in a ring can be unidirectional or bidirectional.

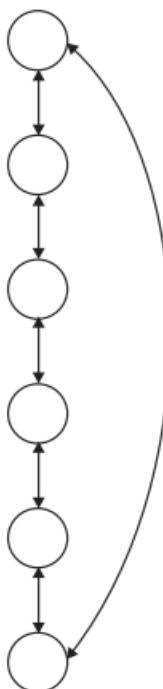


Fig. 3.5. Ring of 6 PEs.

It is a symmetric topology where each node has a degree of 2. Its other network properties are :

Diameter = $N/2$ (bidirectional ring) = N (unidirectional ring)
No. of links (l) = N
Bisection width (b) = 2
degree = 2

III. Star : A star is a two-level tree as shown in Fig. 3.6. It is used in systems with a central supervisor node. The peripheral nodes are each of degree 1, while the supervisor node has a degree of $n-1$, n being the total number of nodes.

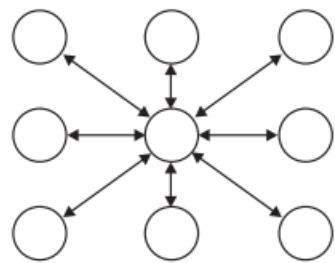


Fig. 3.6. A star topology with $n = 9$

It's other properties are discussed.

degree (d) = $n-1$
Diameter (D) = 2
No. of links (l) = $n-1$
Bisection width (b) = $N/2$

IV. Tree : A binary tree topology with four (0 to 4) levels is shown in Fig. 3.7. In general, a binary tree of k -levels has $(2^k)-1$ nodes. The degree of each node is at the most 3. With a constant node degree, the binary tree is a scalable architecture.

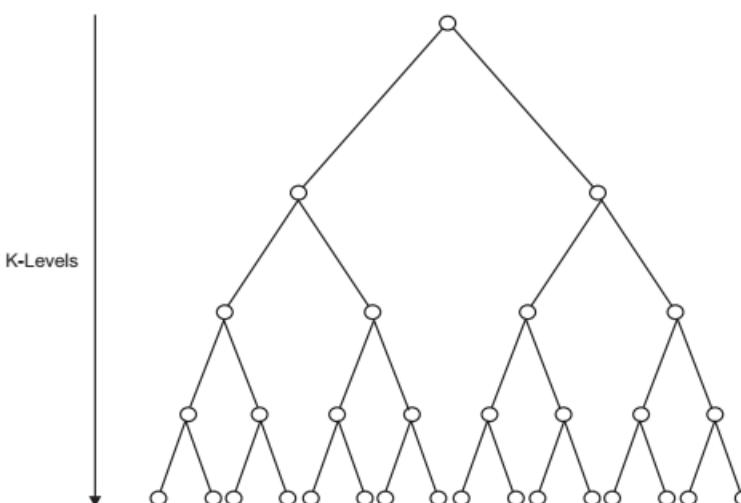


Fig. 3.7. A binary tree with k -levels

It's other properties are :

Node degree (d) = 3
Diameter (D) = 2 (K-1)
No. of links (l) = n-1
Bisection width (b) = 1

However, this conventional binary tree has some problems. The traffic congestion at the root node increases manifold. To overcome this problem, Leiserson (1985) proposed a **fat tree**. It is so called because it looks like a real tree in that its branches get thicker toward the root. This concept has been used on CM-5 machine also.

V. Mesh : A 2D, (3 * 3) mesh network is shown in Fig. 3-8(a). The Illiac-IV, DAP 600 multiprocessors, CM-2 and Intel Paragon use different types of mesh architecture. In general, a K-dimensional mesh has n^k nodes, where n is the number of nodes in each dimension with each node having an internal node degree of $2K$.

Dally's J-machine is a 3D-mesh

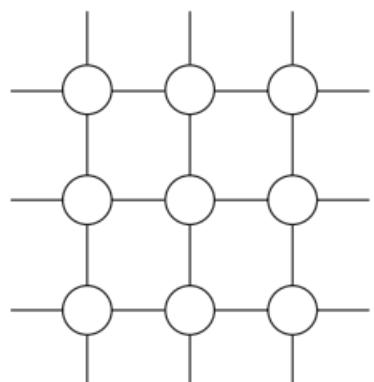


Fig. 3.8 (a). Mesh topology

It's another variant that has been used in Illiac IV is also shown in Fig. 3.8(b).

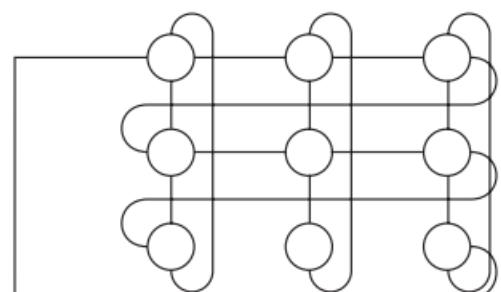


Fig. 3.8 (b). Illiac IV mesh

It's other properties are :

Node degree (d) = 4
Diameter (D) = $2(r-1)$ where ($r * r$) is mesh size and $r = \sqrt{n}$
Number of links (l) = $2n-2r$
Bisection width (b) = r

For Illiac mesh, we have these properties :

Node degree (d) = 4
Diameter (D) = $r-1$
Number of links (l) = $2n$
Bisection width (b) = $2r$

VI. Torus : The torus architecture (topology) is shown in Fig. 3.9. It is another variant of the mesh architecture. This topology combines the ring and the mesh and extends to higher dimensions. The torus has ring connections along each row and along each column of the array. In general, a ($n * n$) binary torus has a node degree of 4. It is a symmetric topology.

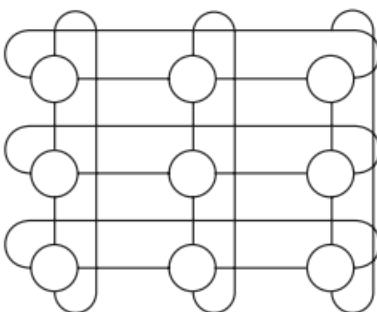
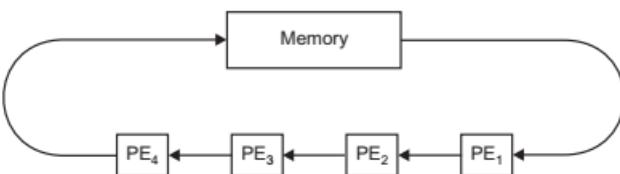


Fig. 3.9. Torus of size (3×3)

It's other properties are :

Node degree (d) = 4
Diameter (D) = $2 \lfloor r/2 \rfloor$
No. of links (l) = $2n$
Bisection width (b) = r (where $r = \sqrt{n}$)

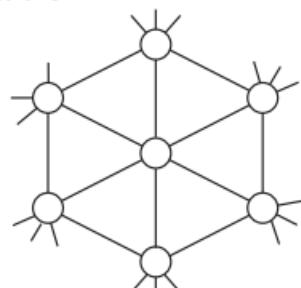
VII. Systolic Arrays : The general idea of pipe lining is used to construct a special purpose high speed electronic circuits for performing various mathematical operations like matrix multiplication, Fourier transformations etc. These electronic circuits are called **systolic arrays**. They are so called because the data from a memory unit is sent in a **rhythmic fashion** through an array similar to the way in which human heart regularly pumps blood via the arteries. The **data is processed by a set of PEs** and the **transformed input is returned to the memory**. A typical systolic system is shown in Fig. 3.10.

**Fig. 3.10.** A systolic array architecture.

The **primary characteristics** of a systolic array are as follows :

1. The flow of data is rhythmic and regular.
2. Data can be bidirectional.
3. Each cell or PE performs identical operation.
4. The processing time of all cells are equal.
5. The communication between cells is serial and periodic.
6. Individual cells are connected to their nearest neighbours.
7. Cells except those at the boundary of the array do not communicate with the outside world.

A systolic array looks like this :

**Fig. 3.11.** Systolic array topology.

Systolic Arrays (SA)

1. Systolic Arrays are a type of specialized *attached processor* (AP) only.
2. None of the PEs have any local memory.
3. Fig. 3.11 shows a macro-pipeline.
4. A homogeneous architecture, multidimensional data flow.
5. On every systolic clock pulse, each PE passes its result to the next processor in chain. So, complex multicycle instructions are not implemented.
6. Systolic arrays are said to be 'lock stepped' or synchronous. There is no master control as is found in array processors.
7. *Please note that when network of SAs is active, no work is being done and vice versa also holds.*
8. SAs are used in *compute bound* applications with *data locality*. By *compute bound*, we mean those computations that require more operations to be performed than there are data items.

Data locality exists if a function's operand are near neighbours in a given data structure.

9. SAs come in variety of different topologies.
10. SAs are modular in design meaning that more PEs can be causally added or removed without any effect on the memory bandwidth.
11. Examples of general purpose systolic arrays are-
 - Matrix-1 by Saxpy computer corporation.
 - Warp by Carnegie Mellon university (Intel iWarp, 1990).
 - Colossus Mark II machine (1944).
12. Kung in 1982, proposed SAs.
13. SAs are pipelined multiprocessors.
14. 2 things synchronize these pipelines
 - A Global Clock and an explicit timing delay.
15. SA architecture is actually a network of PEs that rhythmically compute data by circulating it through the system.
16. SAs are a variation of SIMD computers that incorporates large arrays of simple PEs that use vector pipelines for data flow.
17. SAs exploit a high degree of parallelism (through pipelining) and can sustain a very high throughput.
18. Connections are short, design is simple and thus highly scalable.
19. SAs are robust, highly compact, efficient and cheap to produce.
20. But SAs are highly specialized and quite inflexible as to the types of problems they can solve.
21. **Applications:** SAs are used for repetitive tasks like Fourier transformations, image processing, data compression, shortest path problems, sorting, signal processing and various matrix calculations (like multiplication, inversion etc)

Other Applications

1. Matrix multiplication, inversion, decomposition etc.
 2. Polynomial evaluation.
 3. Image processing.
 4. Speed and signal processing.
 5. In Artificial Neural Networks (ANNs).
22. **Another variant of SA:**

A super systolic arrays is a generalization of SA. Rainer Kress has introduced a generalized systolic array:
 - the super systolic array. He used simulated annealing process.
For eg. Kress Array is a reconfigurable version of the super systolic array.
 23. SAs are data-stream driven whereas John Von Neumann/conventional/sequential machines are instruction-stream driven.
 24. SAs need *multiple data counters* to generate multiple data streams. Hence, it supports data parallelism.
 25. The data streams entering and leaving the ports of the array (PEs) are generated by auto-sequencing memory units (ASMs).
Each ASM has a *data counter*.
 26. Usually they use *mesh-like topology*.
 27. *Systolic Arrays* and a *asynchronous* handshake between PEs forms another architecture called as *wave front arrays*.

Advantages of SAs

1. Degree of pipelining is high.
2. Simple I/O system.
3. High speed and low cost systems.
4. Synchronized multiprocessing.
5. Modular design.

Disadvantages of SAs

1. Signal delays results in Global synchronization limits.
2. Higher B.W. is required for inter-PE communications.
3. Poor run-time fault tolerance.

VIII. Chordal Ring : A chordal ring is shown in Fig. 3.12. If we increase the node degree from 2 to 3 or 4, we get 2-chordal rings. In general, **the more the links are added, the higher is the node degree and the shorter is the network diameter.**

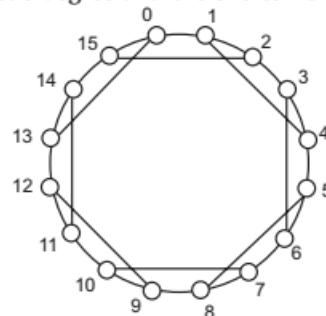


Fig. 3.12. Chordal ring.

IX. Barrel Shifter : A barrel shifter is obtained from the ring only by adding extra links. A node is connected to another node if they are at a distance of power of 2. It's properties are :

Network size, $N = 2^n$
 Node degree (d) = $2n-1$
 Diameter (D) = $n/2$

It's topology is given below in Fig. 3.13.

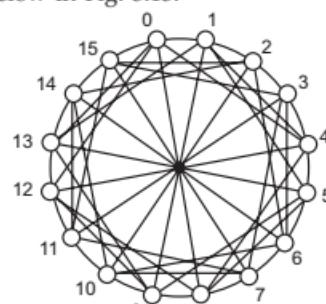


Fig. 3.13. Barrel shifter

X. Hypercube : A hypercube architecture can be of different dimensions. We define its dimension as the node degree of each node. It is a symmetric topology. So, the node degree is same for all nodes. Various hypercubes with different dimensions are shown in Fig. 3.14.

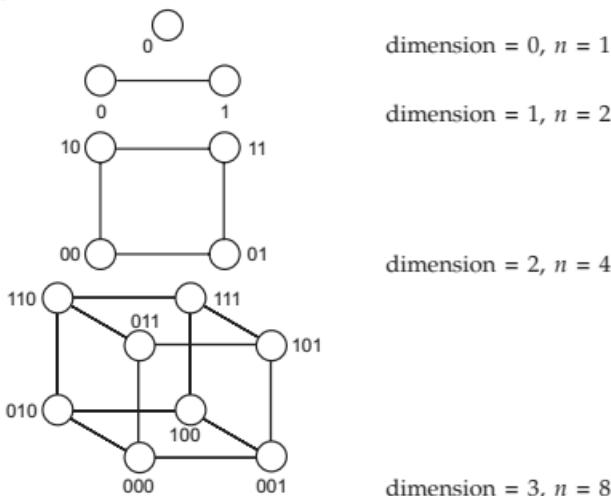


Fig. 3.14. Hypercubes with dimensions 0, 1, 2 and 3.

This can go on. In general, an n -dimensional hypercube has 2^n nodes. Each node is labelled as a binary number. Please note here that an n -dimensional hypercube needs n -bits to label each node. Two nodes are connected if and only if the Hamming distance between them is 1. Messages transit in 'hops' between such node pairs. When two nodes or processors are not directly connected then the messages transit between them in multiple hops. Note that the maximum number of hops necessary under the worst-case for an n -dimensional hypercube is n . This method of message transfer can be done either automatically or manually.

It is also possible to transform one network to another network. Hypercubes can be transformed into different topologies like the mesh or the tree without affecting the connections of the processors at each node. This is known as **Embedding**. Thus, embedding is defined as the process of transformation of one network to another network.

For example :

- (a) A hypercube can embed a mesh.
- (b) A hypercube can embed a tree.
- (c) A ring can be embedded in a torus.

Please note that since hypercube is a symmetric architecture, any processor can be taken as the root of the tree. Also note that when a ring is embedded into a torus, we say that the embedding is *perfect*. That is, each link in the ring exists in a mesh. Similarly, a line can be embedded into a mesh. Such perfect embeddings are impor-

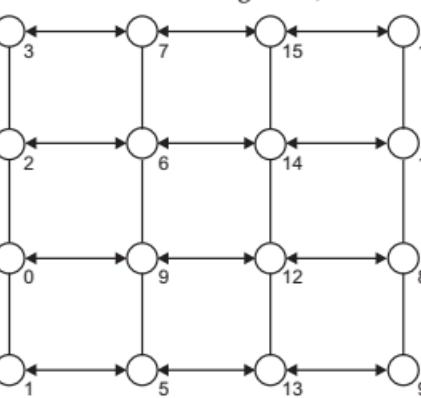


Fig. 3.15. Hypercube embedding a mesh.

tant because we may have an algorithm that requires only communication between the nearest neighbors in a specific network. Fig. 3.15. below shows how a hypercube can embed a mesh.

Similarly, a hypercube can embed a tree also. This is shown in Fig. 3.16. below.

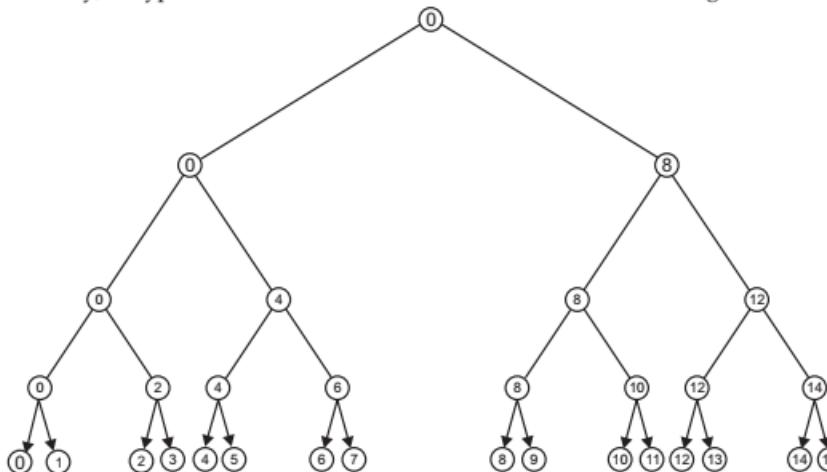


Fig. 3.16. Hypercube embedding a tree.

Say, our multiprocessor system has a mesh architecture and our algorithm has line/ring communication patterns. It is now crystal clear that with the proper allocation of processes to processor nodes, all messages can reach their destinations through one link and thereby encounter minimum delay.

The quality of embedding is indicated by a term – **dilation**. It is defined as the maximum number of links in the embedding network corresponding to one link in embedded Network. *For example :* Embeddings such as a line or ring into mesh or torus or a mesh onto a hypercube have a **dilation of 1**. Also, embedding a tree into a mesh results in a **dilation of 2**. This is because two links are needed from the root to the next nodes of the tree and to the next nodes. While only one link is needed lower down the tree.

A tree can be embedded to a mesh as shown below in Fig. 3.17.

Here (in Fig. 3.17.), the nodes shaded correspond to the nodes in the tree and those marked as "A" are additional nodes in the mesh to form the tree in the mesh.

Similarly, we have second category of dynamic network. These networks can be again sub-divided as single stage or multistage networks. We describe each of them briefly now.

I. Digital Bus

A digital bus is a collection of wires and connectors for doing communications among PEs, memories and I/O devices. These devices are attached to the bus. This bus is used

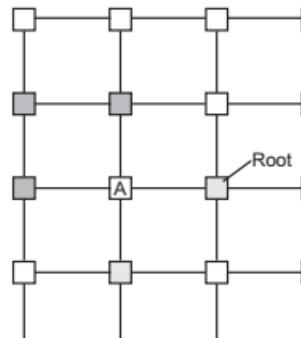


Fig. 3.17. Tree embedded into a mesh

for only one transaction at a time between a source computer (master) and a destination computer (slave). Bus contentions may arise when multiple requests arrive. So, a bus arbitration logic must be able to allocate or deallocate the bus service request one by one. That is why this bus is also known as a *contention bus* or a *time-sharing bus* among multiple functional modules. It is not costlier to implement a bus but the drawback is that it has a limited bandwidth.

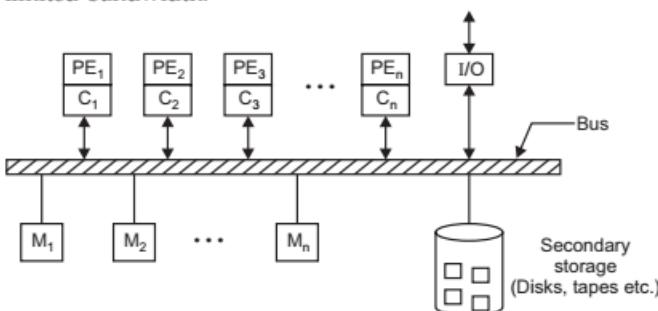


Fig. 3.18. Shows a multiprocessor systems with bus topology.

PE : Processing Element/Processor

M : Main Memory

C : Cache Memory

I/O : Input-output subsystem

As shown in Fig. 3.18, we observe that there is a common digital bus that connects to PE_i ($i \rightarrow 1$ to n), to M_i ($i \rightarrow 1$ to n) and to secondary storages or AUX storages like disks and tapes and to the I/O subsystem.

How this bus is practically implemented ?

The system bus is often implemented on a backplane of a PCB (Printed Circuit Board). Other boards for processors, memories or device interfaces are plugged into the backplane board via connectors or cables.

The **active devices** (or master devices) like PEs or I/O devices, **generate requests** to address the memory. The **passive devices** (or slave devices) like memories or peripherals, respond to the requests. This digital bus is used on a time-sharing basis. We have to take care of issues like bus arbitration, interrupt handling, coherence protocols and transaction processing.

II. Switch Modules/Switches.

A $(m \times n)$ switch module has m inputs and n outputs. A **binary switch** corresponds to a 2×2 switch module in which $m = n = 2$. Note that m and n need not be equal but practically speaking, m and n are chosen as integer power of 2, i.e.,

$$m = n = 2^k \text{ (where } k \geq 1\text{)}$$

So, some other module sizes can be (4×4) , (8×8) .

We now list some of the switch modules alongwith their legitimate states in a tabular form:

Switch sizes	Legitimate states	Permutation connections
1. 2×2	$4 (n^n = 2^2 = 4)$	2
2. 4×4	256	24
3. 8×8	16, 777, 216	40, 320
4. $n \times n$	n^n	$n!$

Here, each input can be connected to one or more of the outputs. However, there should not be any conflicts at the output terminals.

Note : One-to-one and one-to-many mappings are allowed but many-to-one mappings are not allowed due to conflicts at the output terminal.

Actually, dynamic networks involve two classes :

1. Single stage Networks.
2. Multistage Networks.

Single-Stage Networks : A single stage network is a switching network with N input selectors (IS) and N output selectors (OS) as shown in Fig. 3.19 below.

Each IS is essentially a $1 : D$ demux. Each OS is an $M : 1$ MUX where

$$\begin{aligned} 1 \leq D \leq N \\ 1 \leq M \leq N \end{aligned}$$

For example : The crossbar switch network is a single stage network with $D = M = N$. To establish a desired connecting path, different path control signal will be applied to all IS and OS selectors. A 2×2 crossbar switch can connect two possible patterns : straight or crossover. In general, an $n \times n$ crossbar can achieve $n!$ permutations.

The single-stage network is also called as a **recirculating network**. The data items may have to recirculate through the single stage several times before reaching their final destination. The number of recirculations needed depends on the connectivity in the single-stage network. Please note that the higher is the hardware connectivity, the less is the number of recirculations. The crossbar network is an extreme case in which only one circulation is needed to establish any connection path. However, the fully connected crossbar networks have a cost $O(N^2)$. The recirculating networks have the cost of $O(N \log N)$ or even lower.

III. Multistage Networks

Are also called as MINs. They have been used in SIMD and MIMD computers. A MIN network uses **multiple ($a \times b$) switches** in each stage. In between these switches, fixed interstage connections (ISC) are used. The switches can be dynamically set to establish the desired connections between the input and outputs.

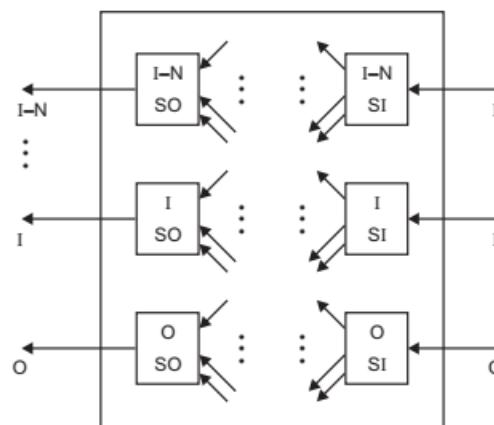


Fig. 3.19. Single Stage interconnection Network – A conceptual view.

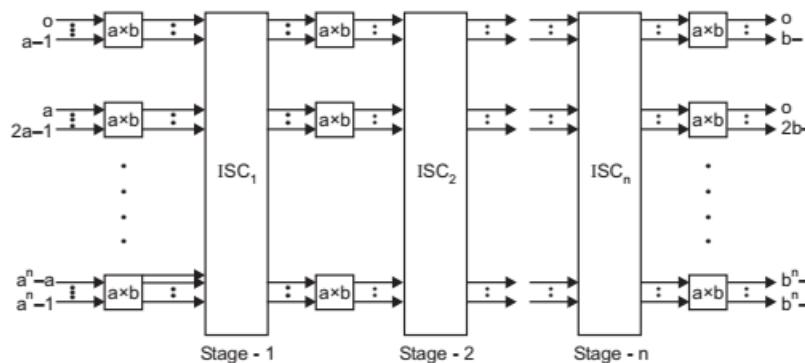


Fig. 3.20. A MIN with $a \times b$ switches.

Where $a \times b$: Switches with a -inputs and b -outputs

ISC_i ($i \leftarrow 1$ to n) : Interstage connection

S_i ($i \leftarrow 1$ to n) : Stages from 1 to n .

So, we observe that many stages of interconnected switches forms a multistage SIMD network. These networks are characterized by three features :

1. The Switch Box.
2. The network topology.
3. The control structure.

Many switch boxes are used in MINs.

What is a switch box ?

Each box is an interchange device with two inputs and two outputs as shown below in Fig. 3.21

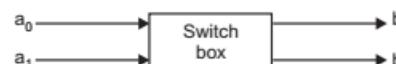
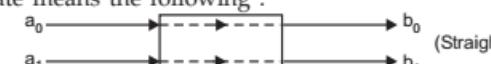


Fig. 3.21. Switch Box.

This switch-box can be in any of these four states :

- | | |
|---------------------|----------------------|
| (a) Straight | (b) Exchange |
| (c) Upper broadcast | (d) Lower broadcast. |

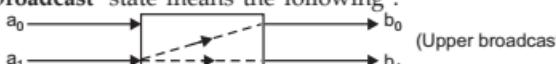
A 'straight' state means the following :



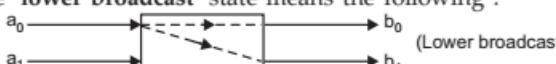
The 'exchange' state means the following :



The 'upper broadcast' state means the following :



Similarly, the 'lower broadcast' state means the following :



Note :

1. A two function switch box can assume either the straight or the exchange states.
2. A four-function switch box can be in any one of the four legitimate states.

Actually, multistage networks are capable of connecting an arbitrary input terminal to an arbitrary output terminal. These networks can be one-sided or two-sided. **One-sided** networks (or full switches) have I/O ports on the same side. The **two-sided multistage networks** have an input side and an output side. These networks can be divided into three classes :

1. Blocking
2. Rearrangeable
3. Nonblocking

Let us discuss these network one by one.

1. Blocking Network : In blocking networks, simultaneous connections of more than one terminal pair may result in conflicts in the use of network communication links.

For example : Omega network, flip, n-cube, baseline network, data manipulator are all blocking networks.

2. Rearrangeable Network : A network is known as a rearrangeable network if it can perform all possible connections between inputs and outputs by rearranging its existing connections so that a connection path for a new I/O pair can always be established.

For example : A Benes network. This network topology has been extensively studied for use in synchronous data permutation and in asynchronous interprocessor communication.

3. Nonblocking Network : A network which can handle all possible connections without blocking is called as a nonblocking network.

For example :

1. **Clos Network.** In this network a one-to-one connection is made between an input and an output.
2. **Crossbar switch Network.** Herein, one-to-many connections are made between an input and an output. Please note that it can connect every input port to a free output port without blocking.

In general, a multistage network consists of n stages where $N = 2^n$ is the number of input and output line.

So, each stage may use $N/2$ switch boxes. The interconnection patterns from stage to stage determine the network topology. Each stage is connected to the next stage by atleast N paths. The network delay is proportional to the number, n , of stages in a network. The cost of a size, N , multistage network is proportional to $N \log_2 N$. It is the control structure of a network that determines how the states of the switch boxes will be set. Three types of control structures are used in a network construction :

- (a) **Partial Stage Control.** It has $(i + 1)$ control signals which are used at stage $- i$ for $0 \leq i \leq n - 1$.
- (b) **Individual Stage Control.** It uses the same control signal to set all switch boxes in the same stage. It requires n sets of control signals to set up the stage of all n stages of switch boxes.

3. Individual box control : A separate control signal is used to set the state of each switch box. It offers more flexibility in setting up of the connection paths. It requires $n^2/2$ control signals which will increase the complexity of the control circuitry.

We shall now discuss one **blocking network i.e., Omega network.**

IV. Omega Network

A network build using $\log_2 N$ cascaded switches using shuffle connection is called an **omega network**. To simplify our design we consider a 3-stage omega network. Also assume that C_1 , C_2 , C_3 are the three control signals. We draw the following Omega network which has 3-stages in Fig. 3.22.

Please note that the input labels of switch at i^{th} stage is obtained by shuffle transforming the output labels of the corresponding $(i - 1)^{\text{th}}$ stage switch. For example the input labels of switch 23 (see Fig.3.22.), namely 001 and 011 are obtained by left circular shifting (*i.e.*, shuffle transforming) the labels 100 and 101 which are the output labels of switch 13. If a PE attached to input i is to be connected to a memory module attached to output j , exclusive OR of the binary equivalents of i and j is found. If $(i \oplus j) = p$, the least significant bit of p is used as control bit C_1 (see Fig.3.23), the next significant bit as C_2 and the most significant bit as C_3 . For example if input 7 is to be connected to output 3, $(111 \oplus 011 = 100)$. Thus $C_1 = 0$, $C_2 = 0$ and $C_3 = 1$. Remember that $C = 0$ direct connects input to output and $C = 1$ cross connects them. By examining Fig. 3.23. it is clear that $C_1 = 0$, $C_2 = 0$, $C_3 = 1$ connects 7 to 3. Observe that several connection between input and output can be made at the same time. The omega network was used in some **experimental multicomputers**.

Another multistage network which has been used in many commercial computers is called a **butterfly network**. A (4×4) switch using 4 butterfly switch boxes is given in Fig. 3.23. Observe that the butterfly connection is not very different from the generalized network. The output switches of Fig. 3.23. are butterfly switches. Alternate switches (see from top to bottom) of the previous stage are also butterfly connected switches.

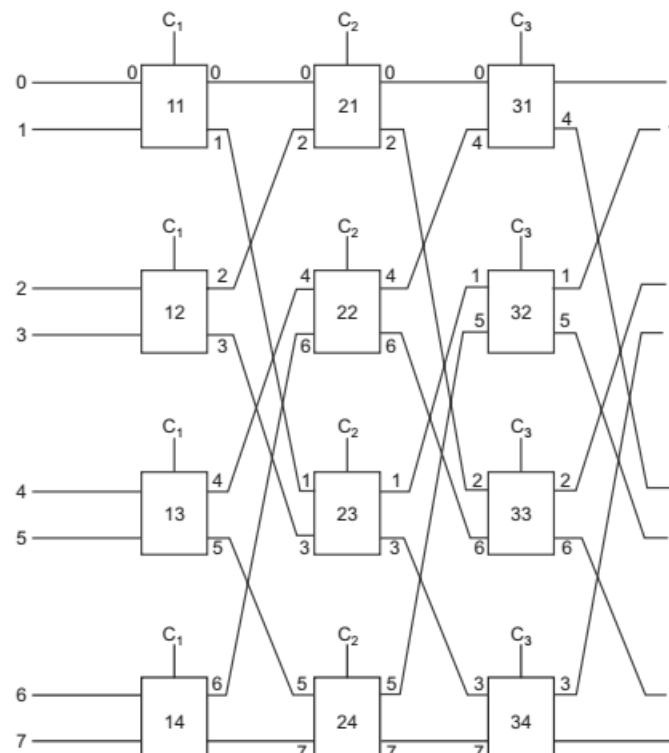


Fig. 3.22. Ω network with stages = 3

A butterfly switch network also does not allow an arbitrary connection of N inputs to N outputs without conflicts. It has been shown that if two butterfly networks are laid back to back so that data flows forward through one and in reverse through the other, then it is possible to have a conflict free routing from any input port to any output port. This back-to-back butterfly is known as a **Benes network**. Benes networks have been extensively studied due to their elegant mathematical properties but not widely used in practice.

V. Baseline Network

As studied by Wu and Feng (1980), a baseline network is a blocking multistage network that is generated recursively. A 8×8 baseline network is shown in Fig. 3.24.

The first stage is 8×8 block in above figure. The second stage is 4×4 sub-blocks. In general, the first stage contains $N \times N$ block and the second stage contains two $(N/2) \times (N/2)$ subblocks as B_0 and B_1 in Fig. 3.25. below.

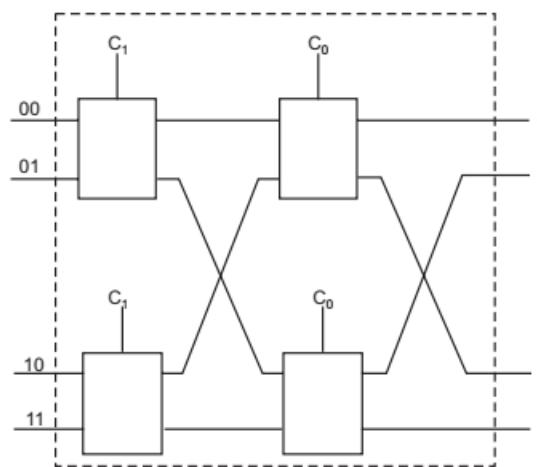


Fig. 3.23. Butterfly network of size (4×4)

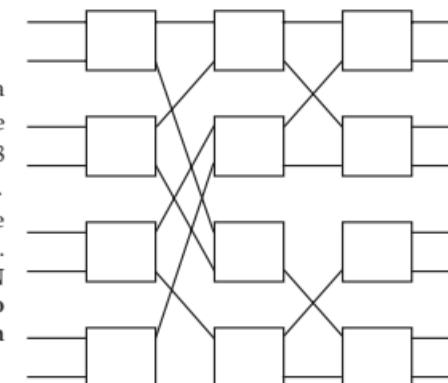


Fig. 3.24. (8×8) baseline network

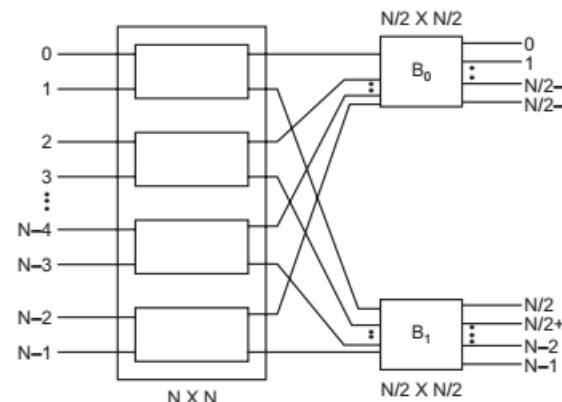


Fig. 3.25. Recursive network

The construction of such a network can be applied recursively.

VI. Crossbar Network

A crossbar network is a single-stage switch network which provides the highest bandwidth and the highest interconnection capability. For example, in case of a telephone switchboard, the crosspoint switches provide dynamic connections between the source and the destination pairs. Each crossbar switch has a *crosspoint switch* which can provide a *dedicated path between a pair*. The switch can be set ON or OFF dynamically upon program demand.

There are 2 types of crossbar network configurations :

Type I : An Interprocessor-memory crossbar network for multiprocessors.

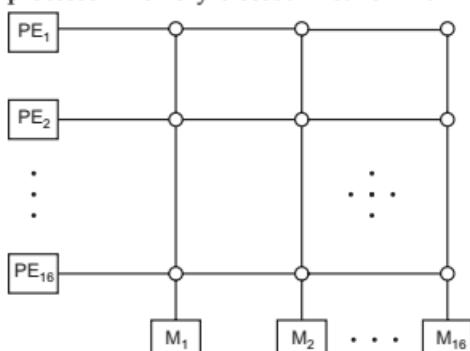


Fig. 3.26. (a)

Type II : An Interprocessors crossbar network for vector processors.

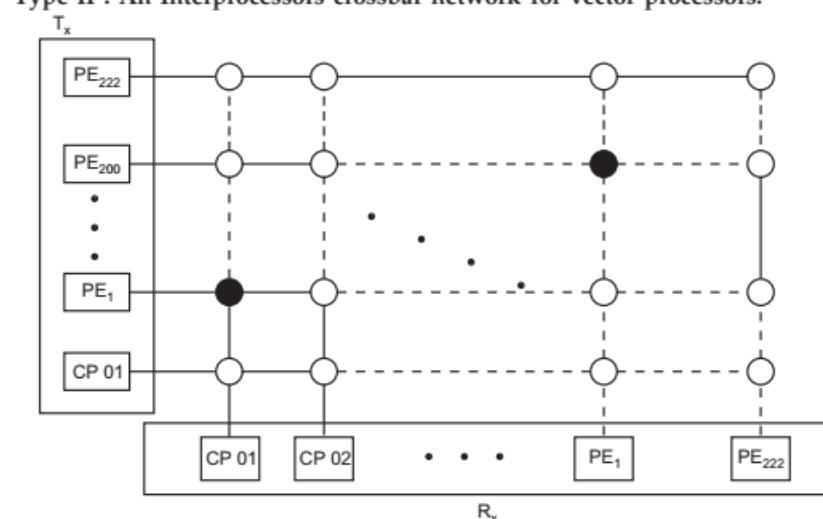


Fig. 3.26. (b)

Note that a 'dark-circle' indicates an ON condition and an OFF state is denoted by an empty circle.

As shown in Figure 3.26 (a), a crossbar network is used between the processors (PEs) and the memory modules ($M_1 \dots M_{16}$). This memory access network has been used in C.mmp multiprocessor at Carnegie-Mellon University. This multiprocessor has implemented a 16×16 crossbar network which connects 16 PDP 11 processors to 16 memory modules, each of which has a capability of 1 million words of memory cells. The 16 memory modules can be accessed by at most 16 PEs simultaneously. Please note that each memory module can satisfy only one PE request at a time. When multiple requests arrive at the same memory module simultaneously, the crossbar must resolve the conflicts. The behavior of each crossbar switch is very similar to that of a bus. However, each PE can generate a sequence of addresses to access multiple memory modules simultaneously. Thus in Fig. 3.26. (a) only one crosspoint switch can be set on in each column. However note that several crosspoint switches can be set on simultaneously in order to support parallel memory accesses.

However, in Type-II category (Fig. 3.26. (b)) there is a large crossbar used in a vector parallel processor. The PEs are processors with attached memory. The CPs stand for control processors. These CPs supervise the entire system operation. Herein, only one crosspoint switch can be set on in each row and each column. Since only one to one connections are provided so the $(n \times n)$ crossbar connects at most n (source, destination) pairs at a time.

3.5 INTERPROCESSORS COMMUNICATION NETWORK

3.5.1 Multiport Memory Model

As shown in Fig. 3.27. a multiport memory model consists of separate buses between each memory module and each processor. The bus consists of address, data and control lines.

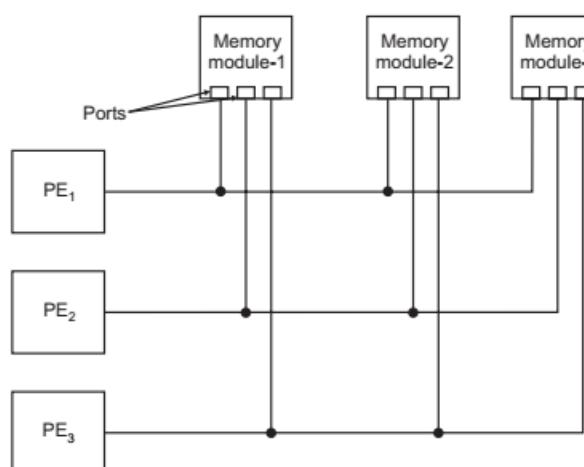


Fig. 3.27. A multiport Memory Model.

Each memory module accommodates the buses from all PEs. So, each memory module must have the number of ports equal to the number of processors (or PEs) i.e.,

Number of ports = Number of processors (PEs)
--

That is why it is named as a multiport memory. Each memory module has control and arbitration logic. This logic determines which port will have access to memory module at any given time. When more than one processors request simultaneously to access the same memory module then the arbitration logic within the memory module decides the priorities according to some algorithm and grants the access to high priority processor. Mostly the priorities are resolved using static priority algorithm such as daisy chaining scheme. **For example**, if PE-1 and PE-2 requests to access memory module-1 at the same time, then PE-1 will have a higher priority than PE-2. This is because in memory module-1, the physical location of port of PE₁ is near to bus arbiter than that of port for processor or PE-2.

The multiport memory system can also support nonblocking access to the memory if a fully-connected topology is used. Since each access is a separate operation, it also permits the exploitation of interleaved memory addresses for access by a single processor.

Examples : Univac 1100/90, IBM 370-168 are some multiport systems.

Let us now compare three types of interconnection structures that are used in the multiprocessors :

I. Multiprocessors with line shared bus :

1. Lowest overall system cost for hardware and are least complex.
2. Very easy to physically modify the hardware system configuration by adding or removing functional units.
3. Overall system capacity is limited by the bus transfer rate. **Failure of the bus is a catastrophic system failure.**
4. Expanding the system by addition of functional units may degrade overall system performance *i.e.*, throughput
5. The system efficiency attainable is the lowest of all the three basic interconnection systems.
6. This organization is usually appropriate for smaller systems only.

II. Multiprocessors with crossbar switch :

1. This is the most complex interconnection system. There is a potential for the highest total transfer rate.
2. The functional units are the simplest and cheapest since the control and switching logic is in the switch.
3. This organization is cost-effective for multiprocessors only as a basic switching matrix is required to assemble any functional units into a working configuration.
4. Adding new functional units usually improves overall performance. There is the highest potential for system efficiency such as for system expansion without reprogramming of the O.S.
5. The expansion of the system is limited by the size of the switch matrix which can often be modularly expanded within initial design or other engineering limitations.
6. The reliability of the switch and therefore the system, can be improved by segmentation and/or redundancy within the switch.

III. Multiprocessors with multiport memory :

1. They require the most expensive memory units since most of the control and the switching circuitry is included in the memory unit.
2. The characteristics of the functional units permit a relatively low cost uniprocessor to be assembled from them.
3. There is a potential for a very high total transfer rate in the overall system.
4. The size and configuration options possible are limited by the number and the type of memory ports available. This design decision is made quite early in the overall design process and is difficult to modify.
5. A large number of cables and connectors are required.

Advantages of Multiport Memory :

1. High throughput because of multiple data transfer paths between processors and memory modules.
2. More security against unauthorized access.

Disadvantages of Multiport Memory :

1. As the number of PEs increases, more and more expensive control and arbitration logic is required within the memory module.
2. For large multiprocessor systems, more hardware is required because of increase in the interconnection paths between PEs and memory modules. Thus, the expansion of the system is costly.

3.5.2 Memory Contention

Control variable (like locks, event counters etc) in an O.S. are frequently changed during the runtime. Conventionally, they have been logically grouped into tables and each table is loaded into a consecutive segment of memory at runtime. **Please note that this is a good practice for building a structure system.** This is also acceptable when all the previous OSs were designed to be run on uniprocessors. However, it is not acceptable in a multiprocessor system where all the CPUs can access a table at the same time. In a highly parallel multiprocessor system, the contention may exist at a memory module containing a heavily accessed table. It is even worse if each PE has its own local cache. As the accesses to the control variable are generally considered as exclusive, a cache line containing such variable may only be allowed to reside in one cache at a time. The line can be **ping-ponging** among the caches of the CPUs which need the line closely in time. A technique to alleviate this pingponging is to put variables that are very frequently referenced but are logically independent in separate cache lines.

However, the shared memory systems lack scalability due to the contention problem. The winner can access the memory while the losers must wait. **Please note that the larger the number of PEs, the higher is the probability of memory contention.** Beyond a certain number of PEs, this probability is so high that adding a new processor to the system will not increase performance. To reduce this memory contention problem, shared memory systems are extended with special, small size local memories called as **cache memories**. Whenever a PE accesses a memory, this cache is checked first. If the required data is available in cache (hit), it is accessed without any memory contention. If the required data is not available (miss), then the required page containing the data is transferred into it. **Please note here that the main assumption is that the shared memory programs generally provide a good locality of reference.**

For example : Many a times, a procedure or a function may need to access just local data that is there in cache. But this may not always be true. This reduces the cache performance. So, it introduces a cache-coherency problem. We have schemes to solve these problems and will be discussed later.

3.6 STRUCTURE OF PARALLEL COMPUTERS

We have seen and studied the various ways of exploiting parallelism. Now let us see how to organize these PEs so that they all work cooperatively to solve a given problem faster than a single computer. There are various methods of doing so. They are as follows:

1. Shared Memory Multiprocessors.
2. Message Passing Multicomputers.
3. Pipelined Parallel Computers.
4. Array Processors.
5. Multilink Multidimensional System (MMS)

We shall describe each of them now.

I. Shared Memory Multiprocessors : In this type of computer, around 8 to 128 microprocessors or very powerful processors are connected to a set of memory modules using a communication network. This set of memory modules can be accessed by any PE & forms the main memory with one common address space. The program to be executed, data & results are stored in it. The most commonly used communication network to connect PEs to main memory is a bus. The bus is shared by processors & the memory. There will be lines in the bus to address locations in memory & also to address PEs. The bus is a common resource used by PEs to access memory & other PEs. **Two PEs cannot use the bus at the same time.** Thus if 2 or more PEs want to access memory simultaneously in such a system, a queue is formed. A shared bus limits the number of PEs which can be effectively used in this system. Due to sharing of the bus & memory, the effective speed of multiprocessor is reduced by 55%. **We will solve a problem now.**

Example. Compute the maximum speed of processing for the following data with 10 PEs :

Processing speed of PE = 1 MIPs

Data handled = 64 bits (8 bytes)

Time to store result in memory = 0.25μ sec.

Bus speed = 40 Mega bytes/Sec. ? [KUD, B.E. (CSE) 8th sem., 1997]

Solution. Time to carry out 1 instruction

$$= \frac{1}{10^6}$$

$$= 1\mu \text{ sec}$$

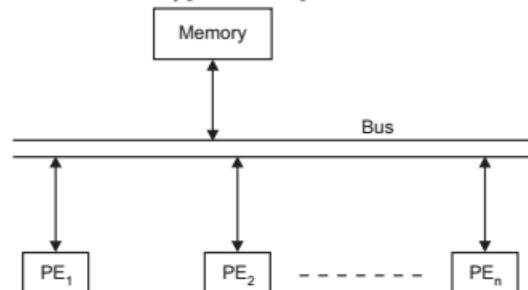


Fig. 2.28. A shared memory multiprocessors

If memory is connected to a bus and the bus speed = 40 Mega Bytes/sec (given)

$$\text{Then, time to transport the result (8 bytes) on the bus} = \frac{8}{(40 \times 10^6)} = 0.2\mu \text{secs}$$

It is given that 10 PEs share the bus. And that PEs request access to the bus with equal probability.

\therefore The minimum time taken to get access to the bus = 0.

The maximum time to get access to bus = $9 \times 0.4 = 3.6\mu \text{ secs}$

(if FCFS queue is assumed)

\therefore Average time to get access to the bus

$$= \frac{3.6}{2}$$

$$= 1.8\mu \text{ secs}$$

The average time to carry out an instruction

$$= (\text{Average bus access time}) + (\text{Time on bus}) + \left(\begin{array}{l} \text{average instruction execution time} \\ \text{access time} \end{array} \right)$$

$$= 1.8 + 0.2 + 0.2 + 1$$

$$= 3.2\mu \text{ secs}$$

\therefore The effective speed of 10 PEs

$$= \frac{10}{3.2} \text{ mips}$$

$$= 3.125 \text{ mips}$$

Note that if all PEs work simultaneously without the need to access a shared memory using a bus then,

$$\text{Maximum speed of computing} = \frac{10}{1.4} = 7.14 \text{ mips}$$

We find that due to sharing of the bus and the memory, the effective speed of the multiprocessor is reduced by 55%.

II. Message Passing Multicomputers :

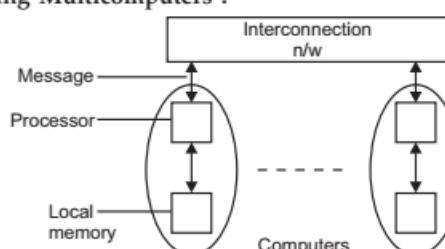


Fig. 3.29. Message Passing Multicomputer.

Here, each computer consists of a processor & local memory that is not accessible by other processors. In a multicomputer, memory is distributed among the computers & each computer has its own address space. A processor can only access a location in

its own memory. The inter-connection network is provided for processors to send messages to other processors. These messages can include data that other processors may require for their computations. Such multiprocessor systems are usually called **message passing multiprocessors/multicomputer**.

III. Pipelined Parallel Computer : These computers divide a job into a sequence of tasks in such a way that the time taken to carry out each task is identical (T). Specialized PEs are designed to carry out each of these tasks. These PEs are connected serially. Say, we want to add 2 floating point nos., a_1 and b_1 . The floating point representation of a_1 and b_1 are

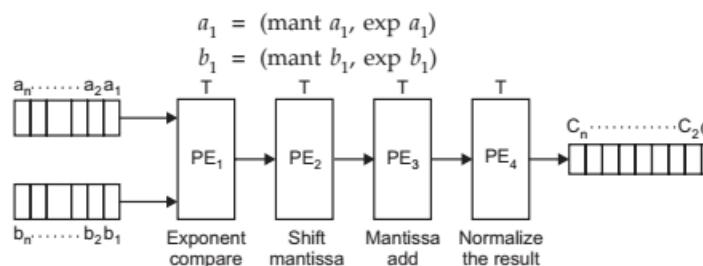


Fig. 3.30. FP Adder

The addition operation is broken up into 4-tasks. Each task takes T units of time to execute. A sequence of operand pairs to be added $(a_1, b_1), (a_2, b_2) \dots (a_n, b_n)$ are queued by & one pair sent every T units of time through the PEs. *The first sum $a_1 + b_1 = c_1$ will be obtained after $4T$ units of time.*

$$\begin{aligned} \therefore c_2 &= a_2 + b_2 \text{ will be ready after } 5T \\ c_3 &= a_3 + b_3 \text{ will be ready after } 6T \text{ & So on} \\ &\vdots \\ &\vdots \\ c_n &= a_n + b_n \text{ will be ready after } (4 + (n-1))T. \end{aligned}$$

This system will work only if same operation is to be performed on a sequence of similar operand pairs. An array of operands of same type is known as a vector. So, here, we are adding a vector $(a_1, a_2, a_3 \dots a_n)$ to another vector $(b_1, b_2, b_3 \dots b_n)$ to produce sum-vector $(c_1, c_2, c_3 \dots c_n)$

IV. Array Processors : Yet another method of adding 2 vectors is to have an array of n PEs with each PE storing a pair of operands i.e.,

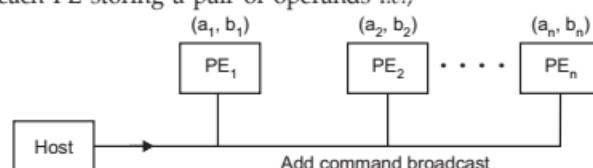


Fig. 3.31. Array Processor/SIMD

An add instruction is broadcast to all the PEs. All PEs then start adding their respective operand pairs simultaneously. If each PE takes T units of time to add, then the time to add vector is T . Such an organization of PEs is called an **array processor**. The array processor is attached to the host computer. The host computer stores the program to be executed. The data to be processed are stored in local memories of PEs of

this array. When an arithmetic operation on vectors is encountered by host computer, it is broadcast to all PEs in the array. All PEs independently & simultaneously perform arithmetic operation on data stored in their private data memories. **Note that an array processor uses data parallelism.** In above Fig. 3.31 , an ADD instruction is broadcast to all PEs, simultaneously. This computer is also known as a SIMD computer.

V. Multilink Multidimensional Computer System (MMS)

This system has been built at Indian Institute of Science, Bangalore. There were several reasons for the same. Some of them were :

1. There was a need to build a cheaper multicomputer that can be used as a **test-bench to test ideas in parallel computing.**
2. The aim was to evolve an **inter-computer communication device** which is capable of transferring data between computers at a faster rate
3. Also, to build a system which uses well established computing systems as individual nodes so that field proven OS and language compilers available for such a machine can be used in a multicomputer mode by adding the software features for parallel computing.

MMS has been built using IBM-PC family of computing systems as a **computing element (CE).** The **communication interface** which connects these CEs is compatible with any member of this family.

Main features of communication interface are :

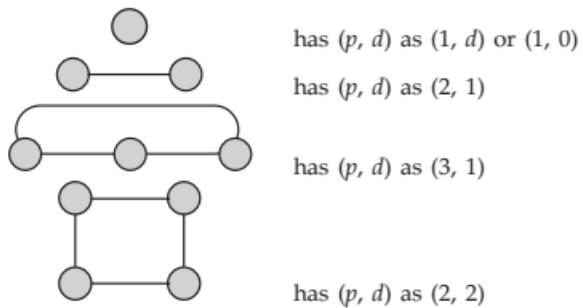
1. It **maintains the order** in which messages are sent and enforces synchronization of data interchange.
2. It is possible to create **multiple communication paths between CEs.**
3. The interface allows **many topologies** of the interconnection of CEs.
4. It is **simple and inexpensive.**
5. It takes only few bytes of the main memory of the CE.
6. The measured speed of communication is 75 kbytes/sec between CEs.

The communication interface has been built using a storage unit called FIFO (First-in-First-Out) buffer store. **Three bidirectional communication links are fabricated on one printed circuit board (PCB)** and plugged into a free expansion slot available in IBM PC compatible CE. The topology used in MMS architecture interconnects a number of CEs in each dimension as a fully connected network using bidirectional FIFOs. On this network, a CE can broadcast data to all other CEs connected to it. **Please note that it can also selectively broadcast (multicast) data to a smaller set of CEs.** MMS architecture has a number of such fully connected multicast networks which are replicated in multiple dimensions. These networks are independent of one another. These systems can be described by two parameters :

- (a) **Drop (p).** It is defined as the number of CEs which are fully connected by a single multicast network. It is denoted by symbol, p .
- (b) **Dimension (d).** It is defined as the number of fully connected multicast networks to which a CE is connected. It is denoted by symbol d . **Please note that an MMS structure with p -drops and d -dimensions has p^d CEs.**

Also note that each CE in MMS can be addressed uniquely using the (p, d) parameters.

For example :



We shall now draw a 9-CE/PE MMS architecture. It is shown in Fig. 3.32. below.

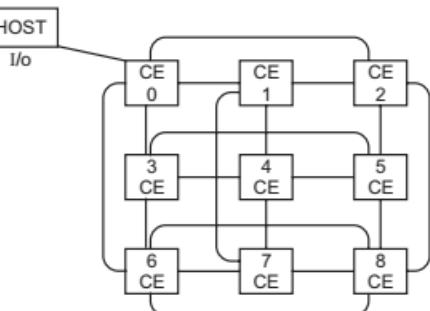


Fig. 3.32. 9-CE/9 PE MMS architecture.

As shown in Fig. 3.32, a **host computer** is connected to MMS. It is responsible for compiling a program and scheduling tasks to all CEs in the system. It also handles all I/O operations. It supports the main disk storage. A programmer interacts with the MMS through the host.

How to program an MMS ?

A programming environment is required. The environment allows an application programmer to view a communication link and the CE at its other end as a device. An application programmer can initiate a process in a remote computer and inter change data with it. It is possible to simultaneously initiate operations on many links and CEs connected to these links. This is done by using a **bit pattern called as a mask to address these CEs**. Some instructions that are used for its programming are as follows :

1. **Fork (Mask)** : This instruction initiates processes at all the CEs identified by Mask. The initiated processes inherit all the initiator's code, data and stack. All processes simultaneously execute the statements following the Fork call.
For example : FORK (001) will enable CE3.
2. **Terminate** : This is executed by the initiated processes at the end of execution.
3. **Send (Mask, variable name)** : This sends the contents of variable name to all CEs specified by Mask. If any link is full, control is returned to sender only after the receiver empties part of the FIFO.

For example : Send (010, x)

This instruction will send the contents of variable x to CE-2.

4. **Receive (Mask, variable name)** : This receives into the variable name, data sent by CEs specified by Mask and stores them.

For example : Receive (y , 010)

This instruction stores in y , the data sent by the Mask (010).

This system with 9 CEs/9 PEs has been used to run many applications programs.

3.7 COMPARISON OF PARALLEL COMPUTER ARCHITECTURES

Let us now compare in a tabular form, all the parallel architecture on the basis of type of parallelism used by them, task sizes, programming ease and scalability.

Features	Pipelined Computers	Array Processors	Shared Memory Multi-processors	Message Passing Multi-computer
1. Type of Parallelism used	Temporal parallelism	Data parallelism	MIMD	MIMD
2. Task sizes	Fine grain	Fine grain	Medium Grain	Medium Grain
3. Programming ease	Easy	Easy	Easy due to global address space.	Needs good task allocation & synchronization
4. Scalability	Good	Good	Poor	Very Good

SUMMARY

In this chapter, we have studied various network properties and topologies for the parallel computers. What makes a connection static and what makes it dynamic. The role of multiport memory and memory contentions have also been discussed. How can we organize our PEs to get different types of parallel computers ? The answer to all these questions have been provided. Also, we have discussed about the figures of merit for multi-processors and multi-computers. A general purpose parallel computer must thus be able to accept an arbitrary task graph and execute as many tasks as possible in parallel. Each task is a small program which is stored in a CE's private memory. We also studied in this chapter that message passing systems are scalable whereas shared memory machines are not. We need to program these computers using parallel programming languages.

MULTIPLE CHOICE QUESTIONS (MCQS) WITH ANSWERS

- The number of nodes in the graph is called as :
 - Network size
 - Graph
 - Grain
 - None of the above
- A node degree (d) is given by :
 - $d = \text{in-degree} - \text{out-degree}$
 - $d = \text{in-degree} * \text{out-degree}$
 - $d = \text{in-degree} + \text{out-degree}$
 - None of the above.
- Common data routing functions are :
 - Permutations
 - Broad cast
 - Multicast
 - All of the above.

ANSWERS

- | | | | | |
|---------|---------|---------|---------|---------|
| 1. (a) | 2. (c) | 3. (d) | 4. (c) | 5. (a) |
| 6. (b) | 7. (b) | 8. (a) | 9. (a) | 10. (b) |
| 11. (c) | 12. (a) | 13. (c) | 14. (a) | 15. (b) |
| 16. (a) | 17. (b) | 18. (a) | 19. (c) | 20. (b) |

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

1. Define the following terms :
 - (a) Node degree
 - (b) Network diameter
 - (c) Bisection bandwidth
 - (d) Static connection network
 - (e) Dynamic connection network
 - (f) Nonblocking networks
 - (g) Multicast and broadcast
 - (h) Mesh versus torus
 - (i) Symmetry in networks
 - (j) Multistage networks
 - (k) Crossbar networks
 - (l) Digital buses.

Ans. (a) The number of edges incident on a node is known as a **node degree**.
(b) The maximum number of distinct links between any two nodes is known as **network diameter**.
(c) The minimum number of edges that must be removed in order to divide the network into two halves
(d) Static networks are formed of point-to-point direct connections which will not change during program execution.
(e) Dynamic networks are implemented with switched channels which are dynamically configured to match the communication demand in user programs.

- (f) A network which can handle all possible connections without blocking is called as a non-blocking network.
- (g) Multi-cast corresponds to a mapping from many-to-many whereas Broadcast refers to one-to-all mapping.
- (h) Mesh and torus topologies are different at the boundary nodes only. So, their network diameters are different.
- (i) A network is symmetric if the topology is the same looking from any node in the network.
- (j) A multi-stage network consists of multiple stages of interconnected switches. A number of switches are used in each stage. Interstage connection patterns are specified between the switches in adjacent stages. The switches can be dynamically set to establish connections between different input and output pairs.
- (k) A crossbar network is a single-stage network. It provides a dynamic connection between any source and any destination without blocking. An $(n \times m)$ crossbar network requires $(n \times m)$ crosspoint switches.
- (l) A digital bus is a collection of wires and connectors for data transaction among PEs, memories and I/O devices attached to the bus. One device can use the bus at a given time. When multiple requests arrive, we can use a bus arbitration logic to allocate or deallocate the bus.
2. Consider an (8×8) Illiac mesh, a binary hypercube and a barrel shifter with 64 nodes each ($N_0 \dots N_{63}$). All network links are bidirectional.
- (a) List all nodes reachable from node N_0 in three steps for each of the three networks.
- (b) What is the upper bound on the minimum number of routing steps needed to send data from any node N_i to another node N_j .
- (c) Repeat part (b) for a larger network with 1024 nodes.

Ans. (a) For an Illiac mesh (8×8) , we use this equation :

$$(a + b + c) \bmod 64$$

Where a , b and c can be $+1$, -1 , $+8$, or -8 .

If a , b and c are all different then 4 combinations are possible.

If two of them are equal then 12 combinations are possible.

If $a = b = c$ then 4 combination are possible.

∴ Totally, 20 combination are possible. However, 8 of the combinations contain the pair $+1$ and -1 or the pair $+8$ and -8 , making them reachable in one step. Such nodes have to be eliminated from the list. Hence, 12 nodes can be reached from N_0 in three steps. The addresses of these nodes are 3, 6, 10, 15, 17, 24, 40, 47, 49, 54, 58 and 61.

For a binary 6-cube, the binary address $a_5 a_4 a_3 a_2 a_1 a_0$ of a node reachable in three steps from N_0 has exactly three 1s. There are

$$\begin{aligned} C(6, 3) &= \frac{6!}{3!(6-3)!} \\ &= \frac{6!}{3!3!} \\ &= \frac{6 \times 5 \times 4 \times 3 \times 2 \times 1}{3 \times 2 \times 1 \times 3 \times 2 \times 1} \end{aligned}$$

= 20 possible combination

The addresses of these nodes are 7, 11, 13, 14, 19, 21, 22, 25, 26, 28, 35, 37, 38, 41, 42, 44, 49, 50, 52 and 56.

For a barrel shifter, we firstly list all 6-bit numbers which contain three 1s. There are 20 such numbers. We take 1s complement of each number and then add 1 to each of the resulting numbers. If a new number has three or four 1s in its binary representation and the 1s are separated by atleast one 0 then both the nodes whose addresses are the original number and the new number can be reached in exactly three steps. The addresses of these nodes are 11, 13, 19, 21, 22, 23, 25, 26, 27, 29, 35, 37, 38, 39, 41, 42, 43, 45, 51 and 53.

(b) The upper bound on the minimum number of routing steps needed to send data from any node to another for an 8×8 Illiac is given by $(\sqrt{64} - 1) = 8 - 1 = 7$.

For a 6-cube, upper bound is 6 and for a 64-node barrel shifter, it is :
 $\log_2 64/2 = 3$

(c) The upper bound on the minimum number of routing steps needed to sent data from any node to another is 31 for a (32×32) Illiac mesh, 10 for a 10-cube and 5 for a 1024-node barrel shifter.

3. Compare buses, crossbar switches and multistage networks for building a multi-processor system with n PEs and m shared-memory modules. If w represents a word-length; Compare on the basis of

- (a) Minimum latency for unit data transfer,
- (b) Bandwidth per processor.
- (c) Wiring complexity.
- (d) Switching complexity.
- (e) Connectivity and routing capability.
- (f) Remarks ?

Ans.

Network Characteristics	Bus System	Multistage Network	Crossbar Switch
Minimum latency for unit data transfer	Constant	$O(\log_k n)$	Constant
Bandwidth per processors	$O(w/n)$ to $O(w)$	$O(w)$ to $O(nw)$	$O(w)$ to $O(nw)$
Wiring Complexity	$O(w)$	$O(nw \log_k n)$	$O(n^2 w)$
Switching Complexity	$O(n)$	$O(n \log_k n)$	$O(n^2)$
Connectivity and routing capability	Only one-to-one at a time.	Some permutations ; broadcast if network is unblocked	All permutations one at a time.

Remarks	Say, n PEs are on the bus ; w bits is bus-width	$(n \times n)$ MIN using $K \times K$ switches with line width of W -bits	Assume $(n \times n)$ crossbar with a line width of W -bits
---------	---	---	---

4. Answer the following questions related to multistage networks :
- How many legitimate states are there in a (4×4) switch module including both broadcast and permutation ? Justify your answer with reasoning.
 - Construct a 64-input Omega network using (4×4) switch modules in multiple stages. How many permutations can be implemented directly in a single pass through the network without blocking.
 - What is the percentage of one-pass permutations compared with the total number of permutations achievable in one or more passes through the network ? [GGSIPU, B. Tech 7th Sem; 2007 & M. Tech 2003 & UPTU, B. Tech (CSE) 8th Sem.; 2003-04]

- Ans. (a) For each output terminal, there are 4 possible connections ; one from each of the input terminals so that there are $4 \times 4 \times 4 \times 4 = 256$ legitimate states.
- (b) To construct a 64-input Ω network, we need $16 \times 3 = 48$ switch modules of size 4×4 . There are $41 = 4 \times 3 \times 2 \times 1 = 24$ permutation connections in a 4×4 switch module. So, a total of (24^{48}) permutations can be implemented in a single pass through the network without blocking.
- (c) The total number of permutations of 64 inputs is $64!$. So, the fraction is given by :

$$\text{Fraction} = \frac{24^{48}}{64!} \approx 1.4 \times 10^{-23}$$

5. Consider a K -ary, n -cube network and answer the following questions :
- How many nodes are there ?
 - What is the network diameter ?
 - What is the bisection bandwidth ?
 - What is the node degree ?
 - Explain the graph theoretic relationship among K -ary, n -cubes network and rings, meshes torus, binary n -cubes and Ω networks.
 - What is the difference between a conventional torus and a folded torus ?

[GGSIPU, M. Tech (IT), 1st Sem., Dec 2008]

- Ans. (a) K^n
 (b) $n[K/2]$
 (c) $2K^{n-1}$
 (d) $2n$
 (e) (i) A K -ary 1-cube is a ring with K -nodes.
 (ii) A K -ary 2-cube is a $2D (K \times K)$ torus.
 (iii) A mesh is a torus without end-around connections.
 (iv) A 2-ary n -cube is a binary n -cube.

- (v) An Ω network is the multistage network implementation of shuffle-exchange network. Its switch modules can be repositioned to have the same interconnection topology as a binary n -cube.
- (f) The conventional torus has long end-around connections but the folded torus has equal-length connections.
6. Explain the advantages of using binary fat trees over conventional binary trees as a multi-processor interconnection network ?

Ans. In a tree network, a message going from PE_i to PE_j goes up the tree to their least common ancestor and then back down according to the LSBs of j . Message traffic through lower-level (closer to the root) nodes is heavier than that of higher-level nodes. The lower level channels in a fat tree has a greater number of wires and hence a higher bandwidth. This will prevent congestion in the lower-level channels.

7. Define what is a universal fat tree ? Also prove that the capacities of a universal fat tree grows exponentially as we go up the tree from the leaves.

Ans. A universal fat tree is defined as a fat tree of n -nodes with the root capacity of w , where $n^{2/3} \leq w \leq n$ and for each channel, C_k , at level K of the tree, the capacity is given by the formula :

$$C_k = \min (\lceil n / 2^k \rceil, \lceil w / 2^{2k/3} \rceil)$$

Case 1 : If $K > 3 \log(n/w)$:

$$\text{Then } \lceil n / 2^k \rceil < \lceil w / 2^{2k/3} \rceil$$

$$\begin{aligned} \therefore C_k &= \lceil n / 2^k \rceil = (n+1) / 2^k \\ &= 1, 2, 4, \dots \text{ for} \\ K &= \log(n+1), \log(n+1)-1, \log(n+1)-2, \dots \end{aligned}$$

Case 2 : If $K \leq 3 \log(n/w)$:

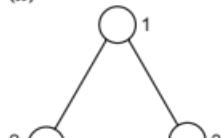
$$\text{Then } \lceil n / 2^k \rceil \geq \lceil w / 2^{2k/3} \rceil$$

$$\begin{aligned} \therefore C_k &= \lceil w / 2^{2k/3} \rceil \text{ which is } w/8^{2/3}, w/4^{2/3}, w/2^{2/3}, \text{ for} \\ K &= \dots, 3, 2, 1 \end{aligned}$$

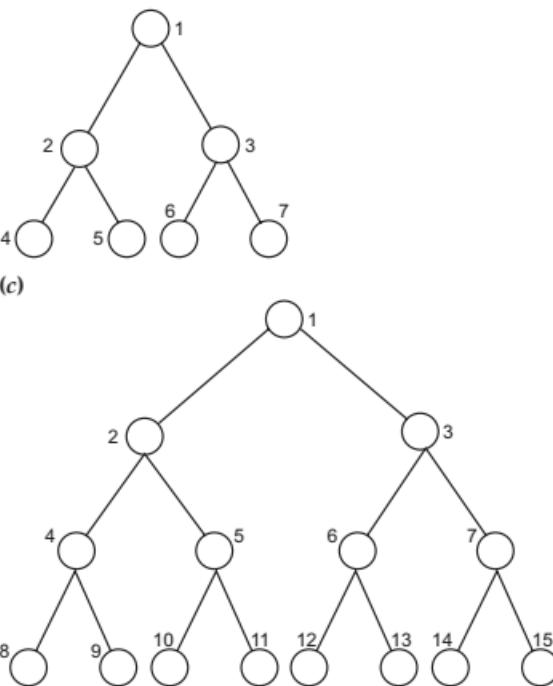
Please note here that initially the capacities double from one level to the next towards the root. However, at levels less than $3 \log(n/w)$ away from the root, the channel capacities grow at the rate of $\sqrt[3]{4}$. Hence proved.

8. Convert the following binary trees (balanced) into a mesh form using embedding technique :

(a)



(b)



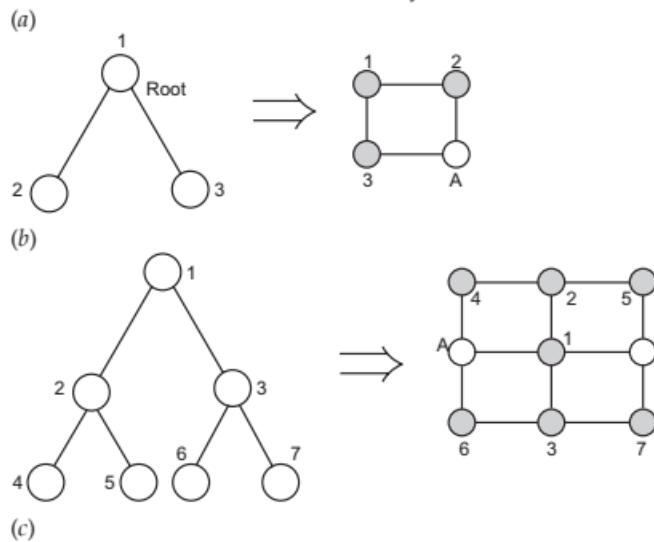
Ans. The procedure that we follow for an **embedding process** is as follows :

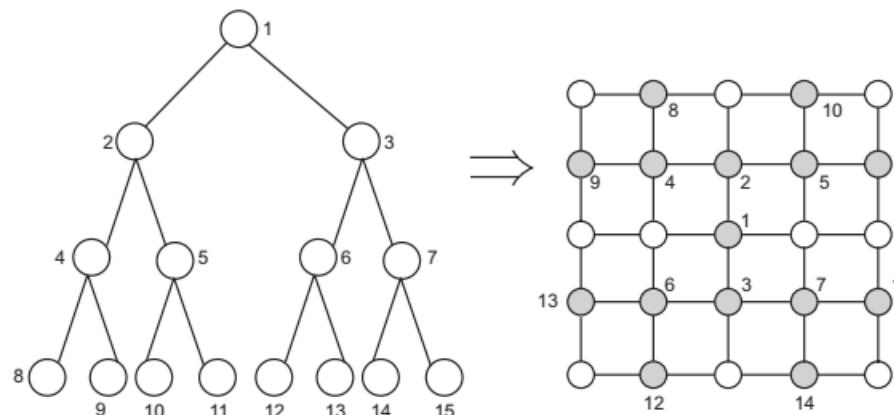
Step 1 : The root of a tree is mapped to the center node of a mesh.

Step 2 : The leaf nodes are mapped to the outlying mesh nodes.

Step 3 : Auxillary nodes (A) may be added to complete the mesh. The process is recursive.

So, we draw their mesh networks one by one :





9. Say, two floating point numbers are to be added. Let the two numbers be a_1 and b_1 . Their floating point representation is as follows :

$$\begin{aligned} a_1 &= (\text{mant } a_1, \text{ exp } a_1) \\ b_1 &= (\text{mant } b_1, \text{ exp } b_1) \end{aligned}$$

It takes T units of time to execute each task. What are the four steps that we follow to add these numbers in parallel ?

Ans. We need to divide this problem into four tasks that can be run on a pipelined computer. They are :

Task 1 (PE₁) : Compare exp a_1 with exp b_1 . Find the larger exponent and the difference between the exponents. Let the difference be d .

Task 2 (PE₂) : Right shift the mantissa of the operand with the smaller exponent by d bits.

Task 3 (PE₃) : Add the shifted mantissa to the other mantissa. Let the result mantissa be q .

Task 4 (PE₄) : Count the leading zeros, if any, in q . Let it be p . Shift q left by p bits. Adjust the result exponent.

10. Say, we need to replace a sequence of values $(x_{n+2}, x_{n+1}, \dots, x_1)$ in a memory by a new sequence $(y_n, y_{n+1}, \dots, y_1)$ where

$$y_1 \leftarrow w_1 x_3 + w_2 x_2 + w_3 x_1$$

$$y_2 \leftarrow w_1 x_4 + w_2 x_3 + w_3 x_2$$

$$y_3 \leftarrow w_1 x_5 + w_2 x_4 + w_3 x_3$$

⋮

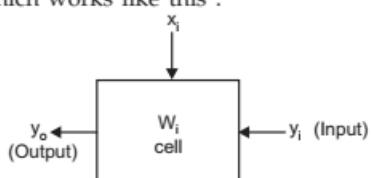
⋮

⋮

$$y_n \leftarrow w_1 x_{n+2} + w_2 x_{n+1} + w_3 x_n$$

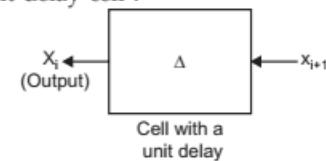
How can you perform such an operation using systolic arrays ?

Ans. Consider a cell which works like this :

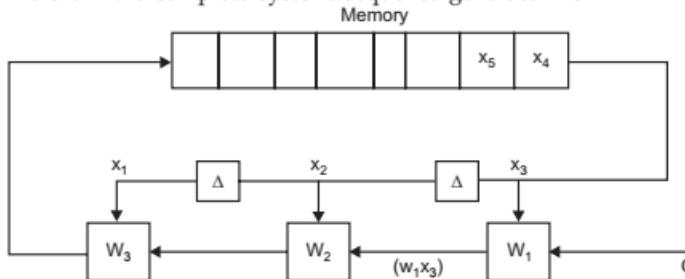


$$\& y_0 = y_i + w_i x_i$$

Also, consider a unit delay cell :



So, we draw the complete systolic sequence generator now :

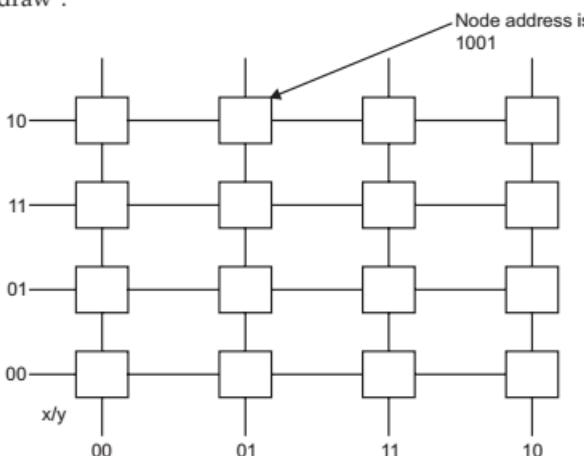


Herein, the contents of the serial memory are shifted by one position to the right and pushed into the PEs at each clock interval. The MSB of this memory is pushed to right and moved to w_1 , w_2 and w_3 at each clock interval and with a unit delay (Δ).

11. How will you embed a mesh into a hypercube ?

Ans. A mesh/torus can be embedded into a hypercube by numbering the nodes of the mesh or torus. Each node has an address that is composed of an x -coordinate and a y -coordinate. Each coordinate is labelled in a Gray Code. In this Gray Code, adjacent numbers differ by one bit.

So, we draw :



Please note that the numbers adjacent to node 10 are 00 and 11 nodes. Both of these differ by one bit position from 10. Such a mesh will perfectly embed into a hypercube.

12. There are two basic ways of message transmission – circuit and packet switching. Tabulate the difference between them.

Ans.

Circuit Switching	Packet Switching
<ul style="list-style-type: none"> 1. It has a dedicated copper path. 2. Fixed Bandwidth available. 3. No store-and-forward transmission. 4. Each packet follows the same route. 5. Regular call setup is needed. 6. Charging is needed on per minute basis. 7. Data is not sent as packets. 8. Resources are reserved in the network. <p><i>For example :</i> Circuit switching is used in telephone networks/system. The network resources are reserved. When you place the call and released when you hang up.</p>	<ul style="list-style-type: none"> 1. There is no such path. 2. Dynamic Bandwidth available. 3. Works on store and forward transmission. 4. Each packet does not follow the same route. 5. Regular call setup is not needed. 6. Charging is needed on per packet basis. 7. Data is sent as packets. 8. Resources may or may not be reserved in the network. <p><i>For example :</i> Internet is packet switched. The network resources are used only during the time it takes to transmit each packet.</p>

13. What is a MPP ? Explain its architecture in detail. What are its image processing applications ? [KUD, BE (CSE) 8th sem., 1997]

Ans. Any machine having hundreds or thousands of processors operating in parallel is called as MPP or massively parallel processors. A large SIMD computer has been developed for satellite imagery applications also. It is so called as it has (128×128) active PEs.

As the name suggests, it is a large-scale computer system, which uses large number of processors. Earlier NASA Goddard Flight Centre used MPP for the processing of Satellite imagery. For this application, MPP was constructed with 16384 microprocessors. Each microprocessor was a bit slice processor capable of performing fast bit slice operations. Now, let us study the architecture and important features of MPP.

MPP Architecture : Fig.(a) shows the architecture of MPP. It consists of hundreds or thousands of processors. It is designed mainly for parallel computing. There are multiple nodes/subsystems in the architecture. Each node consists of uniprocessor or multiprocessors, private caches, local memory and local disks. All these units are interconnected by local interconnection network within the node. But all nodes in MPP are interconnected by high speed communication network. For this each node has a network interface through which it connects to high speed network. The system has distributed memory. Internode communication takes place through message passing mechanism.

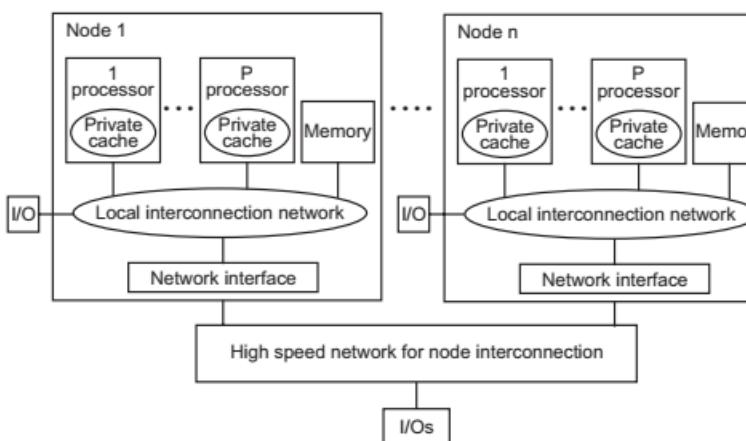


Fig. (a) MPP Architecture.

Note : MPP = Many PEs + Distributed Memory + Network communication

The biggest advantage of MPP is that it has a highly scalable architecture. Scalable design helps to improve the performance/cost ratio in terms of parallel computing. **Scalability means scaling the existing system by adding more processors or other resources such as memory, disks etc.** Upgraded processors can be used in scaling the system. Thus, MPP becomes the large size system which provides large memory and input/output capabilities.

Features of MPP :

- The MPP architecture is designed not only for scientific supercomputing but also for industrial, commercial and business applications.
- The MPP architecture is highly scalable not only in terms processors but also other resources.
- It has distributed memory which facilitates higher scalability. High speed memories can be used in scaling to enhance the performance.
- It uses distributed operating system.
- It supports standard programming models and standard programming languages such as FORTRAN, C etc.
- It supports for asynchronous MIMD mode in which processes interact by message passing.
- It offers flexibility in scaling the components to reduce the overall system cost.

Applications of MPP :

It has large number of scientific, engineering, commercial and industrial applications. It is used in :

- Satellite imagery processing such as feature extraction, pattern classification and real time scene analysis.
- Space science for the study of black hole and solar activity.
- Biomedical field, for the research on living cells and tissues.

- Medical diagnostics for the research on DNA sequencing, understanding human joint mechanisms through advanced computational models.
- Environmental monitoring for simulation of large scale high resolution ecosystem model.
- Product of design for simulation of material properties.
- Data warehousing, decision support systems, engineering simulation, digital signal processing and many more applications.

Some image processing applications of MPP are :

1. Real-time scene analysis

Real-time images can be trapped for scene analysis.

2. Image Segmentation

The segmentation of images into coherent areas of significance can be done.

14. Write short notes on Chip multiprocessing.

Ans. Chip multiprocessor consists of multiple processor cores on a single VLSI Chip. It uses symmetric multiprocessing and multiprocessor OS to exploit thread level parallelism. Multiple applications can be run on multiprocessor cores. Software should be designed accordingly.

For example : A dual core processor. It is shown in figure-a below. Other examples are IBM Power 4, IBM Power 5, Ultra SPARC IV etc.

Advantages of chip multiprocessor are :

- One silicon chip can accommodate more than one processor cores on the same space due to advanced VLSI technology. This reduces the power requirement and also the cost.
- As the multiprocessors are on the same die, the communication becomes fast due to shorter distances.
- The performance of the chip multiprocessor is better than single highly complex processors.
- It supports for thread level parallelism. Multiple applications can be executed efficiently by multithreading.

15. What are wave front arrays? How are they different from systolic arrays?

Ans. More or less both wave front arrays and systolic arrays are similar but there exist some difference too—

- Wave front arrays *do not have a global pulse* used to synchronise all PEs. They pass intermediate results (or data items) whenever they are ready. The receiving PE then confirms by returning an acknowledgment signal called as *handshaking signal*. Hence, they are said to be self-synchronising or *asynchronous*.

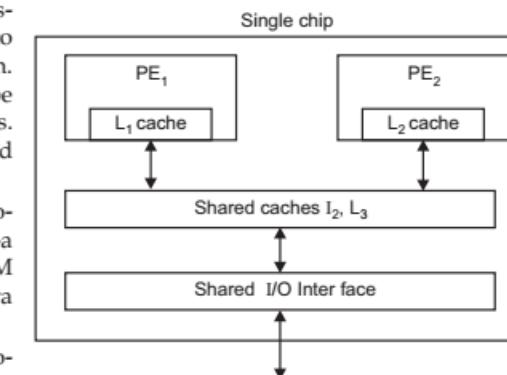


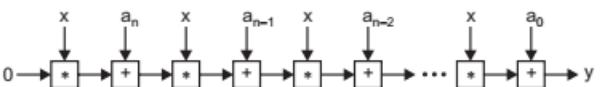
Fig. (a) Chip Multiprocessor.

- Wave front arrays can be more generally applied than systolic arrays because they allow a multiple or variable number of instructions to be performed at each stage.
∴ Wave front = Systolic data pipelining + Asynchronous data flow execution.

$y = a_0 + a_1x + a_2x^2 + \dots + a_kx^k$ Horner's rule is—

$$y = (((((a_n x + a_{n-1}) \times x + a_{n-2}) \times x + a_{n-3}) \times x \dots a_1) \times x + a_0$$

Ans. A linear systolic array in which the processors are arranged in pairs, can be used to evaluate a polynomial using Horner's Rule. This is shown in Fig.



Herin, one processor multiplies its input by x and passes the result to the right. The next PE adds a_j and passes the result to the right. After an initial latency of $2n$ cycles to get started, a polynomial is computed in every cycle.

EXERCISE QUESTIONS

- Mention the number of legitimate state and permutation connections for $n \times n$ and 8×8 switch modules. [GGSIPU ; B.Tech. 7th sem ; 2007]
 - Draw a 3×3 mesh, Illiac and torus network. [GGSIPU ; B.Tech. 7th sem ; 2007]
 - Compare K-ary, n -cube, hypercube and star network type based on the following static network characteristics :
 - Node degree
 - Network diameter
 - No. of links
 - Bisection width
 - Symmetry.
[GGSIPU ; B.Tech. 7th sem ; 2007]
 - Is Omega network-blocking or a non-blocking network ? What is the difference between blocking and non-blocking network ? [GGSIPU ; B.Tech. 7th sem ; 2004]
 - Answer the following questions related to the multistage networks :
 - How many legitimate states are there in a 4×4 switch module, including both broadcast and permutation ? Justify with reasons.
 - Construct a 64-input Ω network using 4×4 switch modules in multiple stages. How many permutations can be implemented directly in a single pass through the network without blocking ?
 - What is the percentage of one-pass permutations compared with total number of permutation achievable in one or more passes through the network.
[GGSIPU ; M.Tech. 1st sem ; 2003]
 - (a) Prove that the number of legitimate states in a $K \times K$ switch module equals K^k .
 (b) Determine the percentage of permutation that can be realized in one pass through a 64-input Ω network built with (2×2) switch modules. [GGSIPU ; M.Tech. 1st sem ; 2003]

7. Write short notes one (any two) :
 - (a) Omega network.
 - (b) Crossbar network.
 - (c) Hierarchical Bus System.

[GGSIPU ; M.Tech. 1st sem ; 2003]
8. (a) Define the following terms :
 - (i) Digital Bus
 - (ii) Network diameter.
 - (iii) Non-blocking network.
 - (iv) Multistage network.
- (b) Differentiate between static and dynamic connection networks, give a few examples.
[GGSIPU ; M.Tech. 1st sem ; 2003]
9. (a) With the help of diagrams, discuss chordal ring, barrel shifter, mesh and torus inter connection networks with respect to node degree, network diameter and bisection width.
(b) With the help of diagram, explain the switching process in a (16×16) Ω network.
[GGSIPU ; M.Tech. 1st sem ; 2004]
10. Define an array processor ? Explain its working. Give examples for array processors.
[KUD ; B.E. (CSE) ; 8th sem ; 1996]
11. (a) Explain the fundamental decisions which determine appropriate architecture of an inter-connection network.
(b) Explain the Illiacs mesh network and Barrel shifter network for $N = 16$. Compare their routing functions.
[KUD ; B.E. (CSE) ; 8th sem ; 1996 & 97]
12. Write short notes on :
 - (a) 'Systolic array structures.'
 - [IPTU, B. Tech (CSE) 8th Sem.; 2008-09]
 - (b) Embedding.
 - & KUD ; B.E. (CSE) ; 8th sem ; 1996 & 97]
13. Draw a single-stage recirculating mesh connected network for $N = 16$ PEs. Give the four permutations $R + 1, R - 1, R + 4, R - 4$ and the upper bound on steps to route data from PE_i to PE_j ?
[KUD ; B.E. (CSE) ; 8th sem ; 1996]
14. Describe multistage network by the following characterising features :
 - (a) Switch Box
 - (b) Network topology
 - (c) Control structure.
[KUD ; B.E. (CSE) ; 8th sem ; 1997]
15. (a) List the various processor organization methods of connecting processors in parallel computer. Discuss any two methods with an example in details.
(b) What are the main important properties of the communication interface designed for MMS ? And what are the basic instructions necessary for programming MMS ?
[KUD ; B.E. (CSE) ; 8th sem ; 1996]
16. Write short notes on :
 - (a) Static and Dynamic Networks.
[DU-DCE ; M.E. (CSE) ; 2003]
17. What is the significance of bisection bandwidth ? What is the bisection bandwidth of a 3-stage Ω network ?
[DU-DCE ; M.E. (CSE) ; 2003]
18. What are the various interconnection networks used in SIMD system ? Give salient features of each.
[DU-DCE ; M.E. (CSE) ; 2002]
19. Give the bisection width of the following networks : (assume $N = 2^k$).
 - (a) Mesh Network
 - (b) Hypercube Network
 - (c) Shuffle-Exchange Network
[DU-DCE ; M.E. (CSE) ; 2003]
20. How do static and dynamic topologies of interconnection differ ? What are the factors affecting the performance of interconnection network ? Describe various

static network topologies in terms of network parameters and their merits in relation to communication and scalability.

[UPTU, B. Tech (CSE) 8th Sem.; 2004-05]

[Pune Univ., BE (CSE), 2nd sem., May 2001, 2002, May 2003, 2005]

21. Compare three multiprocessor hardware interconnection organization structures namely time shared bus, crossbar switch and multiport memory with respect to the following features/points.
 - (i) System cost and hardware complexity.
 - (ii) Modification of hardware system configuration.
 - (iii) Transfer rate.
 - (iv) Attainable system efficiency. [Pune Univ., BE (CSE), 2nd sem., May 2006]
22. (a) Draw a 16 input omega network using (2×2) switches as building blocks. Show the switch setting for routing a message from node 1011 to node 0101 and from node 0111 to node 1001 simultaneously. Does blocking exist in this case?
(b) Describe a multiport memory without priority assignment
[GGSIPU., B. Tech (CSE), 7th sem., Dec. 2011]

CHAPTER

4 TYPES OF PROCESSORS

4.0 INTRODUCTION

Powerful General purpose microprocessors may be divided into two groups *i.e.*,

- (i) Pure complex instruction set computer (CISC) processors.
- (ii) Pure reduced instruction set computer (RISC) processors.

Examples of CISC processors are the 8086, 80186, 80286 and 80386 belonging to INTEL family and 68000 and 68040 from MOTOROLA family.

Please note that 80386 is the last pure CISC model of the Intel family because 80486 has the facility of RISC technology partially.

The CISC microprocessors have :

- Complex machine instructions.
- Micro-encoding of machine instructions.
- Best addressing capability.
- Smaller number of useful registers.

Reduced Instruction Set Computer (RISC) have faster clock rates, ranging from 20 to 120 MHz. For example: Intel i860, SPARC, IBM RS/6000 etc. are some of the RISC processors.

In this chapter, we shall study all these advanced processors including super scalar processors, very long instruction word (VLIW) and vector processors.

4.1 ADVANCED PROCESSOR TECHNOLOGY

By the year 2010 (the year of common wealth games!!!), a single VLSI chip will have more than 1 billion transistors.

Pipelining has been used in RISC machines. RISC processors have been succeeded by super scalar processors that execute multiple instructions in one clock cycle. Very Large Instruction word (VLIW) processors are those in which one instruction word encodes more than one operation. As per the Moore's Law –“The number of transistors which can be integrated in a chip has been doubling every 18 months”. To enhance the speed of PEs we now make use of multithreading, VLIW and vector processors. What we actually observe is that as technology evolves rapidly, the clock rates of PEs also slowly move from lower to higher speeds. We shall also study about the super pipelined processors. These computers use multiphase clocks with clock rate ranging from 100 to 500 MHz. Vector computers (to be discussed) use this approach. We should also understand that a scalar processor is different from a vector processor. This is shown in Fig. 4.1 below. Also shown in Fig. 4.2 is the working of a vector processor. What we observe from Fig. 4.1 and Fig. 4.2 is that in case of a scalar processor – one-result/many clock cycles is produced while in case of vector processors one result/clock is produced.

SCALAR VS VECTOR PROCESSING

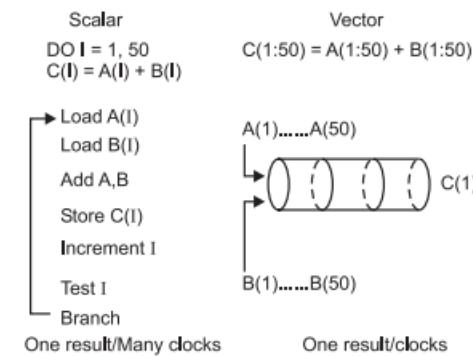


Fig. 4.1. Scalar V/s Vector Processing.

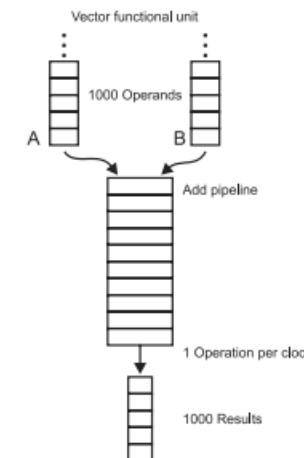


Fig. 4.2. Vector Processing.

Please note that a base scalar processor is defined as a system in which one instruction is issued per cycle. There is one-cycle latency (delay) for a simple operation and a one-cycle latency (delay) between the instruction issues.

As we already know that the execution of any instruction requires four phases:

Fetch \longrightarrow decode \longrightarrow Execute \longrightarrow write-back (results)
 (F) (D) (E) (W)

Further assuming that we have an instruction pipeline (like an assembly line production) which receives successive instructions from its input side and it executes them in an overlapped fashion. Then, we can have three different cases.

Case 1 : Execution in a base scalar processor

Herein, if I draw a graph showing its execution of instruction then we get the following:

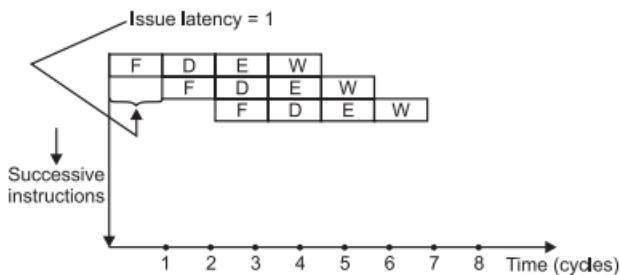


Fig. 4.3. A base scalar processor showing pipelining.

So, here in Fig. 4.3 we find that the pipeline utilization can be improved if successive instructions can execute at the rate of one per cycle.

Case 2 : Pipeline underutilization

Can't we have an issue-latency of more than one? Yes, why not. Say, the issue latency is 2 now. So, we draw Fig. 4.4. Now.

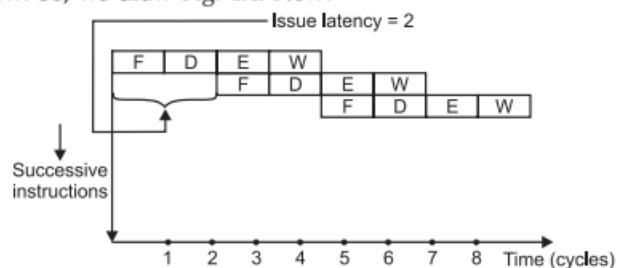


Fig. 4.4. Pipeline underutilization with 2 issue latency.

It is observed from figure 4.4 that if the issue-latency between two instructions is increased (from 1 to 2) then the pipeline can be underutilized.

Case 3 : Poor pipeline utilization

It is also possible to combine two pipeline stages into one stage. So, we draw another Fig. 4.5 now.

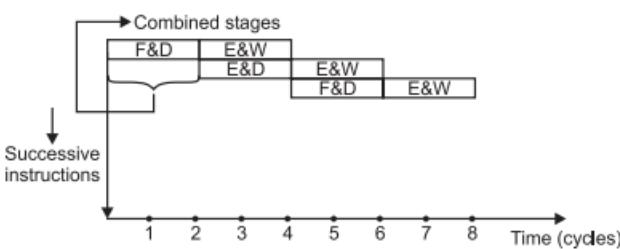


Fig. 4.5. Poor pipeline utilization.

Herein, 2 phases i.e., Fetch (F) and Decode (D) have been merged. Similarly, Execute (E) and write-back results (W) phases have been merged. But this also results in poor pipeline utilization.

Please note here that the effective CPI for case I is 1 (ideal pipeline). Whereas for case II, it is 2 and in case III, it is $\frac{1}{2}$ (half) only. What we observe here is that the pipeline utilization is falling down only and not increasing.

4.2 INSTRUCTION-SET ARCHITECTURES

One of the important aspects of Computer architecture is the design of the instruction set for a processor. It is the instruction set chosen for a particular computer that determines the way in which machine language program are constructed. A computer with a large number of instructions is called as a complex instruction set computer (CISC). A computer that uses a few instructions with simple constructs are called as Reduced instruction set computer (RISC). These instructions can be executed at a faster rate within the CPU with minimum memory references

4.3 CISC SCALAR PROCESSOR

A scalar processor will execute with a scalar data. Today's scalar processor have both integer and floating point unit within single CPU. This is shown in Fig. 4.6 below.

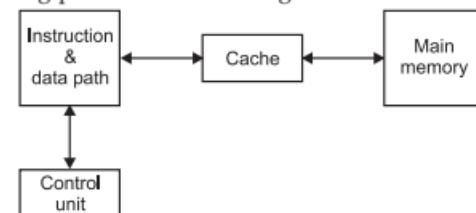


Fig 4.6 A. CISC architecture.

We can use a single chip or many chips to construct a CISC scalar processor. Please note that ideally, a CISC scalar processor should have a performance equal to that of a base scalar processor (discussed earlier). However, in CISC machines under pipelining may occur due to data dependence, resource conflicts, branches, jumps and other reasons.

For example : VAX 8600, i486, M68040 are CISC processors. Others include 8086, 80186/286/386, M68000 etc.

By convention—CISC architecture (Fig. 4.6) uses a **unified cache**. A unified cache holds both instructions and data. So, same bus is used.

4.4 RISC SCALAR PROCESSORS

These processors issue one instructions per cycle. Some of the features of RISC are—

- (a) Reduced instruction set.
- (b) Simpler instructions.
- (c) Hardwired control unit.
- (d) Few addressing modes.
- (e) Instruction pipelining.

For example : Intel i860, M88100, AMD 29000 are some RISC processors.

A RISC architecture is shown in Fig. 4.7

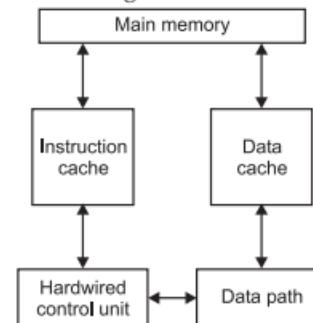
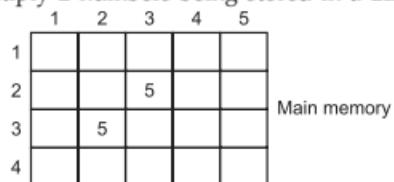


Fig. 4.7. A RISC architecture.

In RISC machines, we have separate instruction (I) and data (D) caches.

Say, we want to multiply 2 numbers being stored in a 2D-memory



At memory location (2,3) = 5 is stored.

At memory location (3,2) = 5 is stored.

Then, **CISC machine** will multiply them in one instruction as follows :

MULT 2 : 3, 3 : 2

Whereas **RISC machine** will use many instructions as follows :

```

Load A, 2 : 3      ; Get First number
Load B, 3 : 2      ; Get Second number
Prod A, B          ; Multiply them i.e., A ← A * B
Store 2 : 3, A     ; Move results from A to
                    ; mem. Loc. 2 : 3
  
```

In CISC approach, very little RAM is required to store instruction. The emphasis is on building complex instructions directly into the hardware. But in RISC approach, more RAM is required.

4.5 COMPARISON OF CISC AND RISC IN TABULAR FORM

CISC	RISC
1. These computers use large set of instructions.	1. These computers use small set of instructions.
2. They support 12–24 addressing modes.	2. They support only 3–5 addressing mode.
3. They use 8–24 general purpose registers (GPRS).	3. They use 32–192 GPRs.
4. They mostly have a unified cache.	4. They use separate (split) data and instruction cache.

5. Clock rates are between 33–50 MHz.	5. Clock rates are between 50–150 MHz.
6. CPI is between 2 and 15.	6. Average CPI is less than 1.5.
7. It uses micro coded control logic.	7. It uses hardwired control logic.
8. They use variable format instructions i.e., 16–64 bits per instruction.	8. They use fixed instruction format i.e., 32 bits per instruction.
Examples : 8086, 186, 286, 386	Examples : SUN SPARC, i860 etc.

Note : Pentium has features of CISC and RISC, both.

4.6 SUPERSCALAR PROCESSORS

A processor in which many instructions are executed concurrently is known as a super scalar processor. These processors exploit more instruction level parallelism (ILP) in user programs. The instruction issue degree (m) is limited by this formula :

$$2 < m < 5$$

These processors can use pipelining with three instructions per cycle also. In general, a super scalar processor of degree m can issue m instruction per cycle. So, a base scalar processor (RISC or CISC) has $m = 1$.

We show a super scalar processor of degree, $m = 3$ in Fig. 4.8

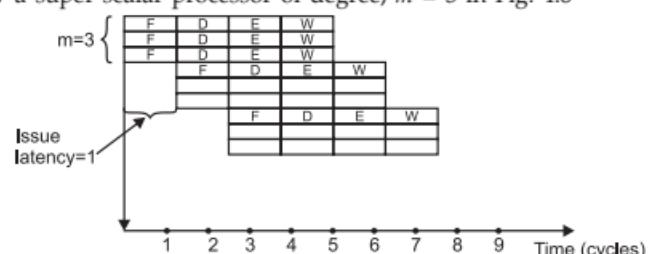


Fig. 4.8. A Super scalar with degree (m) = 3.

That is, three instructions are executed per cycle. In general, if we want to fully utilize a super scalar processor of degree m then m instructions must be executed in parallel. In some clock cycles this may not be true. In that case, some of the pipelines may be stalling in a wait state. In a super scalar processor, the issue latency is 1 as in case of a base scalar processor. Please note that super scalars require a high degree of ILP in programs so it depends more on an optimizing compiler to exploit parallelism. A super scalar machine can issue a fixed-point, floating point, load and branch, all in one cycle also. So, here $m = 4$; ($2 < m < 5$). Here, the parallelism achieved will be the same as in a vector processor. Please remember the following points :-

1. A scalar machine is able to perform only one arithmetic operation at once. A superscalar architecture (SPA) is able to fetch, decode, execute, and store results of several instructions at the same time. It does so by transforming a static and sequential instruction stream into a dynamic and parallel one, in order to execute a number of instructions simultaneously.

2. Upon completion, the SPA reinforces the original sequential instruction stream such that instructions can be completed in the original order. In an SPA instruction, processing consists of the fetch, decode, issue, and commit stages. During the fetch stage, multiple instructions are fetched simultaneously.

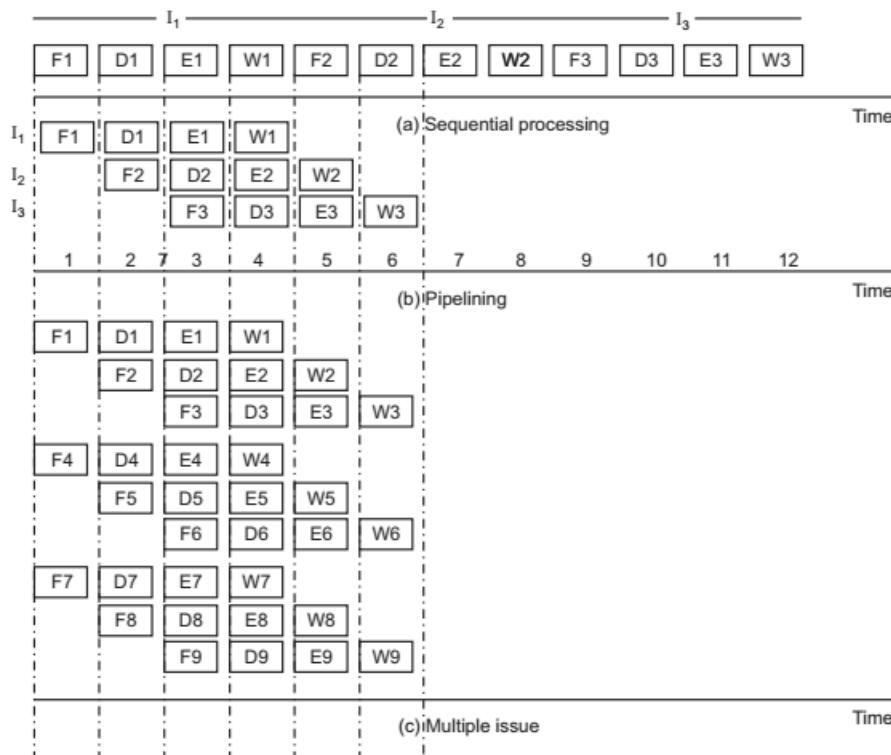


Fig. 4.9. Multiple issue versus pipelining versus sequential processing

3. Branch prediction and speculative execution are also performed during the fetch stage. This is done in order to keep on fetching instructions beyond branch and jump instructions. **Decoding** is done in two steps. **Preddecoding** is performed between the main memory and the cache and is responsible for identifying branch instructions. **Actual decoding** is used to determine the following for each instruction:

- (1) The operation to be performed;
- (2) The location of the operands; and
- (3) The location where the results are to be stored.

4. During the issue stage, those instructions among the dispatched ones that can start execution are identified. During the commit stage, generated values/results are written into their destination registers. The most crucial step in processing instructions in SPA is the dependency analysis. The complexity of such analysis grows quadratically with the instruction word size.

5. This puts a limit on the degree of parallelism that can be achieved with SPAs such that degree of parallelism higher than four will be impractical. Beyond this limit, the dependence analysis and scheduling must be done by the compiler.

Now in the following section we will study various superscalar architectures.

Superscalar Architectures: Pentium Processor

Pentium Features are as follows:

The Pentium is the third among Intel's 32-bit microprocessors, following the i386 & i486

1. It is a CISC architecture that achieves high performance by using many of the organizational features present in RISC processors.

2. This superscalar processor, supported by multiple pipelined operation units, can issue two instructions per clock cycle.
3. The Pentium is a 0.8 micron, BICMOS (Bipolar CMOS) technology, three layer metal 273-pin grid array package microprocessor.
4. It is a 32-bit system with a double 64-bit data bus inside and outside the chip. The gradual development of the Intel 32-bit microprocessors is illustrated in Table

Table: CPU Information

CPU	Year	Technology	Layers	Trans 10 ⁶	Frequency (MHz)
i386	1985	1.5-micron CMOS	2	0.275	16-20
i486	1989	1.0-micron CMOS	2	1.2	25-66
Pentium	1993	0.8-micron BICMOS	3	3.1	60-66

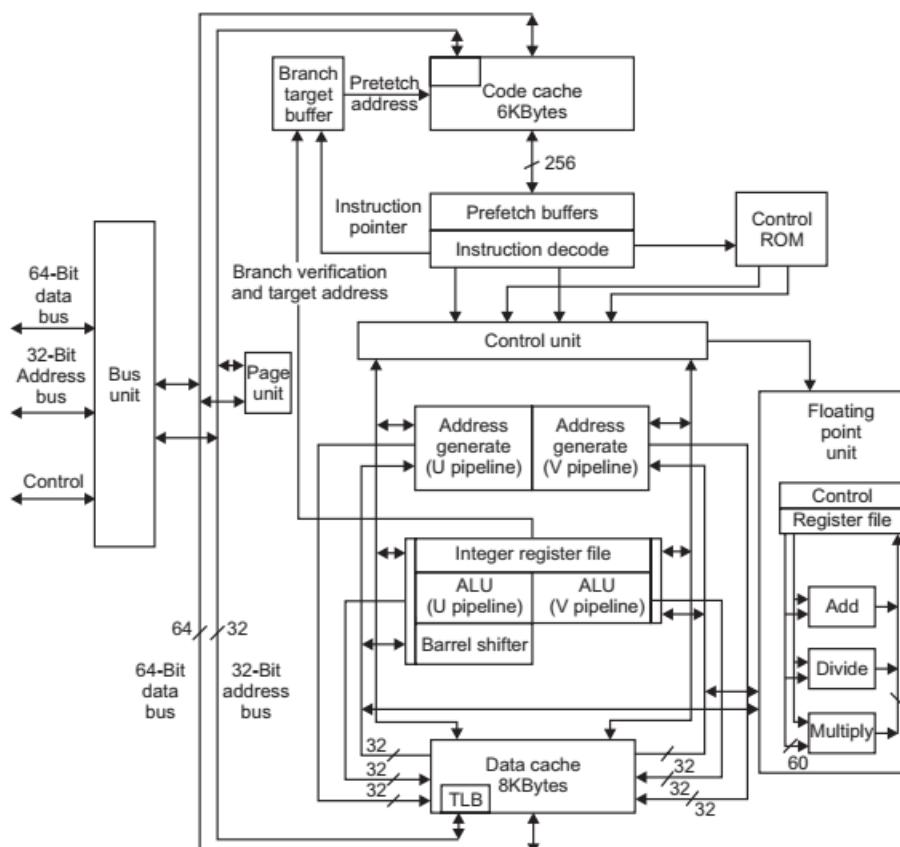


Fig. 4.10. Pentium processor block diagram (Courtesy of Intel Corp.)
Pipelined instruction processing in the pentium:

The Pentium has five pipelines:

- Integer pipeline (5 stages)
- Floating point pipeline (8 stages)
- Load pipelines (5 stages)
- Store pipeline (4 stages)
- Branch pipeline (5 stages)

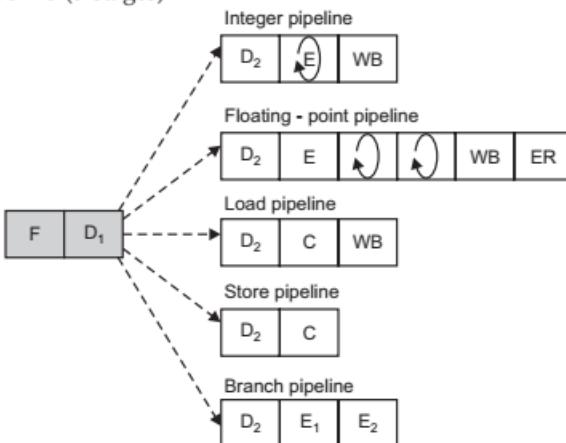


Fig. 4.11. Logical Layout of pentium's pipelines

C : Cache access

D : Decode

E : Execute

ER : Error reporting

F : Instruction fetch

WB : Write back

I : Iteration

For all pipelines two stages are common :

Instruction prefetch (F) and first decode (D1).

Five stage integer pipeline branching out into 2 paths u and v in last 3 stages.

1. PF-prefetch, CPU prefetches code from code cache.

(a) The internal and external data buses are 64 bit, and the internal and external address busses are 32-bit.

(b) There are 2 separate 8 kbytes caches :

One for code and

One for data

Each cache has separate address translation (TLB) with it. The availability of dual and dual TLB permits CPU to handle simultaneously instructions and data operand access facilitating efficient handling of pipeline.

(c) There are 256 lines between code cache and prefetch buffers, so prefetching 32 bytes of instructions.

(d) Supported by multiple pipelined operation units, can issue two instructions per clock cycle.

- (e) There are 2 parallel integer instruction pipelines:
- The U pipeline and V pipeline.
 - U pipeline has a parallel shifter in addition to regular ALU.
- (f) Also, separate FPU pipeline with individual floating point, add, multiply and divide units.
- (g) The integer and floating point register files are present.
- (h) There is also branch target buffer (BTB), supplying jump target prefetch addresses to code cache.
2. D1 (First Decode) CPU decodes instruction to generate control word. A single control word causes direct execution of an instruction. For complex instructions it require subsequent decoding.
 3. D2 (Second Decode) decodes control word generated in D1 for complex instructions and passes it to next stage E.
 4. E (Execute) Instruction is executed in ALU. If necessary, parallel shifter, data cache are used at this stage.
 5. WB (Write Back) CPU stores results and updates the flags.
- 8-stage floating point pipeline.
1. PF – Prefetch
 2. D1 – Decode first
 3. D2 – Second decode.
4. E – Operand fetch. Operands are fetched either from floating point register file or the cache.
 5. X1 – First execute – 1st step in floating point execution by FPU.
 6. X2 – Second execute – 2nd step in floating point execution by FPU.
 7. WF – Write float – Writes result into floating point register file.
 8. ER – Error reporting – The FPU reports internal special situations that might require additional processing to complete execution.

In contrast to the logical pipelines discussed above, physically the Pentium has **two master pipelines (called the U and V pipelines)** and a pipelined FP unit which operates in conjunction with the U pipe.

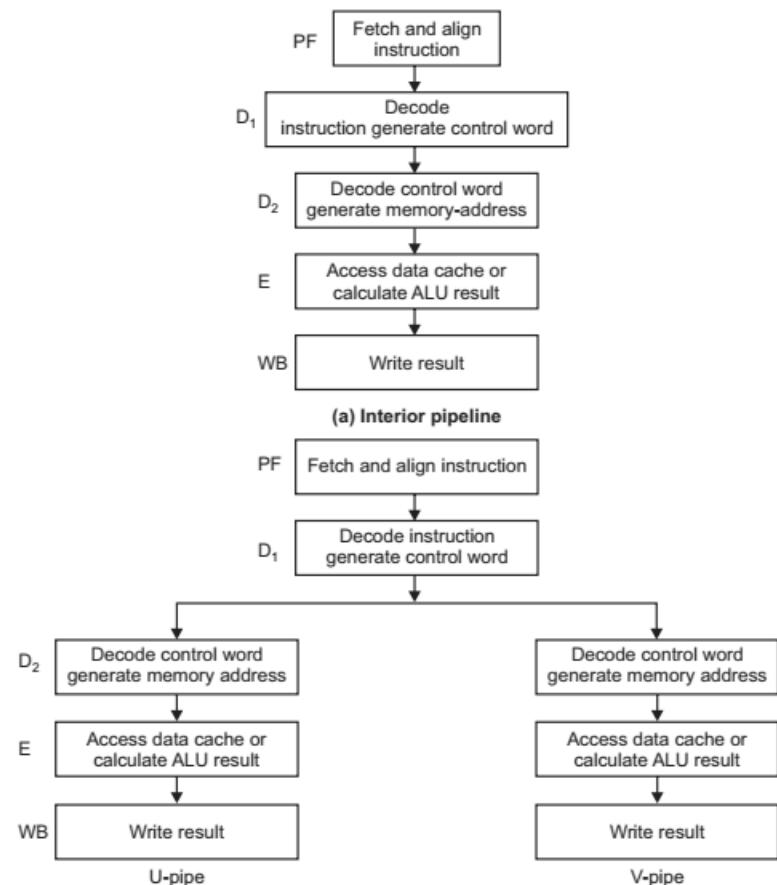


Fig. 4.12. Pentium Superscalar Pipeline

4.7 VLIW ARCHITECTURES

VLIW stands for Very Long Instruction Word. It is a technique that combines multiple standard instructions into one long instruction word. VLIW processors rely on the compiler to exploit ILP for determining the instructions which may be executed in parallel. The long instruction word known as "Multi OP" is composed of multiple arithmetic, logic and control operations. VLIW processor executes these operations within a multi OP concurrently and thereby achieves ILP. VLIW processor's instruction format is shown in Fig. 4.9 below :



Fig. 4.13. VLIW instruction (Multi OP).

Each operation in a VLIW instruction is equivalent to one instruction in a Super scalar or purely sequential processor. The number of operations in a VLIW instruction is equal to the number of execution units in the processor. Each operation specifies

the instructions that will be executed on the corresponding execution unit in the cycle during which the instruction is issued. The hardware need not have to examine the instruction stream for detecting ILP. The compiler is responsible for ensuring that all the operations specified in an instruction can be executed simultaneously. Due to this the VLIW processor has a very simple instruction issue logic as compared to that on a super scalar processor. In each cycle, the instruction logic fetches a VLIW instruction from the memory and issues it to the corresponding execution units.

Next, we will discuss the architecture of VLIW processor.

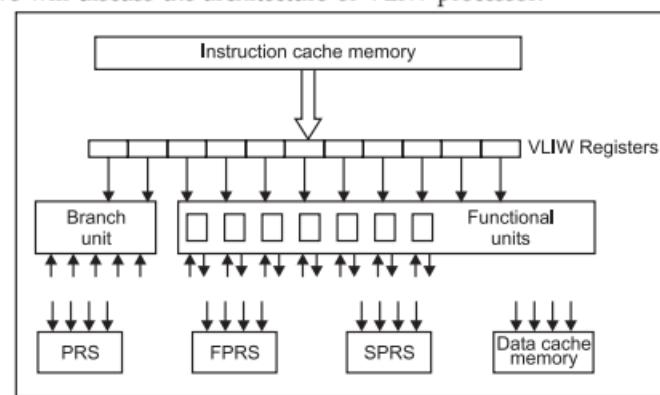


Fig. 4.14. VLIW Processor Architecture.

VLIW architecture are suitable for exploiting ILP in programs. These processors contain multiple functional units. A very long instruction word (VLIW) is fetched from the parallel execution. These capabilities are exploited by the compilers. The compilers generate code by grouping independent primitive instructions together which are executable in parallel. The processors have relatively simple control logic because they do not perform any dynamic scheduling nor reordering of operations as in case of super scalar processors.

Advantages of VLIW. The following are the advantages of VLIW processors :

1. The instruction issue logic is very simple.
2. Clock cycles required for implementing the instructions are shorter than super scalar processor.
3. The compiler has complete control for detecting ILP.
4. VLIW can fit more execution units on a given chip space because of simple instruction issue logic.

Disadvantages of VLIW. The following are the disadvantages of VLIW processors:

1. It is virtually impossible to maintain the **Compatibility between generations of VLIW processor family** because VLIW programs execute correctly when executed on a processor with **same number of execution units** and the same instruction latencies as the processor for which they were compiled.
2. If a compiler cannot find enough parallel operations, it has to insert explicit NOP operations into the corresponding operation slots. This results in **higher**

- memory requirement** as compared to the equivalent program for super scalar processors.
3. VLIWs cannot be programmed in assembly language because of the high task complexity. It is difficult for an assembly programmer to find out that how many execution units are free or busy, say, from 10–20 execution units.
 4. **Binary code compatibility** between two generations of VLIWs is very difficult to maintain as the instruction structure will invariably change.

Example Systems (VLIW) :

1. **The Trace 7/200 system.** It is capable of executing seven instructions cycle. It has a word length of about 256 bits.
2. **ELI 512.** It has a word length of 512 bits.

Applications. VLIW processors are useful for special purpose digital signal processing (DSP) application that actually demands high performance and low cost. However, they are less successful as General-purpose computers where software compatibility among the generations of processor is mandatory.

4.8 COMPARISON OF SUPER SCALAR AND VLIW IN A TABULAR FORM.

A Comparison of super scalar and VLIW is given in table below :

Super-scalar	VLIW
1. The code size is small	1. The code size is larger.
2. Complex hardware for decoding and issuing instructions	2. Simple hardware for decoding and issuing instructions.
3. They are compatible across generations.	3. They are not compatible across generations.
4. No change in hardware is required.	4. Requires more registers but simplified hardware for register ports.
5. They are scheduled dynamically by the processor.	5. They are scheduled statically by the compiler.

4.9 VECTOR AND SYMBOLIC PROCESSORS

Vector Processor : A Vector instruction involves a large array of operands. A **vector processor** is a coprocessor specially designed to perform vector computations. They are used in a multipipelined supercomputer. We have two types of vector operations–

- (a) Register-based vector instructions.
- (b) Memory-based vector instructions.

Register–vector instructions appear in most register-to-register vector processors like Cray supercomputers.

Let

V_i — Vector register

N — length of V_i

S_i — Scalar register

$M[1:n]$ — memory array

o — Vector operator (small circle)

Then, Vector operations can be :

- $V_1 \circ V_2 \rightarrow V_3$ (binary vector)
- $S_1 \circ V_1 \rightarrow V_2$ (Scaling)
- $V_1 \circ V_2 \rightarrow S_1$ (binary reduction)
- $M(1:n) \rightarrow V_1$ [Vector load]
- $V_1 \rightarrow M(1:n)$ (vector store)
- $\circ V_1 \rightarrow V_2$ (unary vector)
- $\circ V_1 \rightarrow S_1$ (unary reduction)

Please note here that the vector length should be equal in all operands used in a **Vector instruction**. Also note that the reduction is an operation of one or two vectors yielding a scalar result.

On the other hand, **memory-based vector operations** are performed in memory-to-memory vector processors like in cyber 205.

For example :

$$\begin{aligned} M_1(1:n) \circ M_2(1:n) &\rightarrow M(1:n) \\ S_1 \circ M_1(1:n) &\rightarrow M_2(1:n) \\ \circ M_1(1:n) &\rightarrow M_2(1:n) \\ M_1(1:n) \circ M_2(1:n) &\rightarrow M(p) \end{aligned}$$

Here, $M_1(1:n)$ and $M_2(1:n)$ are two vectors of length n .
 $M(p)$ – a scalar stored in memory location p .

Please note that the vector length is not restricted by the register length. The vector pipes can be attached to any scalar processor : Super scalar, super pipelined or both. An optimizing compiler is required to convert the sequential code to vectorized form for its vector pipelining.

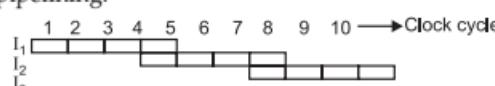


Fig. 4.15. Timing diagram of a vector pipeline.

Let us now study about **symbolic processors**.

Symbolic Processing has been used in many areas like theorem proving, pattern recognition, expert systems and so on. Herein, we have a modern and non-algorithmic approach of problem solving. Symbolic processors include prolog processors Lisp processors or symbolic manipulators. It depends on various features like how knowledge is represented? What common operations are to be performed? What are the properties of algorithms? And so on. In symbolic processing, we deal with logic programs, lists, objects, scripts, net, frames and neural networks. No floating-point operations are done here.

For example : Symbolics 3600 Lisp processor

4.10 CASE STUDY ON PENTIUM PROCESSOR (CISC)

Pentium is a CISC processor with a 2 instruction super scalar pipeline organization. Pentium uses a 5 stage pipeline. It has these stages :

First Stage : Prefetch Stage

Pentium instructions are variable length and are stored in a prefetch buffer. There is a 256-bit path from instruction cache to the prefetch buffer.

Second Stage: Decode – 1 Stage

The processor decodes the instruction and finds the opcode, checks which instructions can be paired for simultaneous execution.

Third stage : Decode-2 Stage

The addresses for memory reference are found in this stage.

Fourth Stage : Execute stage

In this stage, data cache fetch or ALU or floating point unit operation may be done.

Please note that two operations can be carried out.

Fifth Stage: Write-back stage

In this stage, registers and flags are updated based on the results of execution.

Pentium has 2 ALUs called U and V. Two instructions can be executed simultaneously. But there are some restrictions to ensure that no potential conflicts exist. **Two successive instructions, I_1 and I_2 can be dispatched in parallel to U and V units provided the following conditions are satisfied :**

1. Both I_1 and I_2 are simple instructions. An instruction is called simple if it can be carried out in one clock cycle.
2. I_1 and I_2 are not flow dependent or output dependent. That is, the destination register of I_1 is not the source or destination of I_2 and vice versa.
3. Only I_1 may contain an instruction prefix. An instruction prefix is 0-4 bytes long and has information like address size, operand size etc.
4. Neither I_1 nor I_2 contains both a displacement and an immediate operand.

Pentium employs a branch target buffer (BTB). A BTB caches information about recently encountered branch instructions. When an instruction in the instruction stream is fetched, the BTB is checked. If the instruction address is already there, it is a taken branch instruction. The history bits are checked to see if a jump is predicted. If yes, the branch target address is used to fetch the next instruction. If an entry for a branch instruction is not in BTB then it is updated.

4.11 CASE STUDY ON SPARC (RISC)

SPARC (Scalable Processor Architecture) is a RISC microprocessor architecture designed in 1985 by Sun Microsystems. SPARC is scalable across a wide range of semiconductor technologies, chip implementations and system configurations. SPARC is flexible. It is a flexible integration of cache, memory management and floating point units. So, processors are produced at a price that is suitable for systems ranging from leap-tops to super computers.

Solaris runs on SPARC-based systems.

SPARC architecture includes register window. SPARC has 32 GPRS, 8 global registers and 24 registers are in a register window. A window consists of three groups of 8 registers- the out, local and in registers. They are as follows :

Register Group	Register Address	Use
1. Global	$r[0] - r[7]$	Contains values that vary little between execution
2. Out	$r[8] - r[15]$	Contain outgoing parameters.
3. Local	$r[16] - r[23]$	Contain scratch registers.
4. In	$r[24] - r[31]$	Contain incoming parameters

SPARC can have 2 to 32 windows. Most implementations have 7 to 8 windows. At any given time, **only one window is visible that is determined by the current window pointer (CWP)**. This is a five-bit value that can be decremented or incremented by **SAVE** and **RESTORE** instructions respectively. These instructions are generally executed on the **procedure call and return** respectively. The **register window overlap partially**. So the **out register** become the **in registers** of the called procedure. Thus, the memory traffic is reduced when going up and down the procedure call. Since this is a frequent operation, so performance is improved.

The overlapping is shown below in Fig.4.12. :

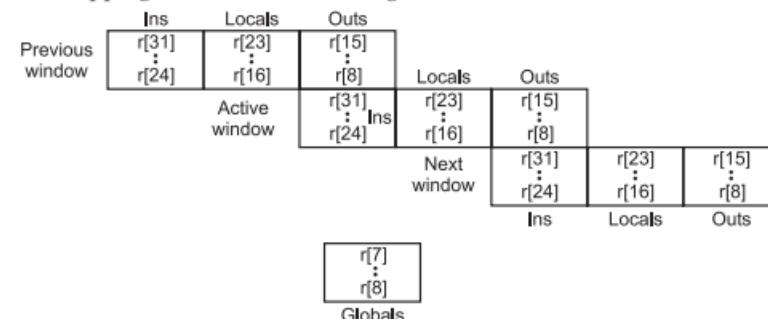


Fig. 416. Three overlapping register windows and global registers.

When a function is called, the **in register** may contain arguments that can be used. The **local registers** be used for anything while the function executes. The **out registers** are used to pass arguments to functions that it calls.

Register windows are also used to save the processor contexts when traps or interrupts occur. The SPARC OS always ensure that there is a register window not being used below the current one. If a trap occurs then the current workspace pointer (CWP) is decremented and the new window saves the processor context.

SUMMARY

The instruction – level parallelism (ILP) is the maximum number of instructions that can be simultaneously executed in the pipeline. For the base machine, all of these parameters have a value of 1. All other machine types are designed relative to the base machine. The ILP is needed to fully utilize a given pipeline machine. We summarize now in a tabular :

Type of Machine	Scalar base machine	Super scalar machine	Super pipelined machine	Super pipelined super scalar machine
Machine pipeline cycle	1(base cycle)	1	$1/n$	$1/n$
Instruction Issue rate	1	m	1	m
Instruction Issue latency	1	1	$1/n$	$1/n$
Simple Operation latency	1	1	1	1
ILP to fully utilize the pipeline	1	m	n	mn

MULTIPLE CHOICE QUESTIONS (MCQS) WITH ANSWERS

1. CISC stands for :
 - (a) Compound Instruction Symbolic Computer.
 - (b) Complex Instruction Set Computer.
 - (c) Common Idea Set Computer.
 - (d) None of the above.
2. CISC microprocessors have :
 - (a) Complex machine instruction. (b) Best addressing capability.
 - (c) Both (a) and (b). (d) None of the above.
3. A system in which one instruction is issued per cycle :
 - (a) Base Scalar processor. (b) Super scalar.
 - (c) Vector. (d) None of the above.
4. For an ideal pipeline, effective CPI is :
 - (a) 1 (b) 2
 - (c) 3 (d) 4
5. RISC stand for :
 - (a) Remote Instruction Set Computer.
 - (b) Reduced Instruction Set Computer.
 - (c) Ram Instruction Set Computer.
 - (d) None of the above.
6. 8086/186/286/386 are examples of :
 - (a) Vectors (b) RISC
 - (c) CISC (d) None of the above
7. RISC has :
 - (a) Unified cache (b) I and D cache
 - (c) L_1 cache (d) None of the above.

8. CISC has :

 - (a) 8-24 GPRs
 - (b) 32-192 GPRs
 - (c) 10-12 GPRs
 - (d) 50-100 GPRs

9. For Super scalar processor :

 - (a) $1 < m < 10$
 - (b) $5 < m < 15$
 - (c) $2 < m < 5$
 - (d) $10 < m < 100$

10. Pentium employs a :

 - (a) BTB
 - (b) PSP
 - (c) ASP
 - (d) DSP

11. A pure CISC processor is :

 - (a) 186
 - (b) 286
 - (c) 386
 - (d) 486

12. More RAM is required in :

 - (a) CISC
 - (b) RISC
 - (c) Vectors
 - (d) None of the above.

ANSWERS

- | | | | | |
|---------|---------|--------|--------|---------|
| 1. (b) | 2. (c) | 3. (a) | 4. (a) | 5. (b) |
| 6. (c) | 7. (b) | 8. (a) | 9. (c) | 10. (a) |
| 11. (c) | 12. (b) | | | |

CONCEPTUAL SHORT QUESTION WITH ANSWER

1. Define the following terms :
 - (a) Processor design space.
 - (b) Latency.
 - (c) Instruction issue rate.
 - (d) Simple latency.
 - (e) Resource conflicts.
 - (f) Processor versus coprocessor.
 - (g) GPRs.
 - (h) Addressing modes.
 - (i) Unified versus split caches.
 - (j) Hardwired versus micro coded control.

Ans. The definitions are given below :

- (a) **Processor design space** : It is defined as a coordinated space with the x and y axis. **X-axis represents clock-rate and y-axis represents CPI**. Each coordinate *i.e.*, point in the space (x, y) represents a design choice of a processor whose performance is determined by the values of the coordinates.
 - (b) **Latency** : It is defined as the time required between issuing two consecutive instructions.
 - (c) **Instruction issue rate** : It is defined as the number of instructions issued per cycle.

- (d) **Simple Latency** : The number of cycles required for the execution of a simple instruction such as add, move etc is known as simple latency.
 - (e) **Resource conflicts** : When two or more instructions attempt to use the same functional unit at the same time then it is known as resource conflict.
 - (f) **Processor versus coprocessor** : A coprocessor is one that is usually attached to a processor. Coprocessor performs special functions at a fast speed.
For example : Math coprocessor, graphical coprocessors etc.
 - (g) **General Purpose Registers (GPRs)** : Those registers that are for general use of the programmer and are not designated for special usage like the special-purpose registers-base registers or index registers.
 - (h) **Addressing modes** : It specifies how the effective address of an operand is generated so that its actual value can be fetched from the correct memory locations.
 - (i) **Unified versus split caches** : In a unified cache, both data and instructions are kept in the *same cache*. In a split caches, both data and instructions are held in the *separate caches*.
 - (j) **Hardwired versus micro coded control** : In **hardwired control**, the control signals for each instruction are generated by proper circuitry such as delay elements.
In **micro coded control**, each instruction is implemented by a set of **microinstructions**, which are stored in the control memory. The decoding of microinstructions generates appropriate signals to control the execution of an instruction.
2. Answer the following questions on designing scalar RISC or super scalar RISC processors :
- (a) Why do most RISC integer units use 32 bit GPR? Explain the concept of register windows implemented in the SPARC architecture?
 - (b) What are the design trade-offs between a large register file and a large D-cache?
 - (c) Explain the relationship between the integer unit and the floating point unit in most RISC processors with scalar or super scalar organization?

[GGSIPU, M. Tech-6th sem., May 2003]

- Ans.**
- (a) A windowing system divides the register file on a machine into groups which are assigned to different processors. There is usually overlap among the register sets to provide a fast communications mechanism among co-operating procedures for parameter passing. The use of a large number of GPRs allows less frequent memory accesses and thus speeds up program execution. **Please note here that access to memory is slower than GPRs.**
 - (b) A large register file and a large data cache both serve the purpose of reducing memory traffic. From implementation point of view, the same chip area can be used for either a large register file or a large data cache. From the programming point of view, registers can be manipulated by program code

but cache is transparent to the user. In fact, data cache is transparent to the user. In fact, data cache is primarily involved in load/store operations. The addressing of a cache involves address translation and is more complicated than that of a register file. **Reservation stations and recorder buffers** are used in super scalar machines to facilitate instruction look ahead and internal data forwarding which are needed to schedule multiple instructions through multiple pipelines simultaneously.

- (c) In most RISC processors, the integer unit executes load, store, integer, bit and control transfer functions. It also fetches instructions for the floating-point unit in some systems. The floating-point unit performs various arithmetic operations on floating-point numbers. The two units can operate concurrently.
- 3. Based on your understanding of advanced processor, answer the following questions on RISC, CISC, super scalar and VLIW architecture :
 - (a) Compare the instruction set architecture in RISC and CISC processors in terms of instructions format, addressing nodes and CPI.
 - (b) Give advantages & disadvantages of using common/separate caches.
 - (c) Distinguish between scalar RISC and super scalar RISC in terms of instruction issue, pipeline architecture and processor performance.
 - (d) Explain the difference between super scalar and VLIW architecture in terms of hardware and software requirement.

[GGSIPU-M. Tech. 6th sem., May 2003]

Ans. (a) We compare CISC and RISC in tabular form below :

Feature	CISC	RISC
1. Instruction format	16–64 bits per Instruction	fixed (32-bit) format.
2. Addressing modes	12–24	Limited to 3–5
3. CPI	2–15, on the average 5	<1.5, very close to 1.

(b) **Advantages of Separate caches :**

- 1. It doubles the bandwidth because two complementary requests are serviced at the same time.
- 2. **It simplifies the logic design** as the arbitrations between instruction and data accesses to cache is simplified or eliminated.
- 3. The **access time is reduced** because both data and instructions can be placed close to the functional units which will access them.

For example : Instruction cache can be placed close to the instruction fetch and decode units.

Disadvantages of separate caches :

- 1. It complicates the problem of consistency because data and instruction may coexist in the same cache block. This is true if self-modifying code is allowed or when data and instructions are intermixed and stored in the same cache block. To avoid this we require a compiler support to ensure

- that instruction and data are stored in different cache blocks.
2. It may lead to inefficient use of cache memory because the working set size of a program varies with time and the fraction devoted to data and instruction also varies. Therefore, *the sum of data cache size and instruction cache size is usually larger than the size of a unified cache*. So, the utilization of the instruction cache and/or data cache is likely to be lower.

Note : For separate caches three things are a must-

1. Dedicated data paths are required for both the data and instruction caches.
2. Separate MMUs and TLBs are also needed to shorten the time of address translation.
3. A higher memory bandwidth should be used for separate caches to support the increased demand.

In practice, there is a trade off between the degree of support provided and the resulting hardware complexity.

(c) **Instruction issue :** The scalar RISC processor issues one per cycle whereas the super scalar RISC can usually issue more than one per cycle.

Pipeline architecture : In an m-issue super scalar processor, up to m pipelines may be active in any base cycle. A scalar processor is equivalent to a super scalar processor with m = 1

Processor Performance : An m-issue super scalar can have a performance m times that of a scalar processor, provided both are driven by the same clock rate, no dependence relation or resource conflicts exist among instructions.

(d)

Super-scalar	VLIW
<ol style="list-style-type: none"> 1. Super scalar requires more sophisticated <i>hardware support</i> such as large recorder registers and reservation tables for efficient use of resources of the system. 2. Support for software is desired to reserve data dependencies and to improve the efficiency. 3. The code size is small. 4. They are compatible across generations. 5. They are scheduled dynamically by the processor. 	<ol style="list-style-type: none"> 1. Both the hardware and software support at run-time is usually simplified. For example: the decoding logic can be simple. 2. The VLIW instructions are compacted by the compiler. It explicitly packs together instructions which can be executed in concurrency based on heuristics or run-time statistics. 3. The code size is larger. 4. They are not compatible across generations. 5. They are scheduled statically by the compiler.

4. What causes a processor pipeline to be under-pipelined ?

Ans. There are 2 reasons for pipelines to get underutilized :

- (a) The instruction latency is longer than one base cycle.
- (b) The combined cycle time is greater than the base cycle.

5. What are the factors that limit the degree of super scalar design?

- Ans.** The dependence among instructions or resource conflicts among instructions can prevent simultaneous execution of instructions.
- 6. What are the differences between scalar and vector instructions ?**
- Ans.** Scalar instructions operate on a number or a pair of numbers at a time. Vector instructions perform identical operations on vectors of length usually much larger than scalars.
- 7. What will be the speed up gain of the vector pipeline over the scalar pipeline?**
- Ans.** Say, the pipeline has k -stages and the length of vector is N .
- ∴ First output is generated in the k^{th} cycle.
 - Later on, an additional output is generated in each cycle.
 - ∴ Last result comes out of the pipeline in $(N + K - 1)$ cycle.
- Using a base scalar machine, it takes NK cycles.
- ∴ Speed-up = $NK/(N + K - 1)$.
- 8. Say, parallel issue is added to vector pipeline execution. Now a parallel issue is also added to a super scalar pipeline of the same degree. What is the speed improvement ?**
- Ans.** If an m -issue vector processing is used then each vector is of length = N/m .
- $$\therefore \text{Execution time} = \left(\frac{N}{m} + k - 1 \right) \text{cycles}$$
- If only parallel issue is used, the execution time = $(N/m)k$.
- $$\therefore \text{Speed improvement} = \frac{N/m + k - 1}{(N/m)k}$$
- $$= \frac{Nk}{N + m(k - 1)}$$
- 9. What are processors and coprocessors ?**
- Ans.** The central (heart) of a computer is known as a CPU or a processor. This CPU is actually a scalar processor, which may consist of multiple functional units like integer arithmetic, ALU etc.
- To perform certain other complex numerical calculations, coprocessors are also used in conjunction with processors. **This pairing is must as coprocessors cannot be used alone.** For example, digital signal processor (DSP) or a LISP processor executes AI programs, floating-point accelerator, vector processors, Math co-processor etc. Please note that coprocessors cannot handle I/O operations. So, coprocessors are also known as attached processors (APs) or slave processors. **Also note that the processor and coprocessor operate with a host-back-end relationship.** An AP may be more powerful than its host.
- For example :*
1. Intel 8087 coprocessor is used alongwith the Intel 8086/8088 processor.
 2. Intel 80287 coprocessor is used with Intel 80286 processor.
- 10. Define the following terms :**
- (a) Instruction pipeline cycle.

(b) Issue latency.

(c) Issue rate.

(d) Simple operation latency.

(e) Resource conflicts.

Ans. (a) The clock period of the instruction pipeline.

(b) The time required, in cycles, between the issuing of two adjacent instructions.

(c) The number of instructions issued per cycle.

(d) Simple instruction like adds, loads, stores, branches, moves etc have lesser delays in execution than the complex instructions like divides, cache misses etc. These latencies are measured in number of cycles.

(e) A situation where two or more instructions demand use of the same functional unit at the same time.

11. Consider a shared bus parallel computer built using 32 bit RISC processors running at 150 MHz with CPI-1. Assume that 15% of the instructions are loads and 10% are stores. Assume 0.95 hit ratio to cache for read and write through caches. The bandwidth of the bus is 2GB/sec.

(a) How many processors can the bus support without getting saturated?

(b) If caches are not there, how many processors can the bus support assuming the main memory is as fast as the cache.

[UPTU, B. Tech (CSE) 8th Sem., 2007-08]

Sol. (i) RISC processor processes 32 bits in $\frac{1}{150 \times 10^6}$ secs [$150 \text{ MHz} = 150 \times 10^6$

Hz]

In 1 sec, RISC processor requires $32 \times 150 \times 10^6$ bits

$$\therefore \text{No. of processor} = \frac{2 \times 10^9}{32 \times 150 \times 10^6} \quad [1 \text{ GB} = 1 \times 10^9 \text{ bits}] \\ = 4.2 \approx 4.$$

(ii) In order to get the bus support, the processors without being saturated, the cache which has additional overhead makes the processor wait for additional memory cycles. So the number of processors goes on increasing.

EXERCISE QUESTIONS

1. (a) Compare CISC and RISC architectures.

(b) With the help of pipeline diagrams, explain super scalar and VLIW processors.

[UPTU, B. Tech (CSE) 8th Sem., 2005-06 & GGSIPU-M.Tech. 1st sem., Dec. 2004]

2. Compare RISC architecture with CISC architecture. [KUD-B.E(CSE) 8th sem. 1996]

3. Discuss the salient features and important applications of RISC and CISC architectures. [KUD. BE.(CSE) 8th sem. 1997]

4. Explain major attributes of RISC machines. [KUD,BE(CSE) 8th sem, 1996]
5. What are the features of scalable processor architecture? Explain register window and its parameter passing between procedures of SPARC architecture.
[Pune Univ.,B.E. (CSE) 2nd sem., May 2004 Dec. 2005]
6. Explain SPARC architecture & VLIW processors.
[Pune Univ. B.E. (CSE) 2nd sem., May 2004, Dec. 2005]
7. Draw and explain SPARC architecture.
[Pune Univ., B.E. (CSE) 2nd sem., Dec. 2004]
8. Discuss the CISC and RISC architectures with the help of their functional block diagrams. Also indicate their advantages and drawbacks.
[UPTU, B. Tech (CSE) 8th Sem., 2003-04]
9. What do you understand by the term branch prediction ? Why do many super-scalar processors include hardware to perform branch prediction?
[GGSIPU, B. Tech (CSE) 7th Sem., Dec. 2011]
10. Write short notes on RISC.
[GGSIPU, B. Tech (CSE) 7th Sem., Dec. 2011]



CHAPTER

5 MEMORY TECHNOLOGY

5.0 INTRODUCTION

The memory of a microcomputer is the space where program and data are stored before execution and computation. The storage that is controlled by the CPU of the computer is known as an **online storage**. For instance, a magnetic disc directly connected with CPU is an online disc file. A storage device not under the control of CPU is the **off-line storage**.

5.1 HIERARCHICAL MEMORY TECHNOLOGY

Practically there is a very large speed gap between CPU and memory. **Memory is about 1000 times slower than CPU**. To close up this speed gap, a hierarchical memory system is used. The memory hierarchy system consists of all storage devices employed in a system from slow but high capacity auxillary memory to a relatively faster main memory, to as even smaller and faster cache memory accessible to the high speed logic. Each memory is involved with different levels in the memory hierarchy system. The reason for having two or three levels of memory hierarchy is economics. The memory hierarchy is shown in Fig. 5.1 below.

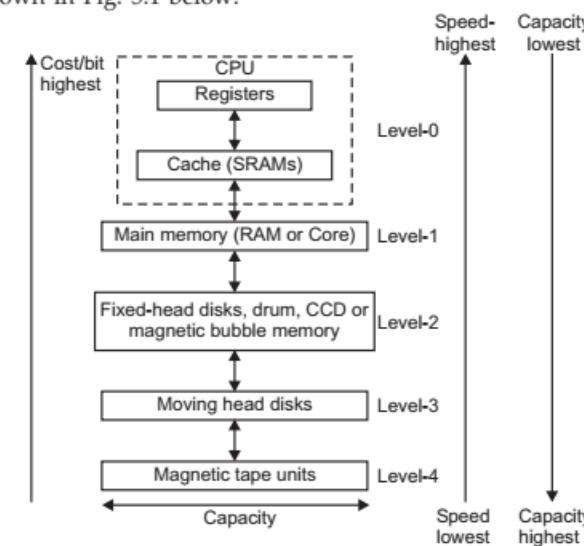


Fig. 5.1. Memory hierarchy.

Please note that to make an efficient computer system, it is not possible to rely only on a single memory component but on a multilevel memory hierarchy. As shown in Fig. 5.1. Register file is the innermost level of memory. It is directly addressable by ALU. Next layer is cache memory. Cache memory acts as a buffer between CPU and the main memory (MM). Main memory can be accessed in blocks by using multiway interleaving across parallel memory modules. At the outer levels of hierarchy, disks and tape units are used to establish the virtual memory space. This hierarchy of fig. 5.1 is used to broaden the memory bandwidth so that it will match with the bandwidth of the CPU.

Five parameters characterize this memory technology :

- (a) Access time (t_i) : It is defined as the round-trip time from the CPU to the i^{th} level memory. It can range from a few nanoseconds (ns) to microseconds (μs).
- (b) Memory size (s_i) : It is defined as the number of bytes or words in level i .
- (c) Cost : The cost of the i^{th} level memory is estimated by the formula :

$$C_i * S_i$$

Where C_i — cost/byte

S_i — No. of bytes/words in level i

- (d) Bandwidth (b_i) : It is defined as the rate at which information is transferred between adjacent levels.
- (e) Unit of transfer (x_i) : It is defined as the grain size for data transfers between the levels i and $i + 1$.

Please note that the memory devices at a lower level are faster to access, smaller in size, more expensive per byte, having a higher bandwidth and using a smaller unit of transfer as compared with those at a higher level. Also note here that :

$t_{i-1} < t_i$
$S_{i-1} < S_i$
$C_{i-1} > C_i$
$b_{i-1} > b_i$
$x_{i-1} < x_i$

Where $i \leftarrow 0$ to 4

For e.g., $i = 0$ means CPU register level.

$i = 1$ means main memory.

:

$i = 4$ means tape units.

5.2 INCLUSION

As shown in Fig. 5.2 below, M_1 corresponds to cache, M_2 corresponds to main memory M_3 corresponds to disk storage. The inclusion property states that :

$$M_1 \subset M_2 \subset M_3 \subset \dots \subset M_n$$

It means that all information is originally stored in the outermost level, M_n . During the processing, subsets of M_n are copied into M_{n-1} . Similarly, subsets of M_{n-1} are copied into M_{n-2} and so on.

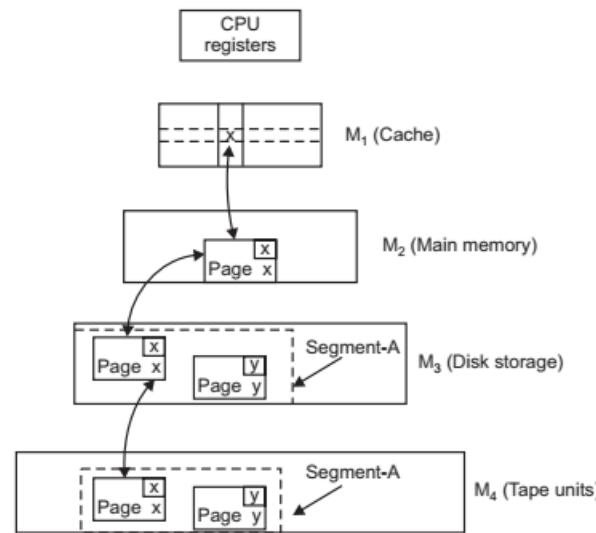


Fig. 5.2. Inclusion property.

If an information is found in M_i then copies of the same word (8, 16 and 32 bits) will also be found in upper levels M_{i+1} , M_{i+2} , ..., M_n (for $i \leftarrow 1$ to 4 above).

Please note that a word stored in M_{i+1} may not be found in M_i . So, if a word is missing at level M_i then it implies that it is also missing from all the lower levels i.e., M_{i+1} , M_{i+2} , ..., M_1 . As shown in Fig. 5.2., the highest level is the backup storage where everything can be found.

Note that the information transfer between the CPU and cache is in terms of words i.e., 4 or 8 bytes each depending on the **word length of a machine**.

5.3 COHERENCE

The coherence property means that the copies of the same information item at the successive memory levels must be consistent. This simply means that if a word is changed in the cache (M_1 -level) then copies of that word must be updated immediately at the higher levels. But the memory hierarchy is to be maintained.

Tip : Store frequently used information at the lower levels (say, M_1) so that access time of memory is minimal.

How this is done ?

In general there are two techniques for maintaining the coherence in this memory hierarchy :

- (a) **Write-through (WT)** : It means an immediate update in M_{i+1} if a word is modified in M_i ($i \leftarrow 1, 2, \dots, n-1$).
- (b) **Write-Back (WB)** : It means delaying the update in M_{i+1} until the word being modified in M_i is replaced or removed from M_i .

5.4 LOCALITY OF REFERENCE

According to Hennessy and Patterson's (1990) **90–10 rule**—"A typical program may spend 90% of its execution time on only 10% of the code." For instance, a program may be spending most of its time on nested innermost loop. This concept is used here. Memory addresses are generated by the CPU. This address is used to access data or instruction. These accesses are actually clustered in certain regions in time, space and ordering. So, we call this as **locality of reference**. There are three types of localities of reference :

1. Temporal locality.
2. Spatial locality.
3. Sequential locality.

Temporal locality tends to cluster the access in the recently used areas.

For example : Once a loop is entered or a subroutine is called, some part of program will be referenced again and again.

Spatial locality tends to cluster the access of items whose addresses are near to one another.

For example : Operations on arrays; $a[0]$ and $a[1]$ are nearby—shows spatial locality.

Sequential locality means sequential execution of instructions in sequential order or accessing of an array shows sequential locality of reference.

For example : Accessing a large data array is sequential.

5.5 MEMORY CAPACITY PLANNING

We need to define three terminologies here. They are given below :

1. **Hit Ratio (h_i)** : Referring back to our Fig. 5.1 of memory hierarchy, we say that when an information item is found at M_i , it is a **hit** else it is a **miss** so, we define the hit-ratio (h_i) at M_i as the probability that an information item will be found in M_i . So, miss-ratio at M_i is defined as $(1-h_i)$.

Please note here that the hit-ratios are independent random variables with their values between 0 and 1.

The access frequency to M_i is given by

$$f_i = (1-h_1)(1-h_2) \dots (1-h_{i-1}) h_i$$

Also $\sum_{i=1}^n f_i = 1$

Actually, $f_1 \gg f_2 \gg f_3 \gg \dots \gg f_n$.

This implies that the inner levels of the memory are accessed more often than the outer levels of the memory.

2. **Effective Access Time (T_{eff})** : T_{eff} of a memory hierarchy is defined as follows :

$$\begin{aligned} T_{\text{eff}} &= \sum_{i=1}^n f_i \cdot t_i \\ &= h_1 t_1 + (1 - h_1) h_2 t_2 + (1 - h_1)(1 - h_2) h_3 t_3 + \dots + \\ &\quad (1 - h_1)(1 - h_2) \dots (1 - h_{n-1}) t_n. \end{aligned}$$

- T_{eff} depends on the program's behavior and the memory design choices.
3. **Total cost of memory hierarchy (C_{total})** : It is given by the formula :

$$C_{\text{total}} = \sum_{i=1}^n C_i \cdot S_i$$

It means that the cost is distributed over n levels. Because $C_1 > C_2 > C_3 > \dots > C_n$ so we have to choose $S_1 < S_2 < S_3 < \dots < S_n$.

An **ideal design** of memory hierarchy should have following parameters :

- (a) $T_{\text{eff}} \approx t_1$ (of M_1).
- (b) $C_{\text{total}} \approx C_n$ (of M_n).

Practically, this is not possible due to the tradeoffs among n -levels.

Our problem now is to minimize

$$T_{\text{eff}} = \sum_{i=1}^n f_i \cdot t_i$$

and the constraints are :

$$\begin{cases} S_i > 0 \\ t_i > 0 \end{cases} \text{ where } i \leftarrow 1 \text{ to } n$$

So, $C_{\text{total}} = \sum_{i=1}^n C_i S_i < C_0$

We need to manage the tradeoffs among t_i , C_i , S_i and f_i (or h_i) at all levels ($i \leftarrow 1$ to n).

We are in a position to solve a problem now.

Example 1. Consider a three level memory hierarchy ($M_1 - M_3$) as shown below :

Memory level	Access time	Capacity	Cost/KB
M_1 (cache)	$t_1 = 25 \text{ ns}$	$S_1 = 512 \text{ KB}$	$C_1 = \$1.25$
M_2 (MM)	$t_2 = ?$	$S_2 = 32 \text{ MB}$	$C_2 = \$0.2$
M_3 (Disk)	$t_3 = 4 \text{ ms}$	$S_3 = ?$	$C_3 = \$0.0002$

It is desired that :

$$\begin{aligned} T_{\text{eff}} &= 10.04 \mu\text{s} \\ h_1 &= 0.98 \text{ (for } M_1) \\ h_2 &= 0.9 \text{ (for } M_2) \end{aligned}$$

But $C_{\text{total}} \leq \$15000$.

Find S_3 , t_2 (marked as ? in table) ?

Solution. We know that the cost of memory hierarchy is given by the formula :

$$\begin{aligned} C &= C_1 S_1 + C_2 S_2 + C_3 S_3 \leq 15000 \\ \text{or } 1.25 (512) + 0.2 (32) + 0.0002 (S_3) &= 15000 \\ \text{or } S_3 &= 39.8 \text{ Gbytes} \end{aligned}$$

Now

$$\begin{aligned} T_{\text{eff}} &= h_1 t_1 + (1 - h_1) h_2 t_2 + (1 - h_1) (1 - h_2) h_3 t_3 \leq 10.04 \\ \text{or } 10.04 \times 10^{-6} &= 0.98 \times 25 \times 10^{-9} + 0.02 \times 0.9 \times t_2 + 0.02 \times 0.1 \\ &\quad \times 1 \times 4 \times 10^{-3} \\ \text{or } t_2 &= 903 \text{ ns.} \end{aligned}$$

Now, if someone wants to double the main-memory (M_2) from 32 MB to 64 MB. In such a case, cache hit ratio is unaffected but the hit ratio of main memory (M_2) will increase. T_{eff} will also increase accordingly.

5.6 VIRTUAL MEMORY TECHNOLOGY

The main memory is also known as a physical memory. But this memory is limited in size. To solve this problem an auxiliary memory is also used. A memory hierarchy consisting of **main memory and auxiliary memory is known virtual memory**.

Auxiliary or back up memory can be like disk arrays. Only active programs or some portions of them are stored in physical memory at one time. All programs are loaded in and out of the physical memory dynamically under the control of O.S. Without virtual memory, it is impossible to implement multiprogramming or time sharing features. Each word in the physical memory is given a **physical address**. This forms a **physical address space**. **Virtual addresses are generated by CPU at compile time**. This is shown below in Fig. 5.3.

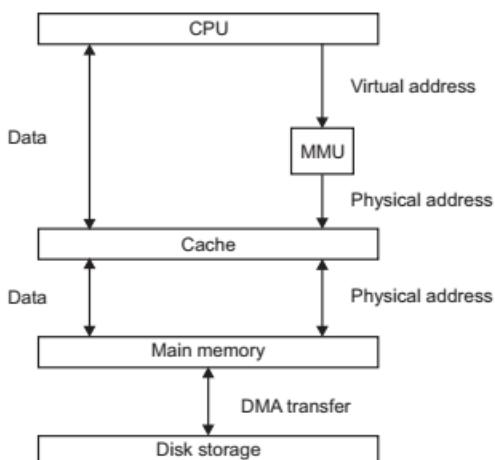


Fig. 5.3. Virtual memory organization.

As shown in Fig. 5.3 above, that the **CPU generates virtual addresses that are to be translated to physical addresses and that too at run time**. This is done by a special hardware called as **Memory Management unit or MMU**.

There are 2 ways of having virtual memories :

- (a) Private virtual memory.
- (b) Global virtual memory.

In **private virtual memory space model**, each processor has its own private virtual address space. These virtual addresses are mapped to the same physical memory that is shared by all the processors. This is shown in Fig. 5.4.

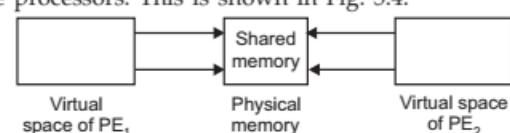


Fig. 5.4. Private VM.

In global virtual memory space model, a single global virtual space is formed. We don't have different spaces for PEs now. This is shown in Fig. 5.5.

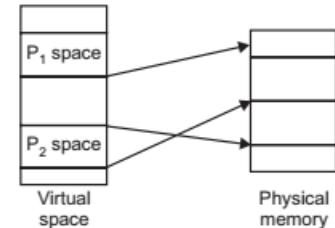


Fig. 5.5. Global VM.

Applications : VAX-11 uses first model.

IBM-801, HP spectrum, standford.

Dash use Second model.

In Second model, all addresses are unique.

Role of TLB and PT for address translations

As we know that the virtual address (by CPU) are translated to physical address at run time by MMU. That is, as shown in Fig. 5.6.

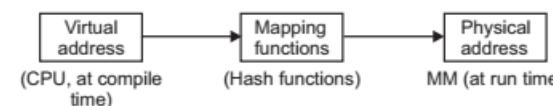


Fig. 5.6. Translation

Basically, we have translation maps that are in form of :

- (a) Translation lookaside buffer (TLB)
- (b) Page Tables (PTs)

On the basis of **locality of reference**, a particular **working set of pages** is referenced within a given context or time window. The use of TLB and PT is shown below in Fig. 5.7.

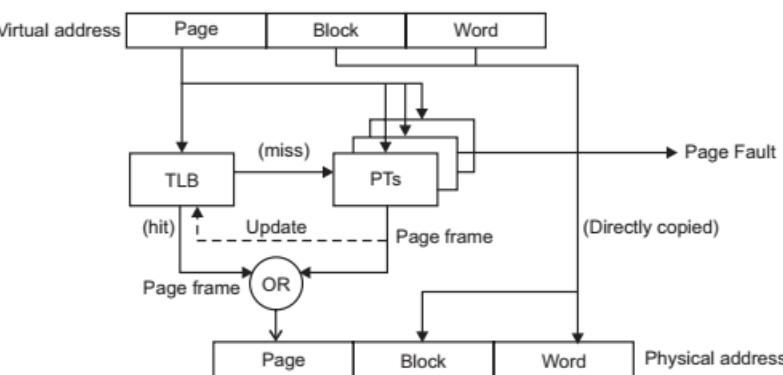


Fig. 5.7. Role of TLB and PT during address translations.

We observe that each virtual address is divided into three fields—virtual page number, cache **block** number and **word** address within the block. Our objective is to produce the physical address (at bottom of Fig. 5.7) having page frame number, block number and the word address.

How it works ? There are some steps.

Step 1 : The virtual page number (leftmost field) is used to search through the TLB for a match. This TLB can be an associative memory or content addressable memory (CAM) or part of cache memory. TLB is a high-speed look up table. It stores recently referenced page entries.

Note : Page entry = virtual page number + page frame number

The pages belonging to the same working set will be directly translated using the TLR entries.

Step 2 : If there is a match (or hit) in the TLB, the page frame number is retrieved from the matched page entry. However, the cache block (middle field) and the word address are copied directly.

Step 3 : If there is no match (or miss) in the TLB, a hashed pointer is used to identify one of the page tables where the desired page frame number can be retrieved.

5.7 PAGE REPLACEMENT ALGORITHMS

When the processor need to execute a particular page, that page is not available in main memory, this situation is said to be "page fault". When the page fault occurs, this page replacement will be needed. The word "page replacement" means select a victim page in the main memory, replace that page with the required page from the backing store (Disk). The students may have a doubt, how to select a victim page. The answer is simple "Page replacement algorithms". The page replacement algorithms select the victim pages.

Now a days the number of page replacement algorithms are available. Each operating system has its own page replacement policy. Now we consider some of the popular page replacement algorithms.

5.7.1 EIFO algorithm : (First in first out algorithm)

FIFO is the simplest page replacement algorithm, the idea behind this is “Replace a page that page is the oldest page of all the pages of main memory” or “Replace the page that has been in memory longest”. FIFO focuses on the length of time a page has been in memory rather than how much the page is being used. To illustrate the page replacement algorithm, we shall use the reference string

genuine, we shall use the reference string:

for a memory with 3 frames. This example incurs 16 page faults under FIFO, as shown in the table.

Table EIEQ Behaviour

This example incurs 16 pages faults. The symbol “*” indicates the new page in the memory. The symbol “✓” indicates page fault. The performance of the page replacement algorithm measured through the page fault rate.

$$\therefore \text{Page fault rate} = \frac{\text{Number of page faults}}{\text{Number of bits in the reference string}} \\ = 16/16 = 100\%$$

An algorithm having the least page fault rate, we can say that algorithm is best one.

If the required page is in the main memory, then page replacement is not required, in that situation we use the ‘x’ symbol. For that consider the next example.

The reference string is 0, 1, 2, 3, 2, 5, 6, 3, 4, 6, 3, 7, 3, 1, 5, 3, 6, 4, 3, 4, 2, 3, 4, 5, 1. For a memory with 4 frames. This example incurs 12 page faults. Consider the table below.

Table FIFO behaviour

Frame	1	2	3	2	5	6	3	4	6	3	7	3	1
0	1*	1	1	1	1	6*	6	6	6*	6	6	6	1*
1	-	2*	2	2*	2	2	2	4*	4	4	4	4	4
2	-	-	3*	3	3	3	3*	3	3	3*	7*	7	7
3	-	-	-	-	5*	5	5	5	5	5	5	3*	3
	✓	✓	✓		✓	✓	✗	✓	✓	✗	✗	✓	✓

Frame	5	3	6	3	4	2	4	3	4	5	1
0	1	1	1	1	1	2*	2	2	2	2	2
1	5*	5	5	5	5	5	5	3*	3	3	3
2	7	7	6*	6	6	6	6	6	6	5*	5
3	3	3*	3	3*	4*	4	4*	4	4	4	1*
	✓	✗	✓	✗	✓	✓	✗	✓	✗	✓	✓

Page fault rate = 16/24 = 75%

5.7.2 Optimal page replacement algorithm

The optimal page replacement algorithm has the lowest page fault rate of all algorithms. The criteria of this algorithm is **“Replace a page that will not be used for the longest period of time”**. To illustrate this algorithm, consider the reference string 1, 2, 3, 2, 5, 6, 3, 4, 6, 3, 4, 6, 3, 7, 3, 1, 5, 3, 6, 3, 4, 2, 4, 3, 4, 5, 1. Assume that the memory size is four frames. Consider the table to calculate the page fault rate.

Table Optimal Behaviour

Frame	1	2	3	2	5	6	3	4	6	3	7	3	1
0	1*	1	1	1	1	1	1	1	1	1	1	1	1*
1		2*	2	2*	2	6*	6	6	6*	6	6	6	6
2			3*	3	3	3*	3	3	3*	3	3*	3	3
3				5*	5	5	4*	4	4	7*	7	7	7
	✓	✓	✗	✓	✓	✗	✓	✗	✗	✓	✗	✗	✗

Frame	5	3	6	3	4	2	4	3	4	5	1
0	1	1	1	1	1	2*	2	2	2	2	1*
1	6	6	6*	6	4*	4	4*	4	4*	4	4
2	3	3*	3	3*	3	3	3	3*	3	3	3
3	5*	5	5	5	5	5	5	5	5	5*	5
	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✓

∴ Page fault = 11/24

The optimal page replacement algorithm is difficult to implement, because it requires future knowledge of reference string, so this algorithm is used mainly for comparison studies.

5.7.3 LRU (Least Recently Used Algorithm)

The criteria of this algorithm is “Replace a page that has not been used for the longest period of time”. This strategy is the “Page replacement algorithm looking backward in time, rather than forward”. To illustrate the algorithm consider the reference string.

0, 1, 2, 3, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5, 6, 7 with 3 main frames.

Frame	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0*	0	0	3*	3	3	2*	2	2	1	1*	1	4*	4	4	7*
1		1*	1	1	0*	0	0	3*	3	3	2	2*	2	5*	5	5
2			2*	2	2	1*	1	1	0*	0	0	3	3	3	6*	6
	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

∴ Page fault rate = 16/16 = 100%

5.7.4 Least Frequently Used Algorithm (LFU)

The least Frequently used algorithm “selects a page for replacement, if the page has not been used often in the past” (or) “Replace page that page has smallest count”. For this algorithm each page maintains a counter, which counter value shows the least count, replace that page. The reason for this selection is that an actively used page should have a large reference count, so don't replace the actively used page. The frequency counter is reset each time a page is loaded. To illustrate this algorithm consider below reference string.

0 1 2 3 0 1 2 3 0 1 2 3 4 5 6 7 with 3 memory frames.

Table LFU Algorithm

Frame	0	1	2	3	0	1	2	3	0	1	2	3	4	5	6	7
0	0*	0	0	0	0+	0	0	0	0+	01	0	3*	3	3	3	3
1		1*	1	1	1	1+	1	3*	3	1*	1	1	1	1	1	1
2			2	3*	3	3	2*	2	2	2	2+	2	4	5*	6*	7*
	✓	✓	✓	✓	✗	✗	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓

∴ Page faults = 12/16

5.7.5 Most Frequently used Algorithm (MFU)

The criteria of this algorithm is replace a page that has the maximum frequency count of all pages. The implementation of this algorithm is fairly expensive.

5.7.6 Belady's Anomaly

The general principle is if the number of frames is increased, the page fault rate will be decreased. For example consider the following reference string.

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

If we apply the FIFO algorithm with 3 frames, there are 15 page faults, if we increase to 4 frames, there are 12 page faults. But consider the reference string, given below :

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

The number of page faults for 4 frames (10) is greater than the number of faults for 3 frames (9). This result is most unexpected and is known as "**Belady's Anomaly**".

Table 4 Frames

Frame	1	2	3	4	1	2	5	1	2	3	4	5
0	1	1	1	1	1+	1	5*	5	5	5	4	4
1		2	2	2	2	2+	2	1*	1	1	1	5
2			3	3	3	3	3	3	2*	2	2	2
3				4	4	4	4	4	4	3	3	3
	✓	✓	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓

∴ Page fault for 4 frames = 10/12

With 3 Frames

Frame	1	2	3	4	1	2	5	1	2	3	4	5
0	1*	1	1	4	4	4	5*	5	5	5	5	5+
1		2*	2	2*	1	1	1	1	1	3	3	3
2			3*	3	3*	2	2	2	2	2	4	4
	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

∴ Page fault for 3 frames = 9/12

Belady's Anomaly was not always true, it is applicable only to FIFO algorithm

SUMMARY

In this chapter, we have seen how levelwise, a memory can be organized. There are three important memory related properties. Virtual memory is a technique that allows the execution of a process, even if the logical address space is greater than the physical available memory. Programs are loaded in the main memory (MM), if the MM is less than the logical memory using swapping concept.

MULTIPLE CHOICE QUESTIONS (MCQS) WITH ANSWERS

1. Magnetic disc is an example of :
 (a) Online storage (b) Offline storage
 (c) Offset storage (d) None of the above.
2. Memory is slower than CPU :
 (a) 100 times (b) 10 times
 (c) 1000 times (d) 10,000 times.
3. Caches are usually built out of :
 (a) sRAMS (b) dRAMS
 (c) PROM (d) E²PROM
4. Round-trip time from the CPU to the i^{th} level memory is known as :
 (a) Execution time (b) Slot time
 (c) Access time (d) None of the above.
5. Inclusion property states that :
 (a) $M_1 \subset M_2 \subset M_3 \subset \dots \subset M_n$ (b) $M_1 \subset M_2$ but $M_2 \not\subset M_3$
 (c) $M_1 \not\subset M_2 \not\subset M_3 \not\subset \dots \not\subset M_n$ (d) None of the above.
6. The information transfer between CPU and cache is in terms of :
 (a) bytes (b) bits
 (c) Words (d) None of the above.
7. A 90-10 rule was given by :
 (a) Henry and kafura (b) Boehm
 (c) Hennessy and Patterson (d) None of the above.
8. Loops show :
 (a) Temporal locality (b) Spatial locality
 (c) Sequential locality (d) None of the above.
9. Arrays show :
 (a) Spatial locality (b) Spatial and sequential locality
 (c) Only sequential locality (d) All of the above.
10. Effective access time (T_{eff}) of memory is given by :
 (a) $T_{\text{eff}} = \frac{1}{\sum_{i=1}^n f_i \cdot t_i}$ (b) $T_{\text{eff}} = \sum_{i=1}^n f_i \cdot t_i$
 (c) $T_{\text{eff}} = \sum_{i=1}^n f_i / t_i$ (d) None of the above.
11. The total cost of memory hierarchy is given by :
 (a) $C_{\text{total}} = \sum_{i=1}^n C_i S_i$ (b) $C_{\text{total}} = \sum_{i=1}^n C_i / S_i$
 (c) $C_{\text{total}} = \sum_{i=1}^n (C_i + S_i)$ (d) None of the above.

12. CPU generates :

 - (a) Virtual addresses
 - (b) Physical addresses
 - (c) Login addresses
 - (d) None of the above.

13. MMU stands for :

 - (a) Main Memory Unit
 - (b) Memory Management Unit
 - (c) Memory to Memory Unit
 - (d) None of the above.

14. TLB and PTs are :

 - (a) Cache memories
 - (b) Translation maps
 - (c) Auxiliary memory
 - (d) None of the above.

15. Page fault rate (PFR) is given by :

 - (a)
$$PFR = \frac{\text{No. of page faults}}{\text{No. of bits in reference string}}$$
 - (b) PFR = No. of page faults only
 - (c) PFR = No. of page faults * No. of bits in reference string.
 - (d) None of the above.

16. In which of the following policies, Belady's anomaly occurs :

 - (a) FIFO
 - (b) LRU
 - (c) LFU
 - (d) NRU

ANSWERS

- | | | | | |
|---------|---------|---------|---------|---------|
| 1. (a) | 2. (c) | 3. (a) | 4. (c) | 5. (a) |
| 6. (c) | 7. (c) | 8. (a) | 9. (b) | 10. (b) |
| 11. (a) | 12. (a) | 13. (b) | 14. (b) | 15. (a) |
| 16. (a) | | | | |

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

- ### 1. Distinguish between sRAM and dRAM ?

[UPTU, B.Tech. (CSE) : 2004-05]

Ans.

Static RAM (sRAM)	Dynamic RAM (dRAM)
<ol style="list-style-type: none"> 1. Static RAM contains less memory cells per unit area. 2. The access times are lesser so they are fast memories. 3. sRAM consists of a number of flip flops. 4. Refreshing circuitry is not required. 5. Cost is more. 	<ol style="list-style-type: none"> 1. Dynamic RAM contains more memory cells per unit area. 2. The access times are greater than sRAMs. 3. dRAM stores data as charges on the capacitor. 4. Refreshing circuitry is required to maintain the charge on the capacitors after every few milliseconds. 5. Cost is less.

2. Consider a two level memory hierarchy (M_1, M_2). M_1 is directly connected to the CPU. Determine the average cost per bit (C) and the average access time (t_a) for the data given below :

Memory level	Capacity (S_i)	Cost (C_i)	Access Time (t_{ai})	Hit (H)
M_1 (Cache)	1024	0.1000	10^{-8}	0.9000
M_2 (Main)	2^{16}	0.0100	10^{-6}	—

Ans. (a) Average cost, $C = \frac{C_1 S_1 + C_2 S_2}{S_1 + S_2}$

$$= \frac{0.1 \times 1024 + 0.01 \times 2^{16}}{1024 + 2^{16}}$$

$$= 0.01138$$

(b) Average access time, $t_a = H t_{a_1} + (1 - H) t_{a_2}$

$$= 0.9000 \times 10^{-8} + (1 - 0.9000) 10^{-6}$$

$$= 1.09 \times 10^{-7}$$

3. Consider a two-level memory hierarchy, M_1 and M_2 . Denote the hit ratio of M_1 as h . Let C_1 and C_2 be the costs per KB, S_1 and S_2 be the memory capacities and t_1 and t_2 be the access times respectively.

- (a) Under what conditions will the average cost of entire memory system approaches C_2 ?
- (b) Find T_{eff} or t_a of this hierarchy.
- (c) Let $r = t_2/t_1$ be the speed ratio of two memories. Let $E = t_1/t_a$ be the access efficiency of the memory system. Express E in terms of r and h .
- (d) Plot E against h for $r = 5, 20, 100$ respectively.
- (e) What is the required hit ratio (h) to make $E > 0.95$ if $r = 100$?

[GGSIPU, M.Tech., 2nd Minor test ; 2004]

Ans. (a) The average cost is :

$$C = \frac{C_1 S_1 + C_2 S_2}{S_1 + S_2}$$

for C to approach C_2 , the conditions are $S_2 \gg S_1$ and $C_2 S_2 \gg C_1 S_1$.

(b) The effective access time is :

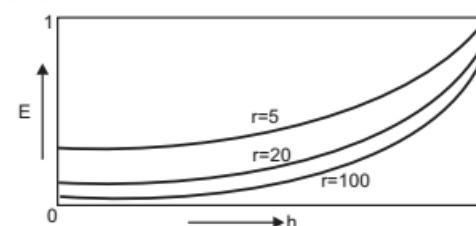
$$t_a = \sum f_i t_i$$

$$= h_1 t_1 + (1 - h_1) h_2 t_2$$

$$= h t_1 + (1 - h) t_2$$

(c) If $t_2 = r t_1$,
then $t_a = (h + (1 - h) r) t_1$
 $\therefore E = \frac{t_1}{t_a} = \frac{1}{(h + (1 - h) r)}$

(d) The plot is as follows :



(e) If $r = 100$, then we get :

$$E = \frac{1}{(h + (1-h) \times 100)} > 0.95$$

Solving this inequality we get this condition :

$$h = \frac{94}{94.05} \approx 99.95\%$$

4. You are asked to perform capacity planning for a two-level memory system.

M_1 (cache) has 3 choices – 64 KB, 128 KB and 256 KB.

M_2 (MM) has 4MB capacity

Given that : $C_1 = 20 C_2$
 $t_2 = 10 t_1$

Cache hit ratios are 0.7, 0.9 and 0.98 respectively for above three choices of M_1 .

(a) Find t_a in terms of $t_1 = 20$ ns.

(b) Find average byte cost of entire system if $C_2 = \$0.2/\text{KB}$.

(c) Compare the three memory designs ?

[GGSIPU, M.Tech. (CSE) 1st sem., Dec. 2008]

Ans. (a) The average access time is :

$$\begin{aligned} t_a &= h_1 t_1 + (1 - h_1) h_2 t_2 \\ &= ht_1 + (1 - h) 10t_1 \\ &= (10 - 9h) t_1 \end{aligned}$$

If $h = 0.7$ then $t_a = 3.7 t_1 = 74$ ns

If $h = 0.9$ then $t_a = 1.9 t_1 = 38$ ns

If $h = 0.98$ then $t_a = 1.18 t_1 = 23.6$ ns

(b) The average byte cost is :

$$\begin{aligned} C &= \frac{C_1 S_1 + C_2 S_2}{S_1 + S_2} \\ &= \frac{20C_2 S_1 + C_2 \times 4}{S_1 + 4000} \\ &= \frac{20 \times 0.2 S_1 + 0.2 \times 4000}{S_1 + 4000} \end{aligned}$$

$$C = \frac{4S_1 + 800}{S_1 + 4000}$$

For $S_1 = 64$, the average cost is 0.26

For $S_1 = 128$, the average cost is 0.32

For $S_1 = 256$, the average cost is 0.43.

(c) For the design choices, the product of average access time and average cost is 19.24, 12.16 and 10.15 respectively.

Therefore, the third option is the best choice.

5. Explain the following :

(a) Inclusion property.

(b) Coherence property.

(c) Write-through policy.

(d) Write-back policy.

(e) Paging.

(f) Segmentation ? [GGSIPU, M.Tech., 6th sem., May 2003]

- Ans.** (a) **Inclusion property** refers to the property that information present in a lower level memory must be a subset of that in a higher-level memory.
 (b) **Coherence property** requires that copies of an information item be identical throughout the memory hierarchy.
 (c) **Write-through policy** requires that changes made to a data item in a lower level memory be made to the next higher level memory immediately.
 (d) **Write-back policy** postpones the update at level $(i + 1)$ memory until the item is replaced or removed from level- i memory.
 (e) **Paging** divides the virtual memory and physical memory into pages of fixed sizes to simplify memory management and to remove fragmentation problem.
 (f) **Segmentation** divides the virtual address space into variable-size segments. Each segment corresponds to a logical unit. The main purpose of segmentation is to facilitate sharing and protection of information among programs.

6. Consider a two level memory hierarchy (M_1 and M_2). Their access times are t_1 and t_2 , costs per byte are C_1 and C_2 and capacities are S_1 and S_2 respectively.

$h_1 = 0.95$ at first level (M_1)

(a) Find t_{eff} of this memory system.

(b) Find the total cost of this system.

(c) Given : $t_1 = 20 \text{ ns}$ $t_1 = ?$

$S_1 = 512 \text{ KB}$ $S_2 = ?$

$C_1 = \$0.01/\text{byte}$ $C_2 = \$0.005/\text{byte}$

The total cost of the system is upper-bounded by \$15,000.

What are S_2 and t_2 ?

- Ans.** (a) The t_{eff} is given by :

$$\begin{aligned} t_{\text{eff}} &= t_1 h_1 + t_2 (1 - h_1) h_2 \\ &= t_1 h_1 + t_2 (1 - h_1) \\ &= 0.95 t_1 + 0.05 t_2 \end{aligned}$$

- (b) The total cost is :

$$C = C_1 S_1 + C_2 S_2$$

- (c) (1) $S_2 = ?$

Now, we have this inequality :

$$0.01 \times 512 \times 1024 + 0.0005 \times S_2 \leq 15000$$

$\therefore S_2$ cannot exceed 18.6 M bytes

- (2) $t_2 = ?$

Now, we get another inequality :

$$20 \times 0.95 + 0.05 \times t_2 \leq 40$$

$\therefore t_2 \leq 420 \text{ ns}$

7. Compare Symbolic and Numeric processing.

Ans.

Symbolic processing	Numeric processing
1. It uses lists, RDBMS, scripts, nets, frames as data objects.	1. It uses integer, floating point numbers, vectors, matrices as the data objects.
2. Locality of reference may not hold.	2. High degree of spatial and temporal localities.
3. Non-deterministic algorithm are used.	3. Deterministic algorithms are used.
4. Inputs can be graphical, audio from the keyboard.	4. Fast I/O is highly desirable.
5. Processors are called symbolic processors which work on large knowledge bases (KB).	5. Processors may be SIMD, MIMD or systolic array types.

8. Compare memory hierarchy on the basis of the following :

- (a) Access type
- (b) Capacity in bytes
- (c) Latency
- (d) Block size
- (e) Bandwidth ?

Ans.

Memory/Feature	CPU	Cache	M.M	Disk	Tape
1. Access type	Random access	Random access	Random access	Direct access	Sequential access
2. Capacity (bytes)	64-1024	8-256 KB	8-64 MB	1-10 GB	1TB
3. Latency	1-10 ns	20 ns	50 ns	10 ms	10 secs
4. Block size	1 word	16 words	16 words	4 KB	4 KB
5. Bandwidth	System clock rate	8 MB/s	1 MB/s	1 MB/s	1 MB/s

9. Given that in a two-level virtual memory :

$$\begin{aligned} t_{a1} &= 10^{-7} \text{ secs} \\ t_{a2} &= 10^{-2} \text{ secs} \end{aligned}$$

Find h so that the access efficiency e is atleast 90% ?

Ans. Given that :

$$\begin{aligned} t_{a1} &= 10^{-7} \text{ secs} \\ t_{a2} &= 10^{-2} \text{ secs} \\ e &= 90\% = 0.9 \\ h &= ? \text{ (hit ratio)} \end{aligned}$$

Now, access time ratio (r)

$$\begin{aligned} &= \frac{t_{a2}}{t_{a1}} \\ &= \frac{10^{-2}}{10^{-7}} \end{aligned}$$

$$\begin{aligned} &= 10^{-2 + 7} \\ &= 10^5 \\ \therefore e &= \frac{1}{r + (1 - r)h} \\ \text{or } 0.9 &= \frac{1}{10^5 + (1 - 10^5)h} \\ \text{or } h &= 0.9999 \end{aligned}$$

EXERCISE QUESTIONS

1. Discuss memory hierarchy technology. Explain inclusion, coherence and locality properties.

[UPTU, B. Tech (CSE) 8th Sem.; 2008-09 & GGSIPU, B.Tech. 7th sem ; Dec. 2007]

2. (a) Describe inclusion, coherence and locality of reference properties of a memory hierarchy.
(b) Explain various address translation mechanism in a virtual memory.

[GGSIPU, M.Tech. 1st sem ; Dec. 2004]

3. In a 2 level memory system there are 8 virtual pages on a disk to be mapped on the 4 page frames in the MM. A certain program generates the following page trace :

1, 0, 2, 7, 1, 7, 6, 7, 0, 1, 2, 0, 6

Compute the page faults and the hit ratio. Assume page frames (PF's) are initially empty.

[GGSIPU, M.Tech. 1st Minor ; 2004]

4. Describe TLB as a part of virtual memory technology.

[GGSIPU, M.Tech. 1st Minor ; 2004]

5. Compare private and globally shared virtual memory ?

6. Write short notes on :

(a) Working set (b) 90 – 10 rule.

7. Prove that :

$$e = \frac{1}{h + (1 - h)r}$$

Where e , h and r have their usual meanings. Assume a two level memory hierarchy as M_1 and M_2 .

[Hint :

t_A (average time for CPU to access a word) is given by :

$$t_A = ht_{A1} + (1-h) t_{A2} \quad \dots(1)$$

If t_B is block transfer time then time to access M_2 is given by :

$$t_{A2} = t_B + t_{A1} \quad \dots(2)$$

Substituting equation (2) in (1) we get :

$$\begin{aligned} t_A &= ht_{A1} + (1 - h) (t_B + t_{A1}) \\ &= ht_{A1} + (1 - h) t_B + (1 - h) t_{A1} \\ &= ht_{A1} + (1 - h) t_B + \frac{t_{A1}}{h} - ht_{A1} \\ &= t_{A1} + (1 - h) t_B \end{aligned}$$

In most cases, $t_{A2} \gg t_{A1}$

$\therefore t_{A2} \sim t_B$ and we have

$$t_{A1} + (1 - h) t_{A2} = t_A$$

if r = access time ratio (of 2 level memory) = $\frac{t_{A2}}{t_{A1}}$

then, e = access efficiency = $\frac{t_{A1}}{t_A}$

$$= \frac{t_{A1}}{ht_{A1} + (1 - h)t_{A2}}$$

Dividing numerator and denominator by t_{A1} we get :

$$e = \frac{1}{h + (1 - h)r} \quad \left[\because r = \frac{t_{A2}}{t_{A1}} \right]$$

$$= \frac{1}{h + r - rh} = \frac{1}{r + (1 - r)h} \quad \text{Hence proved}]$$

8. Explain how instruction set, memory hierarchy, compiler technology, CPU implementation and control affect the CPU performance and justify the effect in terms of program length, clock rate and effective CPI.

[UPTU, B. Tech (CSE) 8th Sem.; 2003-04]



CHAPTER

6 BACKPLANE BUS SYSTEM

6.0 INTRODUCTION

When several devices fight for the same bus then contentions may arise. Only one of the devices may be granted access at a time. It has been observed that :

$$\text{B.W. effective} \quad \frac{1}{\text{(with each processor)}} \propto \frac{1}{\text{no. of PEs fighting for the bus}}$$

That is why bus systems are usually small in size.

6.1 BACKPLANE BUS SPECIFICATION

A back plane bus interconnects PEs, storages and peripheral devices in a tightly coupled system where there is high degree of Resource sharing. The communication is done by the bus following some protocols. A backplane bus system is shown in Fig. 6.1

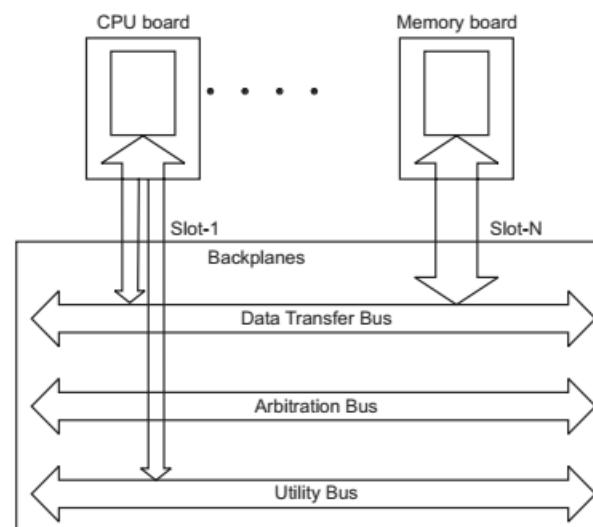


Fig. 6.1. Backplane Bus System.

As is clear from fig 6.1 above that various functional boards are plugged into the slots on the backplane. Each slot has one or more connectors for inserting the boards.

Data, address and control lines together form a **data transfer bus (DTB)**. The process of assigning control of the DTB to a request is called as **arbitration**. Interrupt lines will handle the interrupts, if any.

There are two types of **printed circuit boards (PCBs)** that are connected to a bus :

- (a) Active boards.
- (b) Passive boards.

Active boards can act as bus masters or slaves at different times.

For example : Processors

Passive boards can act only as slaves.

For example : Memory boards.

The master can initiate a bus cycle and the slaves respond to request made by a master.

A **broadcast** is a read operation involving multiple slaves placing their data on the bus lines. A **broadcast** is a write operation involving multiple slaves.

The bus timing may be synchronous or asynchronous. A **synchronous data transfer** is shown in Fig. 6.2

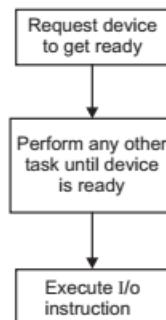


Fig. 6.2. Flowchart of synchronous data transfer.

An **asynchronous data transfer** has some steps as shown below in Fig 6.3.

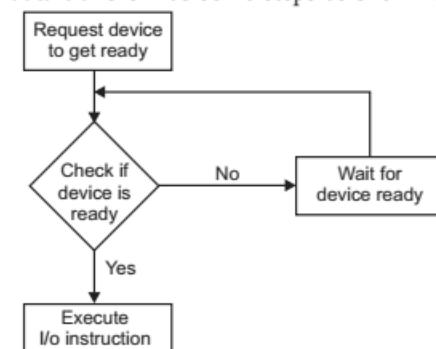


Fig. 6.3. Flowchart of asynchronous data transfer.

An overall asynchronous bus needs no fixed clock cycle. Asynchronous bus is more flexible.

6.2 ARBITRATION SCHEMES

The process of selecting the next bus master is known as arbitration. The duration of a master's control of the bus is called **bus tenure**. The different bus arbitration schemes are :

- (a) Daisy-chained bus arbitration.
- (b) Independent Requests and Grants.
- (c) Distributed Arbitration.

Let us discuss these schemes in detail now.

I. Daisy-chained bus arbitration

This is also called as a **central arbitration scheme**. It uses a central bus arbiter or bus controller. In this technique, the fixed priorities are assigned to all units or devices according to their physical location from the bus controller.

Please note that the unit physically closest to the bus is assigned the highest priority. Fig. 6.4 shows a daisy chain scheme.

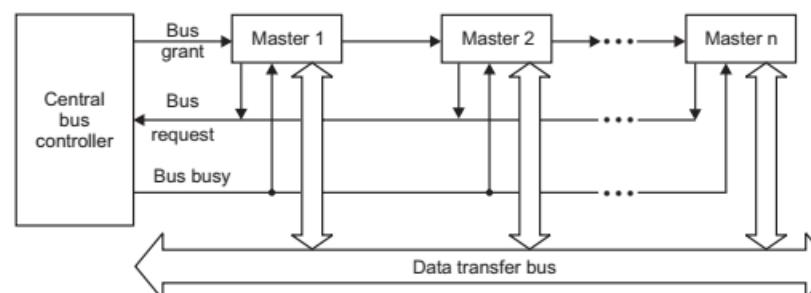


Fig. 6.4. Daisy-chain.

Observe that the potential masters are **daisy-chained in a cascade**. A fixed priority is set in a daisy-chain from left to right. Whenever a higher-priority device fails, all the lower priority devices on the right of the daisy-chain cannot use the bus. Please note that we need to bypass a failing device or remove the device from the daisy-chain itself.

The bus transaction timing are shown in Fig. 6.5 below :

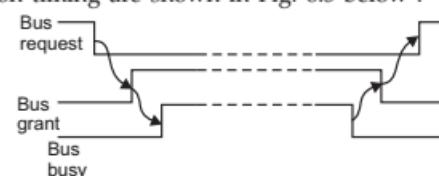


Fig. 6.5. Daisy-chain bus timing.

Each master can send a bus request. However, all requests share the same bus-request line. As shown in Fig 6.5, the bus is busy only when the Grant of Bus is given and grant is given if a bus is requested. The reverse process occurs when the bus is released.

Advantages of Daisy-chaining :

1. Simplicity.
2. Low cost.
3. It requires the minimum number of control lines irrespective of the expansion of the system or number of master units.

Drawbacks of Daisy-chaining :

1. The priorities of the units are dependent on their physical location.
2. The bus arbitration is slow because of the propagation delay of bus grant signal.
3. When the system is expanded, the propagation delay goes on increasing for the newly added units.
4. The failure of one unit breaks down the system ahead from failed unit.

II. Independent Requests and Grants

It is also possible to have independent multiple bus-request and bus-grant signal lines for each potential master.

In this scheme, no daisy-chaining is used. Still we have a common central arbiter.

Fig. 6.6. shows an independent requests with a central arbiter.

Advantages of this scheme are as follows :

1. More flexibility is provided by bus lines.
2. Faster arbitration as compared to daisy-chaining.

Disadvantages of this scheme

Large number of arbitration lines are required.

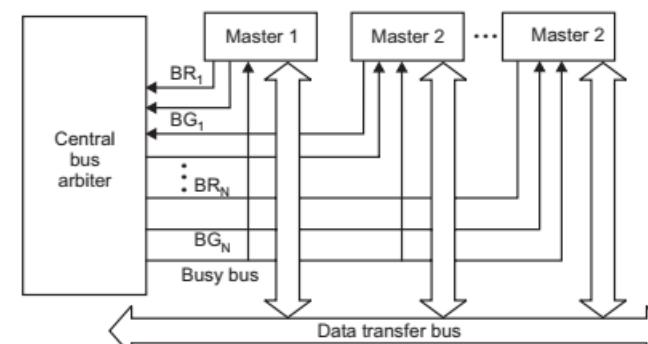


Fig. 6.6. Independent Requests method.

III. Distributed Arbitration

In this scheme, each Master has its own arbiter. Each of these arbiters is given a unique arbitration number (*id*). This *id* is used to resolve arbitrations. When two or more devices compete for the bus, the winner is the one whose arbitration number is largest.

A distributed arbiter scheme is shown Fig. 6.7.

BG – Bus Grant.

BB – Bus Busy.

AN – Arbitration Number.

All masters (1 to N) send their arbitration numbers to the shared bus request grant (SBRG) lines on the arbitration bus via their respective arbiters. Each arbiter compares

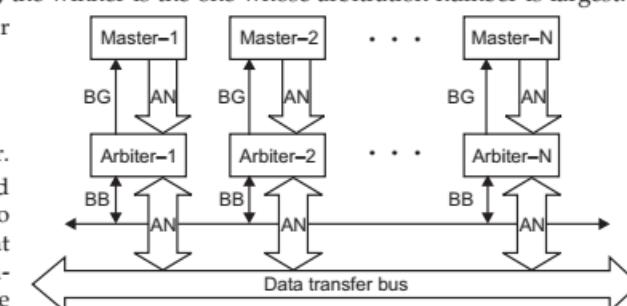


Fig. 6.7. Distributed arbiter scheme.

the resulting number on the SBRG lines with its own arbitration number. If the SBRG number is greater, the requester is dismissed. At the end, the winner's arbitration number remains on the arbitration bus. After the current bus transaction is completed, the winner takes the control of the bus. Please note that this policy is also priority based.

For example : Multi bus II and Future bus use such a scheme.

6.3 INTERRUPT

Any request from I/O or other devices to a PE for some service is called as an interrupt. A priority interrupt bus is used to pass the interrupt signals. The status and the identification information is to be saved. An interrupt handler module is a module/function that handles this interrupt. Interrupts can be handled by message passing using the data bus lines on a time sharing basis. The use of time-shared data bus lines to implement interrupts is called virtual interrupt.

6.4 CACHE ADDRESSING MODELS

The transfer of information from main memory (MM) to cache memory is conducted in units of cache blocks (or cache lines)

4 block placement schemes are as follows :

- (a) Direct mapping cache.
- (b) Fully Associative cache.
- (c) Set-Associative cache.
- (d) Sector mapping cache.



Fig. 6.8. MM and Cache memory.

Blocks in caches are called as 'block frames' in order to distinguish them from 'blocks' of MM.

Block-frames (of cache) are denoted as \bar{B}_i for $i \leftarrow 0$ to m .

& Blocks (of MM) are denoted as B_j for $j \leftarrow 0$ to n .

Now, various mappings can be defined from the set $\{B_j\}$ to $\{\bar{B}_i\}$

Also, we assume that $n \gg m$ where $n = 2^s$ and $m = 2^r$

Each block (of MM) or block frame (of cache) is assumed to have b -words where $b = 2^w$.

$$\begin{aligned} \therefore \quad \text{Cache has} &= m.b \text{ words} \\ &= 2^r \cdot 2^w \text{ words} \\ &= 2^{r+w} \text{ words} \end{aligned}$$

$$\begin{aligned} \text{and} \quad \text{M.M. has} &= n.b \text{ words} \\ &= 2^s \cdot 2^w \text{ words} \\ &= 2^{s+w} \text{ words} \end{aligned}$$

Where these words have $(s + w)$ bits, when the block frames are divided into $v = 2^t$ sets, then

$$\frac{m}{v} = \frac{2^r}{2^t} = 2^{r-t} = K \text{ blocks are in each set.}$$

6.4.1 Direct-mapping Cache

* It is based on direct mapping of $\frac{n}{m} = \frac{2^s}{2^r} = 2^{s-r}$ memory blocks to one block frame in cache.

* Block B_j of MM is mapped to block-frame, \bar{B}_i of cache, if $i = j \bmod 4$. i.e., $B_j \rightarrow \bar{B}_i$ if $i = j \bmod 4$

There is a unique block frame \bar{B}_i that each B_j can load into. There is no way to implement block replacement policy.

This direct mapping is very rigid but is the simplest cache organization to implement.

Direct mapping is shown next :

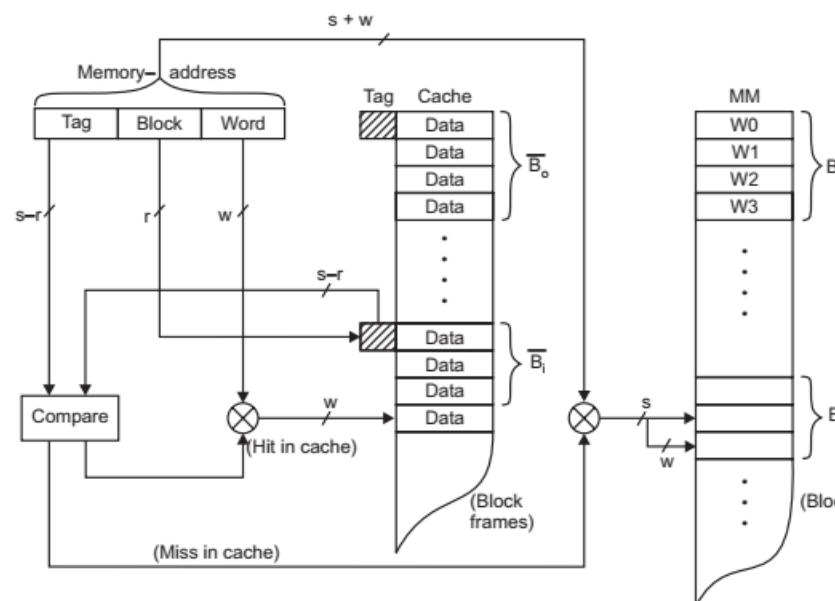


Fig. 6.9. Cache/memory addressing.

Working : We assume a case where each block contains four words (i.e., $w = 2$ bits).

Now, the memory address is divided into three fields :

w bits \rightarrow word offset within each block

s bits \rightarrow specify address in M.M.

$(s - r)$ bits (leftmost) \rightarrow The tag to be matched.

Once the block \bar{B}_i is uniquely identified by this field, the tag associated with the

addressed block is compared with the tag in the memory address.

A cache hit occurs when the two tags match. Otherwise a cache miss occurs. In case of a cache hit, the word offset is used to identify the desired data word within the addressed block.

When a miss occurs, the entire memory address ($s + w$ bits) is used to access the MM. The first 5 bits locate the addressed block and the lower w bits locate the word within the block.

6.4.2 Fully Associative Cache

In this method, the tags are compared with all block tags in the cache.

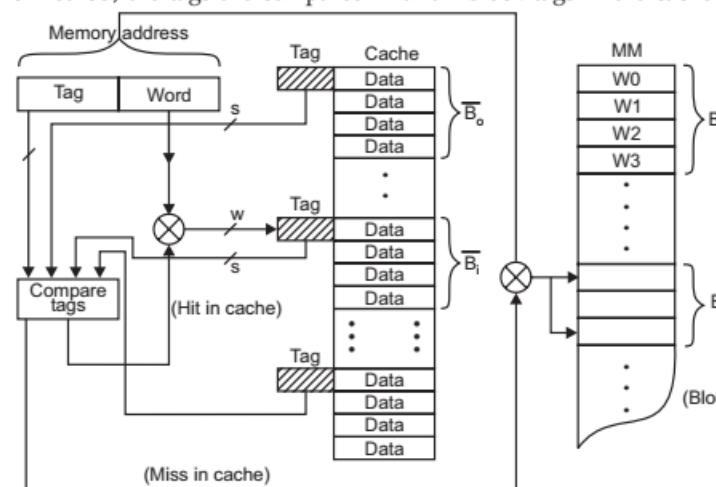


Fig. 6.10. Fully associative cache.

Herein, comparison of all tags will be done. But this comparison is very time consuming. Thus CAM – content addressable memory is needed to achieve a parallel comparison of tags. This is a bit costlier.

The advantage of this method is to allow the implementation of better block replacement policy with reduced block contention.

6.4.3 Set Associative Cache

In this scheme, sets of blocks of cache are formed.

A block B_j can be mapped into anyone of the available frames $\overline{B_f}$ in a set S_i defined below. It is the matched tag that identifies the current block which resides in the frame.

So, $B_j \rightarrow \overline{B_f} \in S_i$ if $j \pmod v = i$

If k represents set size,

m represents cache block frames

Then No. of sets of blocks, $v = \frac{m}{k}$

Where k may be 2, 4, 8, 16 or 64 (depending on need)

For e.g.,

If $k = 2$ (i.e., 2 blocks in each set)

$m = 7$ (i.e., total no. of blocks-frame in cache)

$$\text{Then } v = \frac{7}{2} = 4$$

i.e., there will be 4 sets of 2 blocks (as $k = 2$)

Also, $2^d = v$ (d -bit set number is used to identify each set)

or $2^d = 4$ (as above)

$$= 2^2$$

∴ $d = 2$ for our e.g.,

∴ 2 bit tag is required.

So,

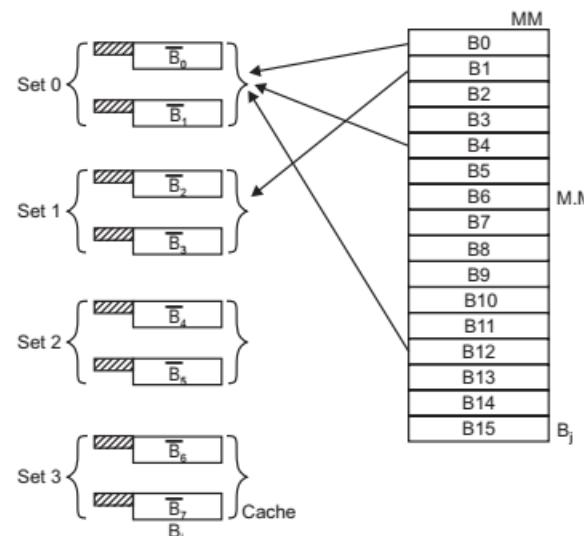


Fig. 6.11. Set associative cache.

∴ Here $k = 2$

$m = 7$

$$\therefore V = \frac{m}{k} = \frac{7}{2} = 4$$

if $j = 0$

then $(0 \bmod v) = (0 \bmod 4) = 0$; So, B_0 goes into S_0 .

6.4.4 Sector Mapping Cache

- In this scheme, we partition both the cache and main memory (MM) into fixed-size sectors.

- Each sector of MM can be placed in any of the **available** sector-frames. (of cache)

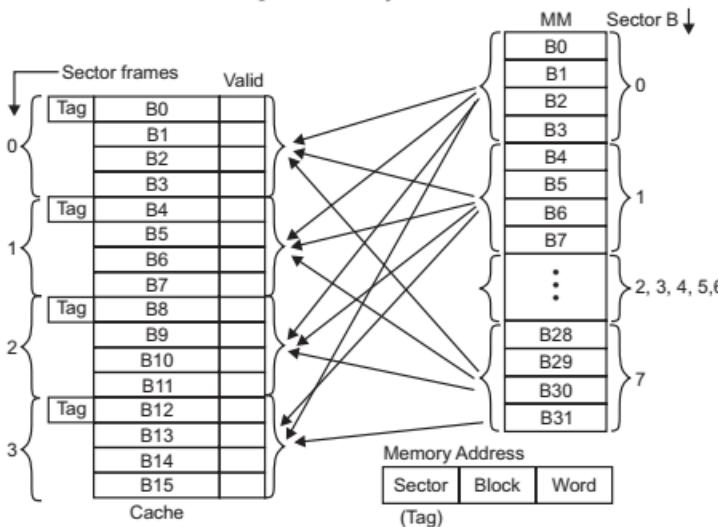


Fig. 6.12. Sector mapping cache.

- If a matched sector-frame is found (a **cache hit**), the block field is used to locate the desired block within the sector frame.
- If a **cache miss occurs**, only the missing block is fetched from the MM and brought into a similar block frame in an available sector.

So, i^{th} block in a sector \rightarrow is placed into i^{th} block frame in a destined sector frame

Also, a valid bit is attached to each block frame to indicate whether the block is valid or invalid. When the contents of a block frame are replaced, the remaining block frames in the same sector are marked invalid. Only the **most recently replaced block frame in a sector** is marked valid for reference.

We are in a position to solve some problems now.

Example 6.1. Consider system with main memory (MM) consisting of 4 K blocks a cache memory consisting of 128 blocks and a block size of 16 words.

Tag	Cache	Main Memory (MM)
3	0 384	0 128 256 384 3968
1	1 129	1 129 257 385
0	2	2 130 258 386
:	⋮	⋮
26	26	⋮
31	127 4095	127 255 383 4095
		0 1 2 3 31

Fig. 6.13. Mapping main memory blocks to cache blocks.

What will be the word field, block field and Tag field length? How many bits are there in the main memory address?

Solution. It is crystal clear from the figure given in the question that there are a total of 32 main memory blocks that map to a given cache block. Like, main memory blocks 0, 128, 256, 384 ... 3968 map to cache block 0. *That is why the direct mapping technique is also called as many-to-one mapping technique.* The main *advantage* of the direct-mapping technique is its simplicity in determining where to place an incoming main memory block in the cache.

Its main *disadvantage* is the inefficient use of the cache. This is so because as per this technique, a number of main memory blocks may compete for a given cache block even if there exist other empty cache blocks. This disadvantage should lead to achieving a low cache hit ratio.

According to the direct-mapping technique, the MMU (Memory Management Unit) interprets the address issued by the processor by dividing it (address) into 3-fields as follows:

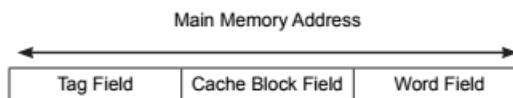


Fig. 6.14. Direct-mapped address fields.

∴ The length (in bits) of each of the fields are—

- (a) Word field = $\log_2 B$, where B is the size of block in words ...(1)
- (b) Block field = $\log_2 N$, where N is the size of cache in blocks. ...(2)
- (c) Tag field = $\log_2 (M/N)$, where M is the size of MM in blocks. ...(3)
- (d) Number of bits in the main memory address = $\log_2 (B \times M)$...(4)

Please note that the total number of bits as computed by first three equations (1), (2) and (3) should add up to the length of the main memory address. This is a test to check whether your result is correct or not.

∴ Given : $B = 16$, $N = 128$

We get—

- (a) Word field = $\log_2 B = \log_2 16 = \log 2 \cdot 2^4 = 4$ bits.
- (b) Block field = $\log_2 N = \log_2 128 = \log 2 \cdot 2^7 = 7$ bits.
- (c) Tag field = $\log_2 (M/N)$
 $= \log_2 (2^2 \times 2^{10} / 2^7)$
 $= 5$ bits.
- (d) Number of bits in the main memory address
 $= \log_2 (B \times M)$
 $= \log_2 (2^4 \times 2^{12})$
 $= 16$ bits.

Example 6.2. Calculate the above given parameters in Q 1 for a memory system having the following specifications—

- Size of MM = 4 K blocks.
- Size of cache = 128 blocks.
- Block size = 16 words.

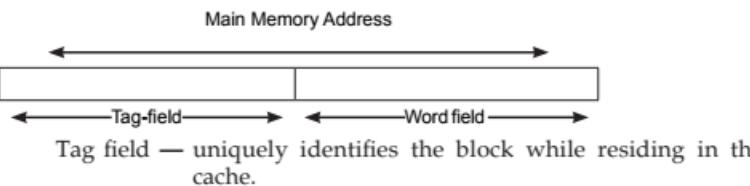
Assume that the system uses associative mapping.

Solution. For associative mapping,

- (a) Word field = $\log_2 B = \log_2 16$
 $= \log 2 \cdot 2^4$
 $= 4 \text{ bits}$
- (b) Tag field = $\log_2 M = \log_2 2^7 \times 2^{10}$
 $= 12 \text{ bits}$
- (c) No. of bits in MM address
 $= \log_2 (B \times M)$
 $= \log_2 (2^4 \times 2^{12})$
 $= 16 \text{ bits}$

Please note the following points regarding fully associative mapping—

1. An incoming MM block can be placed in any available cache block
2. The address issued by the processor need only have two fields—



It is MMU hardware that interprets the address issued by the processor by dividing it into two fields as shown in figure.

3. Also note that the total number of bits as computed by the first two equations should add up to the length of the main memory address. This can be used as a check for correctness of your computations.

Example 6.3. Compute the three parameters—word, set and tag for a memory system having the following characteristic—

- Size of MM = 4K blocks
- Size of cache = 128 blocks
- Block size = 16 words.

Assume that the system uses set-associative mapping with four block per set.

Solution. Again in set-associative mapping,

- (a) Word field = $\log_2 B$, where B is the size of block in word
- (b) Set field = $\log_2 32 = 5 \text{ bits}$
- (c) Tag field = $\log_2 (4 \times 2^{10}/32) = 7 \text{ bits}$
 \therefore Number of bits in the MM

$$\begin{aligned} \text{Address} &= \log_2 (B \times M) \\ &= \log_2 (2^4 \times 2^{12}) \\ &= 16 \text{ bits.} \end{aligned}$$

Note : Why are we talking about blocks and not whole sector because the memory requests are destined for blocks & not for sectors.

6.5 CACHE PERFORMANCE ISSUES

Various cache performance issues includes :

- (a) Cycle count.
- (b) Hit ratio.
- (c) Effect of block size.
- (d) Effect of set number.

Tradeoffs exist between these issues.

6.5.1 Cycle counts

The cache speed depends on :

- (a) Underlying static or dynamic RAM technology
- (b) Cache organization and
- (c) Cache hit ratios.

As shown in Fig. 6.15 below, the graph is between cycle count (y-axis) and cache size, set number and block size (on x-axis).

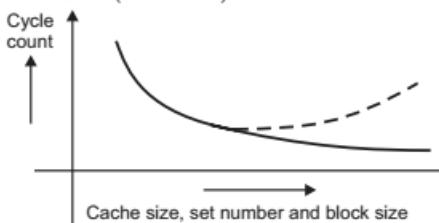


Fig. 6.15. Cycle count versus cache size and other parameter.

The cycle count decreases almost linearly with the increasing value of the above cache parameters. Please note here that the decreasing trend becomes flat and after a certain point turns into an increasing trend as shown by the dashed line.

6.5.2 Hit Ratios

As shown in Fig. 6.16, hit ratio increases with respect to increasing cache size.

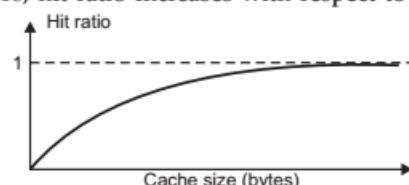


Fig. 6.16. Hit ratio v/s cache size.

Please note that when the cache size approaches infinity, a 100% hit ratio is expected. However, this never happens. It is because the cache size is always bounded by a limited budget. Graph of above Fig. 6.16 can be given by equation $1-C^{-0.5}$, where C is the total cache size. Initial cache loading and change in locality also prevent such an ideal performances.

6.5.3 Block sizes

Let us now draw a graph between Hit ratio (y axis) and block size (bytes) on x-axis. This is shown below in Fig. 6.17

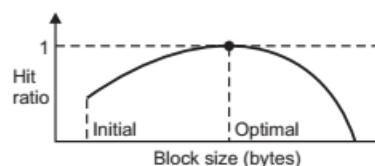


Fig. 6.17. Hit ratio v/s block size.

As the block size increases, initially, the hit ratio also increases. This is because of spatial locality in referencing larger instruction/data blocks. Hit ratio then reaches its peak value at a certain block size. After this point as shown in Fig. 6.25, the hit ratio decreases with further increase in the block size. This is because of the mismatch between program behaviour and block size.

Please note that the hit ratio become zero (0) when the block size equals the entire cache size. For a bus system, Smith in 1997, found out that a optimum block size should be chosen to minimum the effective memory access time. It further depends on the ratio of the access latency and bus cycle time (data transfer rate).

6.5.4 Set numbers

The hit ratio may decrease as the number of sets increases. As the set number increases from 32 to 256, the decrease in hit ratio is smaller. When the set number increases to 512 and beyond, the hit ratio decreases faster.

6.5.5 Other performance factors

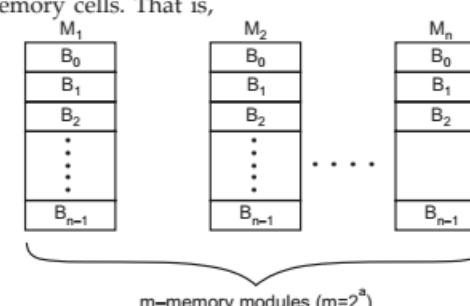
Now, we know that tradeoffs exist among the cache size, set number, block size and the memory speed. Multilevel caches also offer options for expanding the cache effects. An optimal cache hierarchy design is a must.

6.6 INTERLEAVED MEMORY ORGANIZATION

In this technique, the system memory is divided into a number of memory modules and arranges addressing so that successive words in the address space are placed in different modules. Our objective is to broaden the **effective memory bandwidth** so that more memory can be accessed per unit time. We need to match the memory bandwidth with the bus bandwidth and with the processor bandwidth.

What is Memory Interleaving ?

Consider a main memory with m memory modules wherein $m = 2^a$. Each has $w = 2^b$ words of memory cells. That is,



$$\begin{aligned}\therefore \text{Total memory capacity} &= m \times w \\ &= 2^a \times 2^b \\ &= 2^{a+b} \text{ words.}\end{aligned}$$

These memory words are assigned linear addresses.

There are two address formats for memory interleaving :

1. Low-order interleaving.
2. High-order interleaving.

Let us discuss these formats one by one.

I. **Low-order interleaving** spreads the contiguous memory locations across the m modules horizontally. This is shown in Fig. 6.18 below :

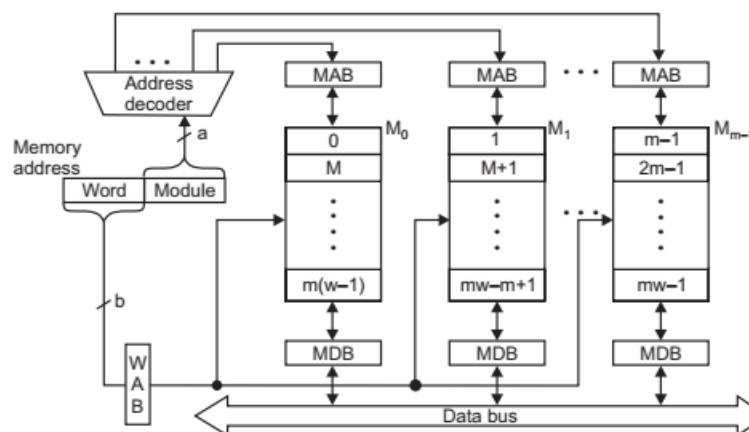


Fig. 6.18. Low order m -way interleaving.

MAB : Module address buffer.

MDB : Memory data buffer.

WAB : Word address buffer.

The **low-order a -bits** of the memory address are used to identify the memory module.

The **high-order b -bits** are the word addresses within each module. **Please note that the same word address is applied to all memory modules simultaneously.** An address decoder is used to distribute module addresses.

II. **High-order interleaving** : It uses the **high-order a -bits** as the module address and the **low-order b -bits** as the word address within each module. Contiguous memory locations are thus assigned to the same memory module. In each memory cycle, only one word is accessed from each module. So, this type of interleaving cannot support block access of contiguous locations. It is shown in Fig. 6.19 below :

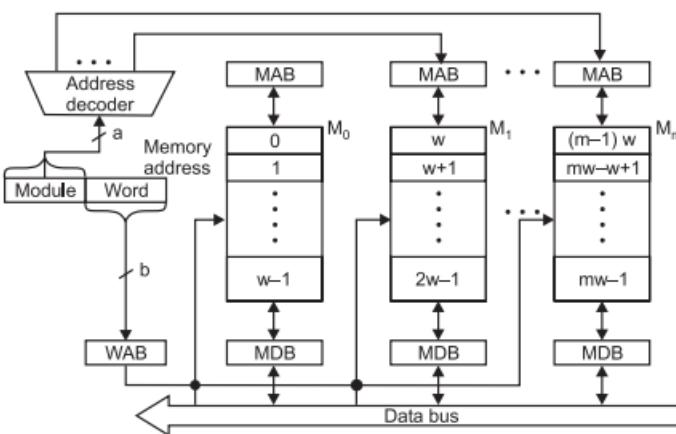


Fig. 6.20.

MAB : Module address buffer.

MDB : Memory data buffer.

WAB : Word address buffer.

Please note that here, there is reversing of addresses. Earlier, in low-order interleaving.

"Low-order *a*-bits of memory address are used to identify the memory module and the high-order *b*-bits are the word addresses within each module". On the other hand, in high order interleaving,

"High-order *a*-bits are used as the module address and the low-order *b*-bits as the word address within each module".

Note that a single memory module is assumed to deliver one word per memory cycle and thus has a bandwidth of 1.

What is memory bandwidth ?

The memory bandwidth (*B*) of an *m*-way interleaved memory is upper-bounded by *m* and lower bounded by

- According to Hellerman, for uniprocessor,

$$B = m^{0.56} \approx \sqrt{m}$$

Where *m* is the number of interleaved memory modules.

If $m = 16$ then $B = \sqrt{16} = 4$

This means that if 16 memory modules are used, then the effective memory bandwidth is four times that of a single module.

What is fault tolerance?

It is easier to locate faulty memory modules in a memory bank of *m*-memory modules due to memory interleaving. However, this fault isolation cannot be carried out in low-order interleaved memory in which a module failure may affect the entire memory bank. Thus, **low-order interleaving memory is not fault-tolerant**.

Pipelined Memory Access

It is also possible to overlap '*m*' memory modules in a pipelined fashion. So, the memory cycle or the **major cycle** is subdivided into '*m*' **minor cycles**.

For eg : An eight way inter based memory (with $m = 8$ and $w = 8$ and thus $a = b = 3$) is shown in Fig. a.

Let ' θ ' be the major cycle

' τ ' be the minor cycle.

Then, these two cycle times are related as follows:

$$\tau = \theta/m$$

Where ' m ' is degree of interleaving. Fig (b) shows the timing of the pipeline accesses of the eight contiguous memory words. *Please note note that this type of concurrent access of contiguous words is called as C-access memory scheme.*

We define *major cycle (θ)* and *minor cycle (τ)* now.

Major cycle (θ): It is defined as the total time required to complete the access of a single word from a module.

Minor cycle (τ): It is defined as the actual time needed to produce word, assuring overlapped access of successive memory modules separated in every minor cycle, τ .

Also note that the pipelined access of the block of or consecutive words is sandwiched between other pipelined block access before and after the present block. Even if the total block access time is 20, the effective access time of each word is reduced to τ as the memory is contiguously accessed in a pipeline fashion.

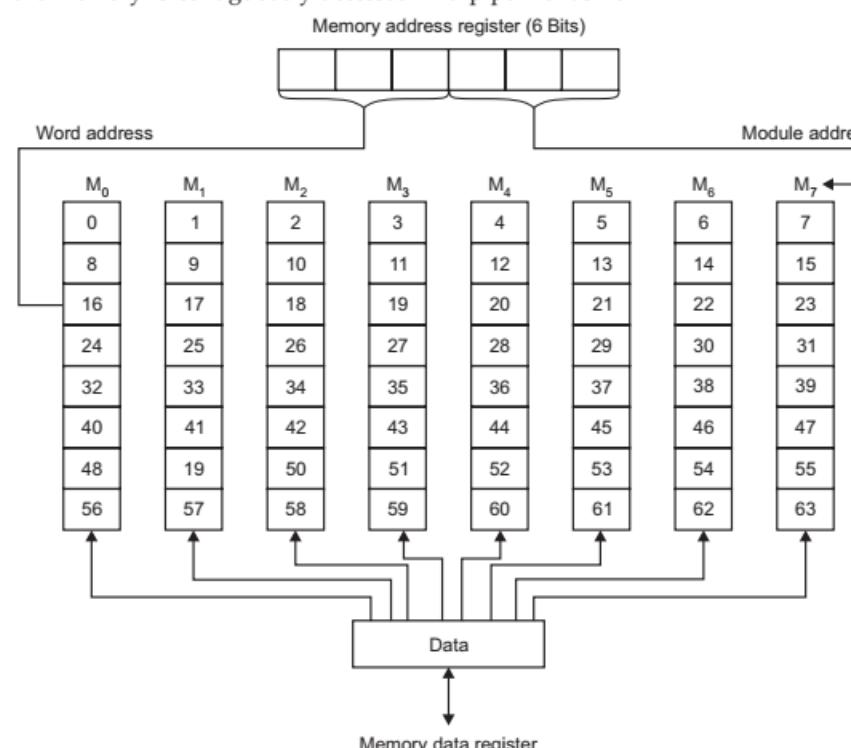


Fig.6.21.(a) 8-way low-order interleaving (Absolute address shown in each memory word)

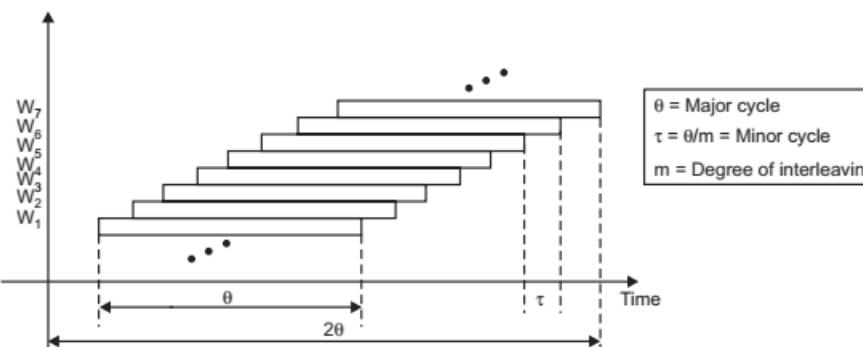


Fig.6.21.(b) Pipelined access of 8 consecutive words in C-access memory.

6.7 MULTI-CORE ARCHITECTURES & CACHE COHERENCE PROBLEM

A Multi-Core Architecture is a method of embedding a number of cores on a single chip. A Multi Core Architecture improves the performance of a system by computing a number of tasks at the same time.

Single-Core CPU chip

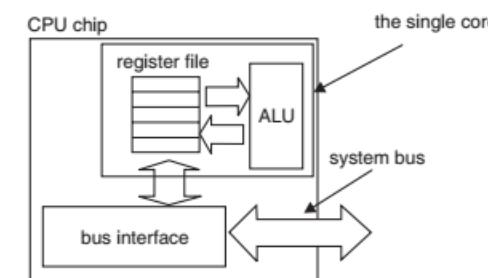


Fig. 6.22. Single Core CPU chip.

Multi-Core architectures

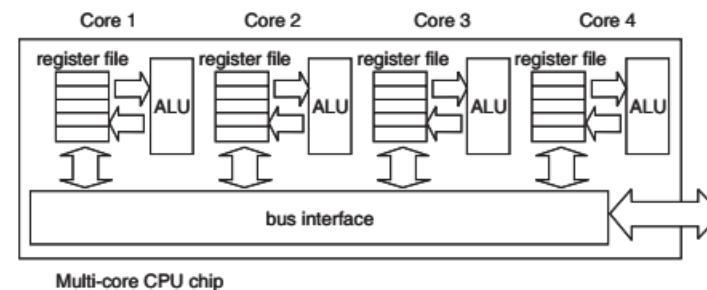
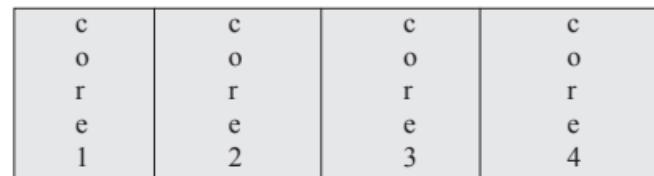
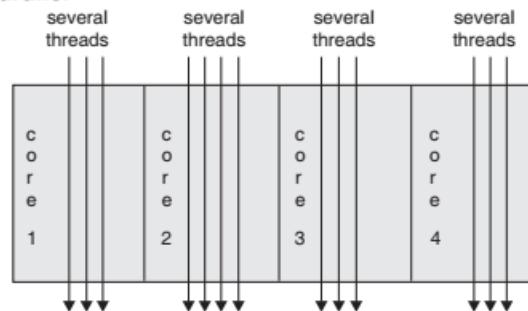


Fig. 6.23. Multi-core CPU chip

Multi-Core CPU chip

- The cores fit on a single processor socket.
- Also called CMP (Chip Multi-Processor)

**The cores run in parallel****Interaction with the Operating System**

- OS perceives each core as a separate processor.
- OS scheduler maps threads/processes to different cores.
- Most major OS support multi-core today: Windows, Linux, Mac OS X, ...

Advantages of multi-core

- Difficult to make single-core clock frequencies even higher.
- Deeply pipelined circuits:
 - heat problems.
 - difficult design and verification.
 - large design teams necessary.
 - server farms need expensive air-conditioning.
- Many new applications are multithreaded.
- General trend in computer architecture (shift towards more parallelism).

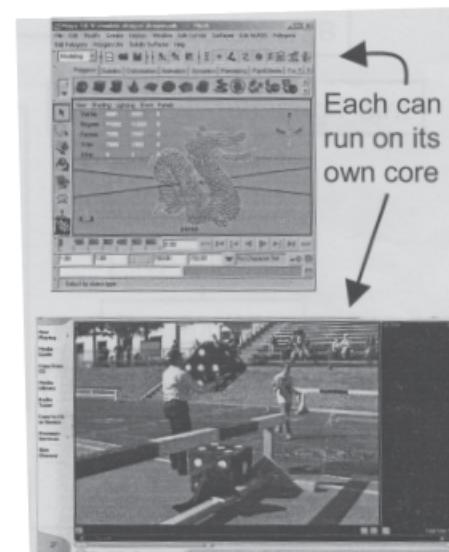
Parallelism

- **Instruction-level parallelism**
 - Parallelism at the machine-instruction level.
 - The processor can re-order, pipeline instructions, split them into microinstructions, do aggressive branch prediction, etc.
 - Instruction-level parallelism enabled rapid increases in processor speeds over the last 15 years.

- **Thread-level parallelism (TLP)**
 - This is parallelism on a more coarser scale.
 - Server can serve each client in a separate thread (Web server, database server).
 - A computer game can do AI, graphics, and physics in three separate threads.
 - Single-core superscalar processor cannot fully exploit TLP.
 - Multi-core architectures are the next step in processor evolution : explicitly exploiting TLP.

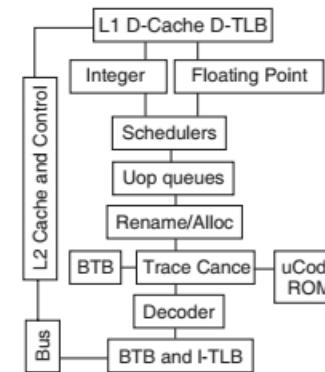
Applications benefit from multi-core

- Database servers.
- Web servers (Web commerce).
- Compilers.
- Multimedia applications.
- Scientific applications, CAD/CAM.
- In general, these are applications with Thread-level parallelism (as opposed to instruction-level parallelism)



A technique complementary to multi-core : Simultaneous multithreading

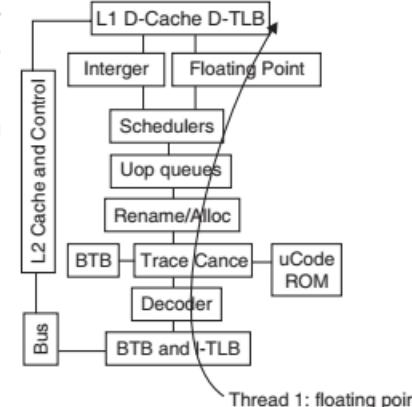
- Problem addressed: The processor pipeline can get stalled:
 - Waiting for the result of a long floating point (or integer) operation
 - Waiting for data to arrive from memory
 - Other execution units wait unused.



Simultaneous multithreading (SMT)

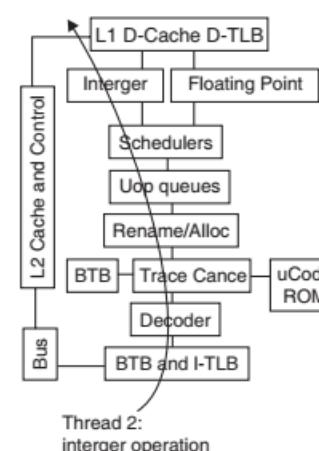
- Permits multiple independent threads to execute SIMULTANEOUSLY on the SAME core
- Weaving together multiple “threads” on the same core
- Example : if one thread is waiting for a floating point operation to complete, another thread can use the integer units.

Without SMT, only a single thread can run at any given time



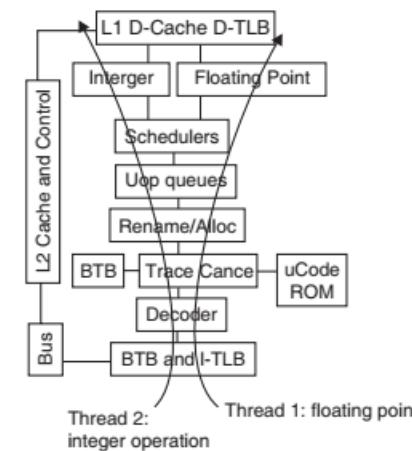
Thread 1: floating point

Without SMT, only a single thread can run at any given time

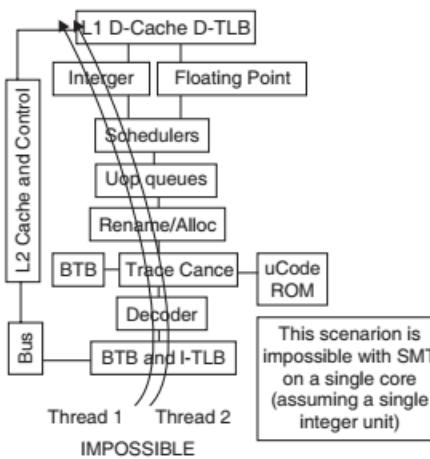


Thread 2:
integer operation

SMT processor : both threads can run concurrently



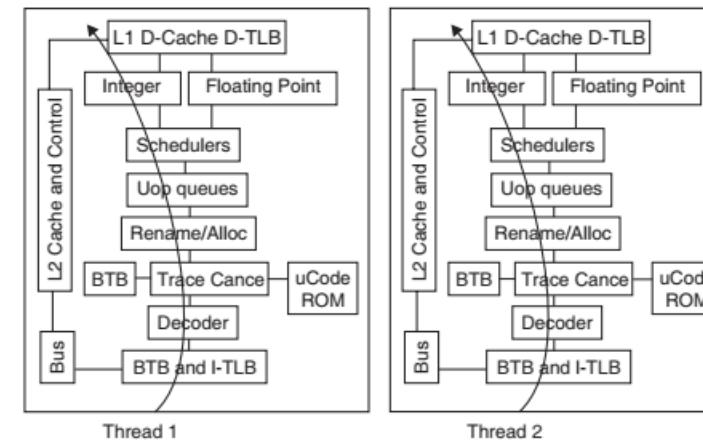
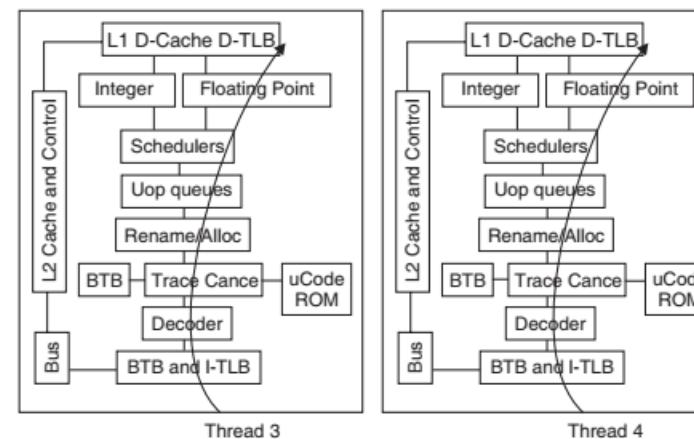
But: Can't simultaneously use the same functional unit



SMT not “true” parallel processor

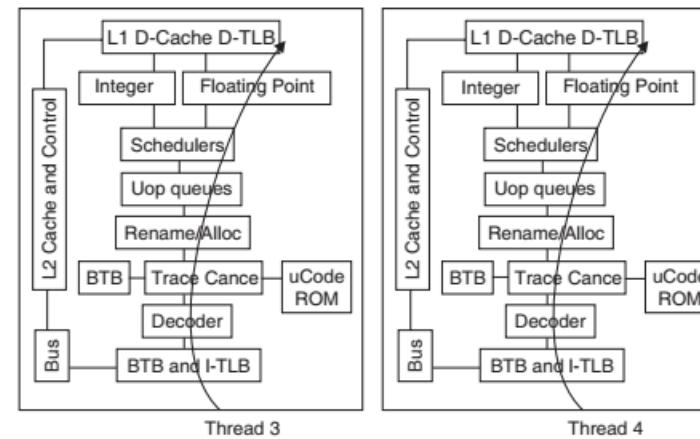
- Enables better threading (e.g. up to 30%).
- OS and applications perceive each simultaneous thread as a separate “virtual processor”.
- The chip has only a single copy of each resource.
- Compare to multi-core : each core has its own copy of resources.

Multi-core: threads can run on separate cores

**Multi-core: threads can run on separate cores****Combining Multi-core and SMT**

- Cores can be SMT -enabled (or not)
- The different combinations :
 - Single-core, non-SMT : standard uniprocessor
 - Single-core, with SMT
 - Multi-core, non-SMT
 - Multi-core, with SMT : our fish machines.
- The number of SMT threads: 2, 4, or sometimes 8 simultaneous threads
- Intel calls them "hyper-threads".

SMT Dual-core : all four threads can run concurrently



Comparison : multi-core VS SMT

- Multi-core:
 - Since there are several cores, each is smaller and not as powerful (but also easier to design and manufacture).
 - However, great with thread-level parallelism.
- SMT
 - Can have one large and fast superscalar core.
 - Great performance on a single thread.
 - Mostly still only exploits instruction-level parallelism.

The memory hierarchy

- If simultaneous multithreading only :
 - all caches shared.
- Multi-core chips :
 - L1 caches private.
 - L2 caches private in some architectures and shared in others.
- Memory is always shared

Cache

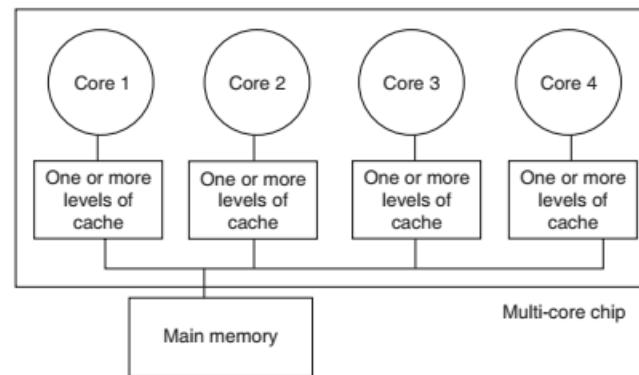
Cache is simply a high speed Static RAM (SRAM). Every processor consists of a cache which is useful in retrieving the data which will be used frequently. The cache is very fast in retrieving the frequently used data when compared to a Dynamic RAM (main memory).

Cache coherence

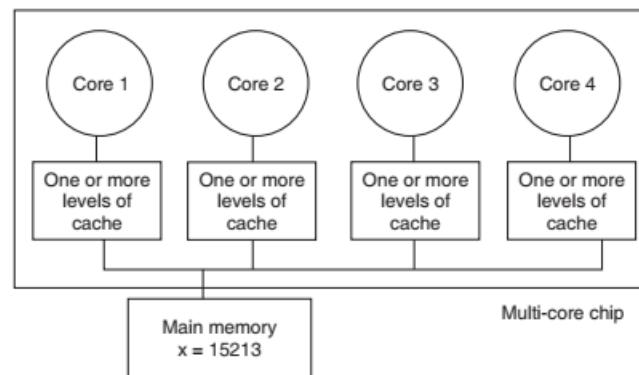
Cache Coherence is protocol which makes the caches of different processors work correctly and efficiently i.e. : the caches may contain an inconsistent data in which one cache may contain one data and the other one may have a different data which causes the data inconsistency problem.

The cache coherence problem

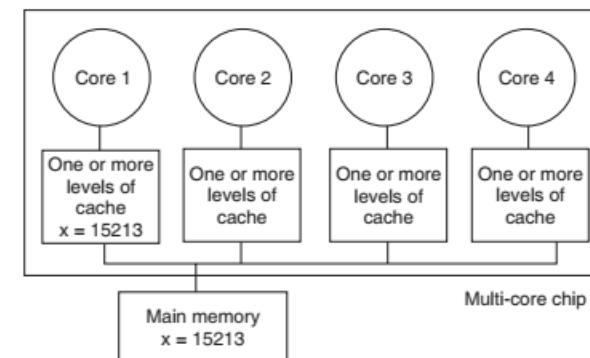
- Since we have private caches: How to keep the data consistent across caches ?
- Each core should perceive the memory as a monolithic array, shared by all the cores

**The cache coherence problem**

Suppose variable X initially contains 15213

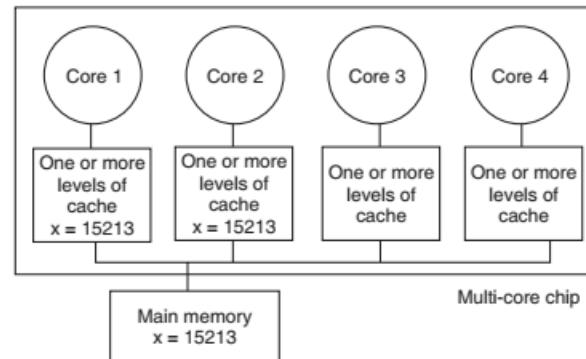
**The cache coherence problem**

Core 1 reads X

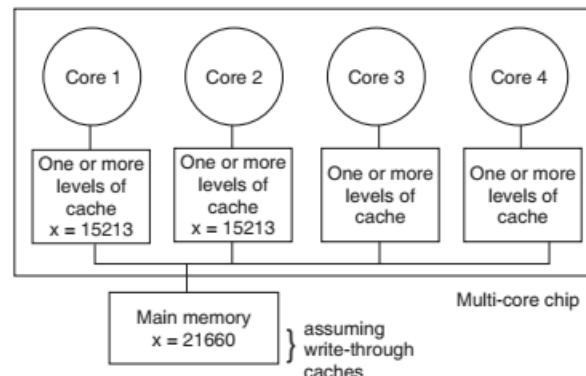


The cache coherence problem

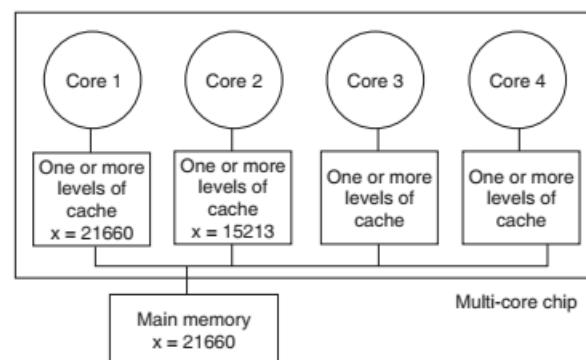
Core 2 reads X

**The cache coherence problem**

Core 1 writes to X, setting it to 21660

**The cache coherence problem**

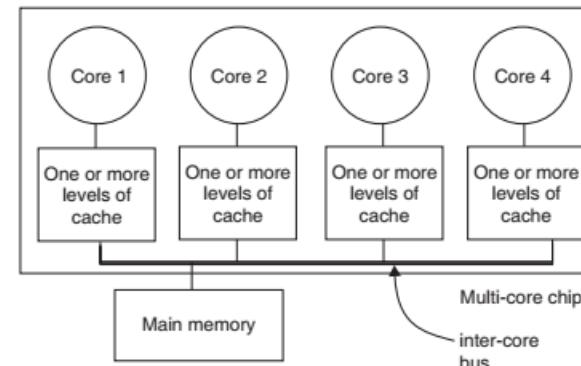
Core 2 attempts to read X... gets a stale copy



Solutions for cache coherence

- This is a general problem with multiprocessors, not limited just to multi-core
- There exist many solution algorithms, coherence protocols, etc.
- A simple solution:
invalidation-based protocol with snooping

Inter-core bus

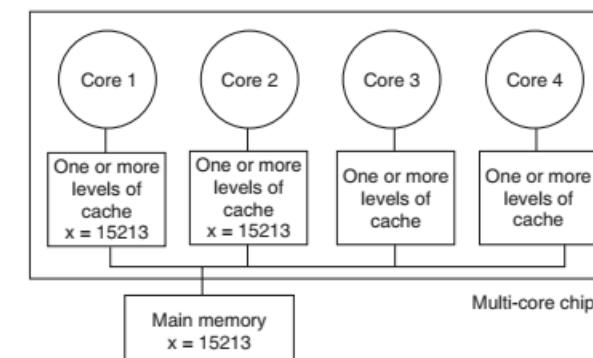


Invalidation protocol with snooping

- Invalidation:
If a core writes to a data item, all other copies of this data item in other caches are invalidated
- Snooping :
All cores continuously “snoop” (monitor) the bus connecting the cores.

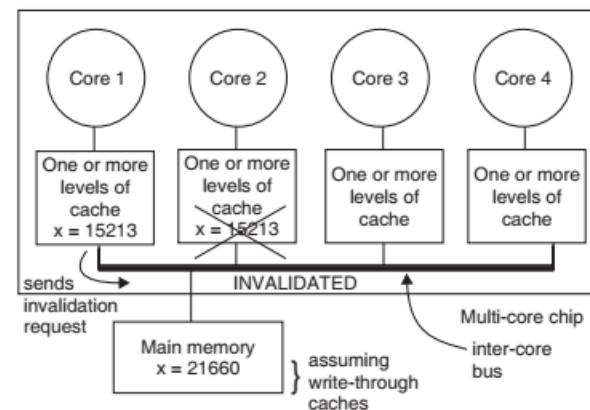
The cache coherence problem

Revisited : Cores 1 and 2 have both read X



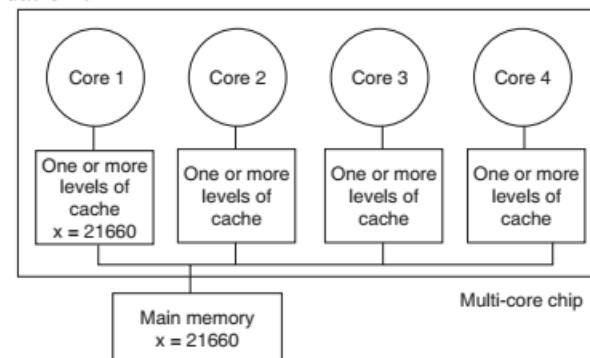
The cache coherence problem

Core 1 writes to x, setting it to 21660



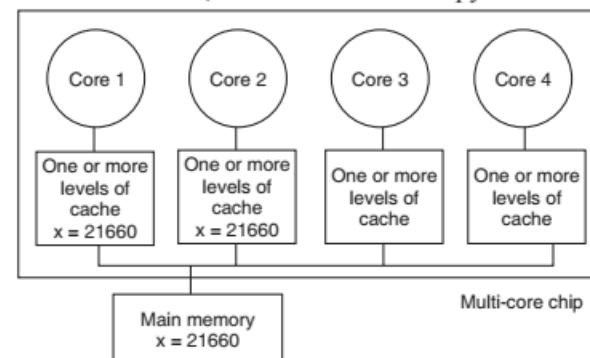
The cache coherence problem

After invalidation :



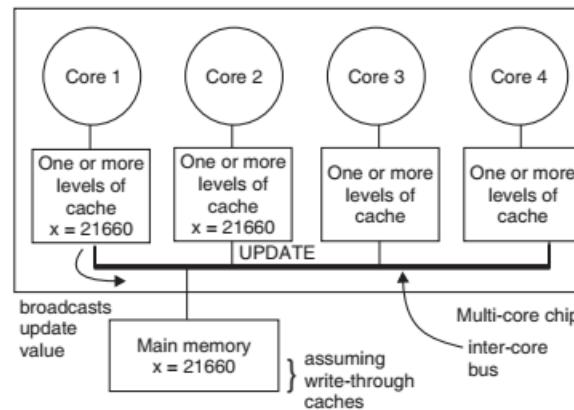
The cache coherence problem

Core 2 reads x . Cache misses, and loads the new copy.



Alternative to invalidate protocol : update protocol

Core 1 writes $x = 21660$



Invalidation vs update

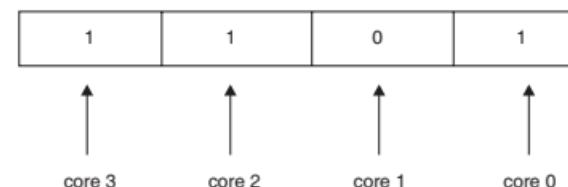
- multiple writes to the same location
 - invalidation : only the first time
 - update : must broadcast each write (which includes new variable value)
- Invalidation generally performs better : it generates less bus traffic

For effective use of multi core

- Programming for multi-core
 - Programmers must use threads or processes.
 - Spread the workload across multiple cores.
 - Write parallel algorithms.
 - OS will map threads/processes to cores.
- Assigning threads to the cores.
 - Each thread/process has an affinity mask.
 - Affinity mask specifies what cores the thread is allowed to run on.
 - Different threads can have different masks.
 - Affinities are inherited across fork () .

Affinity masks are bit vectors

- Example: 4-way multi-core, without SMT



- Process/thread is allowed to run cores 0, 2, 3, but not on core 1.

Difficulties for multi core

- Process migration is costly

- Need to restart the execution pipeline.
 - Cached data is invalidated.
 - OS scheduler tries to avoid migration as much as possible : it tends to keeps a thread on the same core.
 - This is called soft affinity.

Legal licensing issues

 - Will software vendors charge a separate license per each core or only a single license per chip?
 - Microsoft, Red Hat Linux, Suse Linux will license their OS per chip, not core

Conclusion

- Multi-core chips an important new trend in computer architecture.
 - Several new multi-core chips in design phases.
 - Parallel programming techniques likely to gain importance.

SUMMARY

In this chapter, we have learnt the causes of arbitration and their possible solutions. Also we have discussed about cache addressing models. Various cache performance issues have been explained. How an interleaved memory can be designed has also been explained thoroughly in this chapter. Cache coherence problem has been dealt.

MULTIPLE CHOICE QUESTIONS (MCQS) WITH ANSWERS

1. The effective bandwidth of each PE is :
 - (a) Directly proportional to number of PEs fighting for the bus.
 - (b) Inversely proportional to number of PEs fighting for the bus.
 - (c) Equal to number of PEs fighting for the bus.
 - (d) None of the above.
 2. Memory boards are :
 - (a) Passive boards
 - (b) Active boards
 - (c) Score boards
 - (d) None of the above
 3. The central arbitration scheme is also known as :
 - (a) Daisy-chain
 - (b) Independent Requests and Grants.
 - (c) Distributed arbitration
 - (d) None of the above
 4. Block in caches are called as :
 - (a) Blocks
 - (b) Blocks frames
 - (c) Frames
 - (d) Pages
 5. The memory requests are destined for :
 - (a) Sectors
 - (b) Blocks
 - (c) Frames
 - (d) Segments
 6. When cache size increases, hit ratio :
 - (a) Increases
 - (b) Decreases
 - (c) Remains constant
 - (d) None of the above.
 7. When block size equals the entire cache size, the hit ratio becomes :
 - (a) 0
 - (b) 1
 - (c) 2
 - (d) 4

:: ANSWERS ::

1. (b)	2. (a)	3. (a)	4. (b)	5. (b)
6. (a)	7. (a)	8. (b)	9. (b)	10. (b)

CONCEPTUAL SHORT QUESTION WITH ANSWERS

1. A block set associative cache memory consists of 128 blocks divided into four block sets. The main memory consists of 16,384 blocks and each block contains 256 eight-bit words.

 - (i) How many bits are required for addressing the main memory ?
 - (ii) How many bits are needed to represents the TAG, SET, WORD fields ?

Ans. (i) Main memory consists of 16,384 blocks.
Each block consist of 256 eight-bit words.

$$\begin{aligned}
 & \text{of main memory} \\
 & = 16,384 \times 256 \times 8 \\
 & = 4096 k \times 8 \\
 & = 2^{22} \times 8
 \end{aligned}$$

∴ 22 bits are required for addressing main memory.

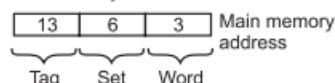
(ii) Associative mapping is being used here.

Now, Word bits required are as follows:

∴ To identify each word we must have ($2^3 = 8$) three bits reserved for it. Next, let us find out the **set bits** required. Now, there are 64 sets (i.e., $128/2 = 64$). To identify each set ($2^6 = 64$) six bits are required.

Now, how many tag bits are required?

The remaining 13 (22-6-3) address bits are tag bits which stores higher address of the main memory. So, the main memory address is formed:



2. What are different cache addressing models ?

Ans. There are 3 cache addressing models:

- (i) Physical Address Caches.
 - (ii) Virtual Address Caches.

When a cache is accessed with a physical memory address, it is called as a **physical address cache**. A unified cache accessed by the physical address is shown in Fig. 1.

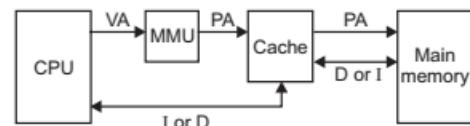


Fig. 1. Physical address caches.

Where

VA = Virtual address

PA = Physical address

I = Instructions

D = Data

So, as shown in Fig. 1 above, the cache is indexed and tagged with a physical address. A **cache hit** occurs when required data/instruction is found in the cache. Else a **cache miss** occurs. After a miss, some data comes to cache from the main memory.

On the other hand, when a cache is indexed or tagged with a virtual address then it is called as a **virtual address cache**. Herein, both cache and MMU translation are done in parallel. It is shown in Fig. 2.

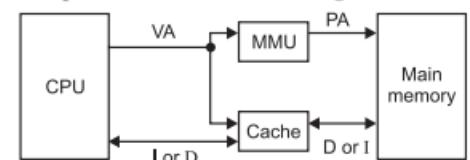


Fig. 2. Virtual address cache.

Remember that CPU generates virtual address (VA) or logical address. This VA is transformed into physical address (or PA) by MMU (as discussed earlier also)

3. What is the Aliasing problem ?

Ans. When different logically addressed data have the same index/tag in the cache, it results in the **aliasing problem**. This problem is found in virtual address cache scheme. Now, multiple processes may use the same range of virtual addresses. This aliasing problem may create confusion if **two or more processes access the same physical cache location**. The solution to this problem is to flush the entire cache whenever aliasing occurs. But please remember that excessive **flushing** may reduce cache performance, reduce the hit ratio and may waste lot of time.

4. A backplane bus multiprocessor has the following design specifications :

1. Bus clock rate = 20 MHz.
2. Word length = 64 bits.
3. Memory access time = 100 ns.
4. Shared address space = 2^{40} words.
5. Maximum no. of signal lines = 96.
6. Synchronous timing protol.
7. Neglect buffer and propagation delays.

Find out :

- (a) Maximum bus bandwidth.
- (b) Effective bus bandwidth.
- (c) Arbitration scheme.
- (d) Name and functionality of each signal line.
- (e) Number of slots required on the backplane ?

Ans. (a) Maximum bus bandwidth = $8 \times 20 = 160$ Mbytes/s.

(b) Total time taken by PE to access one word from the memory is :

$$\begin{aligned} &= 50\text{ns} + 100\text{ns} + 50\text{ns} \\ &= 200\text{ns} \end{aligned}$$

During this time, the bus cannot be used by other processors.

$$\therefore \text{The effective bandwidth} = \frac{8 \text{ bytes}}{200 \text{ ns}}$$

$$= 40 \text{ Mbytes/s}$$

If the memory addresses are interleaved then four words can be performed simultaneously. It takes 50 ns to transmit the address to the memory module, 100ns to get data ready in latches. Then it takes four bus cycles to transfer the four words to the requesting processor.

$$\begin{aligned} \text{Total time required} &= 50 + 100 + 4 \times 50 \\ &= 350 \text{ ns} \end{aligned}$$

$$\begin{aligned} \text{Effective bus bandwidth} &= \frac{4 \times 8 \text{ bytes}}{350 \text{ ns}} \\ &= 91.4 \text{ Mbytes/s.} \end{aligned}$$

- (c) Based on the desired performance and circuit complexity, any of the arbitration scheme can be used.
- (d) 40 address lines are needed.
64 data lines are needed.
- (e) At least 21 slots are needed, one for each processor board, one for each memory board and one for the bus controller.

5. Explain the following terms :

- (a) Write-through versus write-back caches.
- (b) Cacheable versus non-cacheable data.
- (c) Private caches versus shared caches.
- (d) Cache flushing policies.
- (e) Factors affecting cache hit ratios.

[GGSIPU, M.Tech. Dec. 2003, May 2003]

Ans. (a) In **write-through cache**, an update to a cache block causes the corresponding memory block to be updated immediately. There is no data loss. In **write-back cache**, the update to the memory block is postponed until the cache block is replaced. But this system has some disadvantages too. The write-back cache controller is not simple. Also, in case of a power failure, the data in the cache memory is lost.

- (b) Data which are globally shared among several processors and whose values may be updated can be tagged as **non-cacheable**. While instructions, private data and globally shared readable data are called as **cacheable**.
- (c) Private caches are those that are attached to individual processors. **Shared caches** are shared among processors, much like shared memory modules. These two types of caches can coexist in a system. For instance, a shared cache can be used as second-level cache in a multilevel cache system.
- (d) Cache flushing deals with the **aliasing problem** in a virtual address cache. These policies determine when **flushing** should be performed and the level at which flushing takes place *i.e.*, page level, segment level or the context level. These policies are related to the OS design also.
- (e) Cache hit ratio is affected by the factors like cache capacity and block size. A large cache will improve the hit ratio. For a fixed cache size, there is an optimal block size at which hit ratio peaks. A small block size does not take full advantage of locality of reference whereas a large block size can load unneeded data into the cache. Other factors include:
 - (a) The number of sets
 - (b) The set size

These factors affect the hit-ratio particularly in set-associative cache policy.

6. Explain the following terms :
 - (a) Low-order memory interleaving.
 - (b) Physical versus virtual address cache.
 - (c) Atomic versus nonatomic memory accesses.
 - (d) Memory bandwidth and fault tolerance.

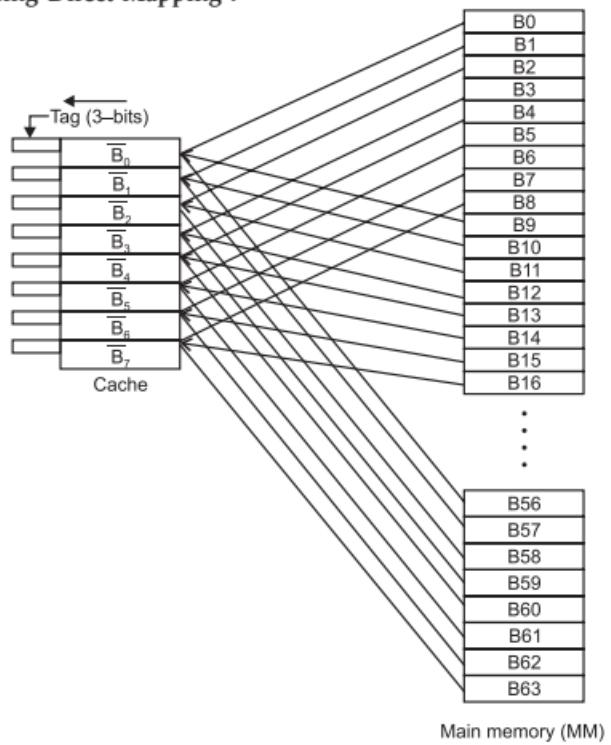
[GGSIPU, M.Tech. Dec. 2003]

- Ans. (a) Please refer to section 6.6 of this chapter.
(b) Please refer to Q.2 of this chapter's conceptual problems.
(c) In a shared memory, if the update to a memory location is observed by all PEs at the same time then the memory access is **atomic**. If the update is not necessarily observed by all PEs simultaneously then the memory access is **non-atomic**.
(e) Please refer to section 6.6 of this chapter.
7. The main memory of the computers has 64 blocks with a block size of 8 words. The cache has 8 block frames. Show the mappings from the blocks in MM to the block-frames in the cache. Draw all lines showing the mappings as clearly as possible for :
 - (a) Direct mapping alongwith address bits.
 - (b) Full associative mapping alongwith address bits.
 - (c) Two-way set associative mapping alongwith its address bits.
 - (d) Sector mapping with 4 blocks/sectors alongwith the address bits.

[GGSIPU, M-Tech. 2nd minor test-2004]

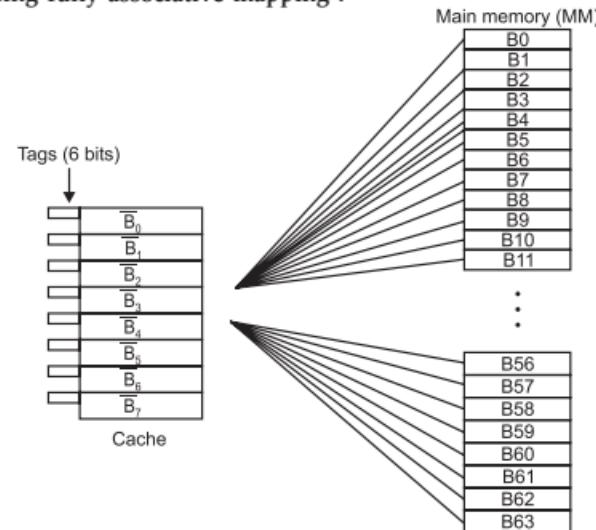
- Ans. The main memory blocks are numbered from 0 to 63. The cache block-frames are numbered from 0 to 7. So, we draw now one by one.

(a) Using Direct Mapping :



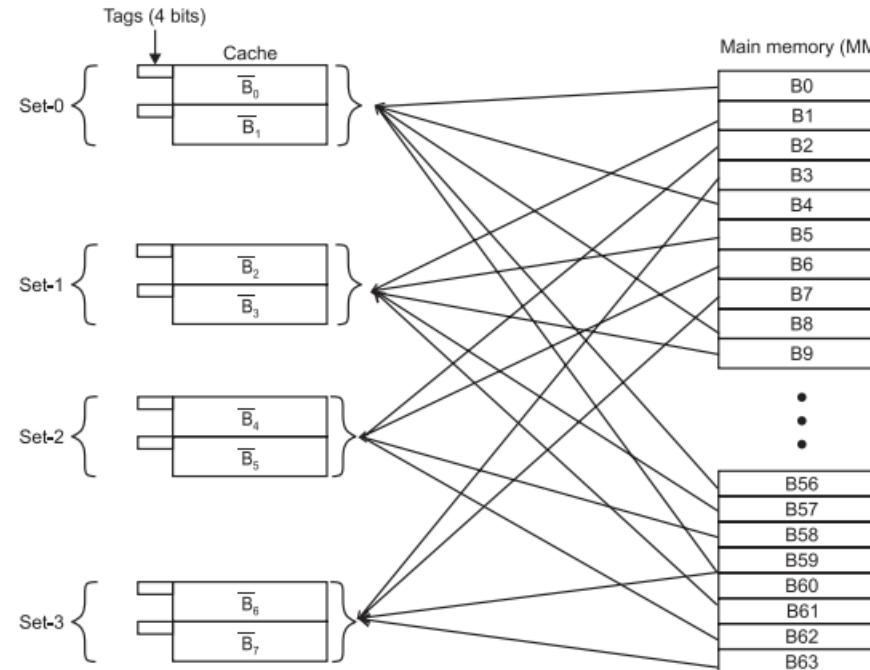
So, here cache address tag = 3 bits long,
 Block address = 3 bits long and word address = 3 bits long.

(b) Using fully associative mapping :



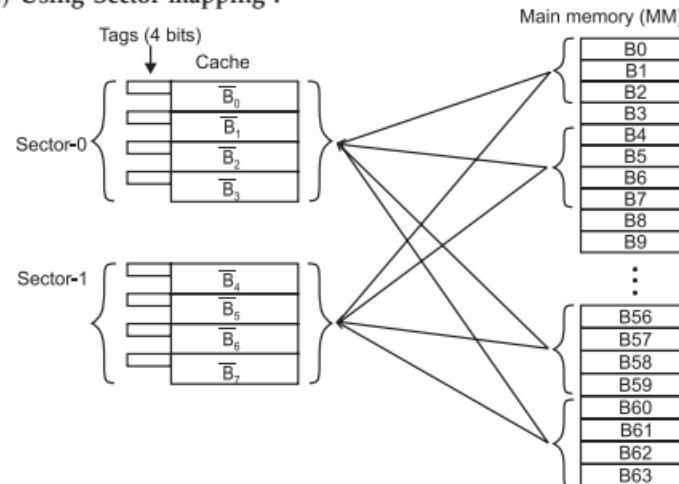
So, In cache address tag = 6 bits
And word address = 3 bits.

(c) Using set associative mapping :



Herein, cache address tag = 4 bits long.
Set number = 2 bits
And word address = 3 bits.

(d) Using Sector mapping :



Herein, sector number tag = 4 bits long.

Block address = 2 bits.

Word address = 3 bits.

8. Consider a cache (M_1) and memory (M_2) hierarchy with the following characteristics :

M_1 : 16k words, 50 ns access time.

M_2 : 1M words, 400 ns access time.

Assume 8-word cache blocks and a set size of 256 words with set-associative mapping.

(a) Show the mapping between M_2 and M_1 .

(b) Calculate the effective memory access time with a cache hit ratio of $h=0.95$?

[IPTU, B. Tech. (CSE) 8th Sem., 2003-04, 2004-05 & 2007-08]

& [GGSIPU, M.Tech-(CSE/IT) 1st Sem., Dec. 2011]

Ans. (a) Each set of the cache consists of

$$256/8 = 32 \text{ block frames.}$$

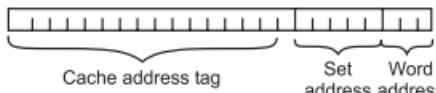
$$\text{Entire cache has } = 16 \times 1024/256$$

$$= 64 \text{ sets.}$$

$$\text{Similarly, memory contains } 1024 \times 1024/8$$

$$= 131072 \text{ blocks.}$$

Thus, the memory address format is drawn as follows :



A block B of the main memory is mapped to a block frame in set F of the cache if $F = B \bmod 64$.

(b) The effective memory access time for this memory hierarchy is given by :

$$= 50 \times 0.95 + 400 \times (1 - 0.95)$$

$$= 47.5 + 20$$

$$= 67.5 \text{ ns}$$

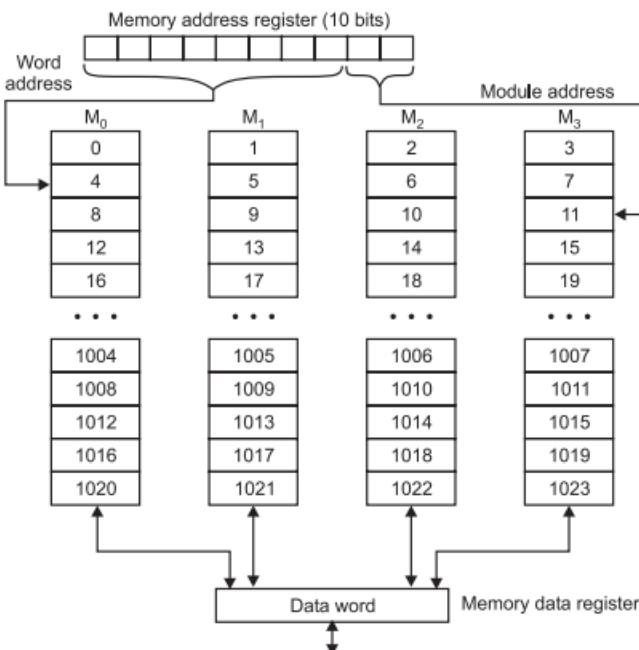
9. Consider a MM consisting of 4 memory modules with 256 words per module. Assume 16 words in each cache block. The cache has a total capacity of 256 words. Set associative mapping is used to assign cache blocks to block-frames. The cache has 4 sets.

(a) Show the address assignments for all 1024 words in a four-way low-order interleaved organization of the MM.

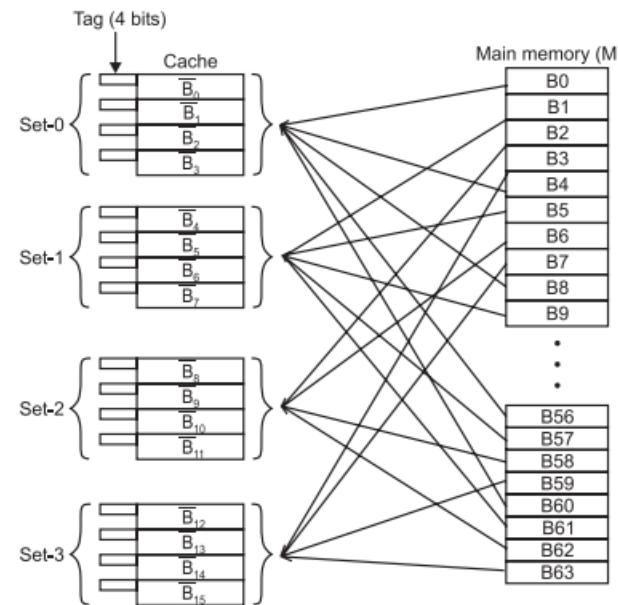
(b) How many 'blocks' are there in the MM? How many 'block-frames' are there in the cache?

- (c) Explain bit fields needed for addressing each word in the 2-level memory system.
- (d) Show the mapping from the blocks in main memory to the sets in the cache and explain how to use the tag field to locate a block frame within each set?
[GGSIPU, M. Tech. 6th sem ; 2004]

Ans. (a) The address assignment is shown below :



- (b) There are $1024/16 = 64$ blocks in the main memory
There are $256/16 = 16$ blocks-frames in the cache.
- (c) 10 bits are required to address each word in the main memory.
2-bits for selecting the memory module and 8 for the offset of a word within the module. 6-bits are required to select a word in the cache – 2 bits to select the set number and 4-bits to select a word within a block. Also that each block frame needs a 4-bit address tag to determine the block resident in it.
- (d) The mapping of memory blocks to the block frames in cache is shown below:



After the set in which a memory block can be mapped into is identified, the address tag of the block-frames in that set is compared by an associative search with the physical memory address to determine if the desired block is in the cache.

10. There are four cache memory organizations Answer the following questions with reasons :
- Rank the four cache organizations in terms of hardware complexity and cost.
 - Rank the four cache organizations with respect to block replacement algorithms.
 - Explain the effects of block mapping policies on the hit ratio issues.
 - Explain the effect of block size, set number and cache size on the performance of a set associative cache organization.

Ans. (a) Direct mapping has the lowest cost. It is because a simple mod operation is sufficient.

Fully associative mapping has highest cost. It is because an associative search on all block-frames is needed.

In set-associative mapping, the associative search is needed within each set.

In sector mapping, it is needed to find the sector. For a fixed cache size, the size of each set or sector will make a difference in the cost.

- In direct mapping**, the block replacement is rigid and trivial. Any replacement algorithms can be implemented with any of the three mappings of

cache. In case of fully associative mapping, the algorithms are applied to the entire cache. In set-associative or sector mapping, only a subset of the cache block frames are examined.

- (c) Effects of block mapping policies on hit-ratios is as follows :
1. **Direct mapping :** Hit ratio is high if the reference pattern leads to uniform distribution of the working set in the cache.
Hit ratio will drop if two or more blocks mapped to the same block-frame are referenced alternately.
 2. **Fully associative mapping :** Hit ratio is independent of the reference pattern.
 3. **Set-associative mapping :** On the average, the hit ratio should be higher than direct mapping and lower than fully associative mapping. Lesser thrashing than direct mapping is observed.
 4. **Sector mapping :** Hit ratio is sensitive to the reference pattern. When a block in a sector is replaced, the other blocks in the same sector are invalidated. This reduces the count of the valid blocks residing in the cache. This will have an adverse effect on the hit ratio.

(d) The effect of block size on the cycle count has been explained in section 6.5 of this chapter.

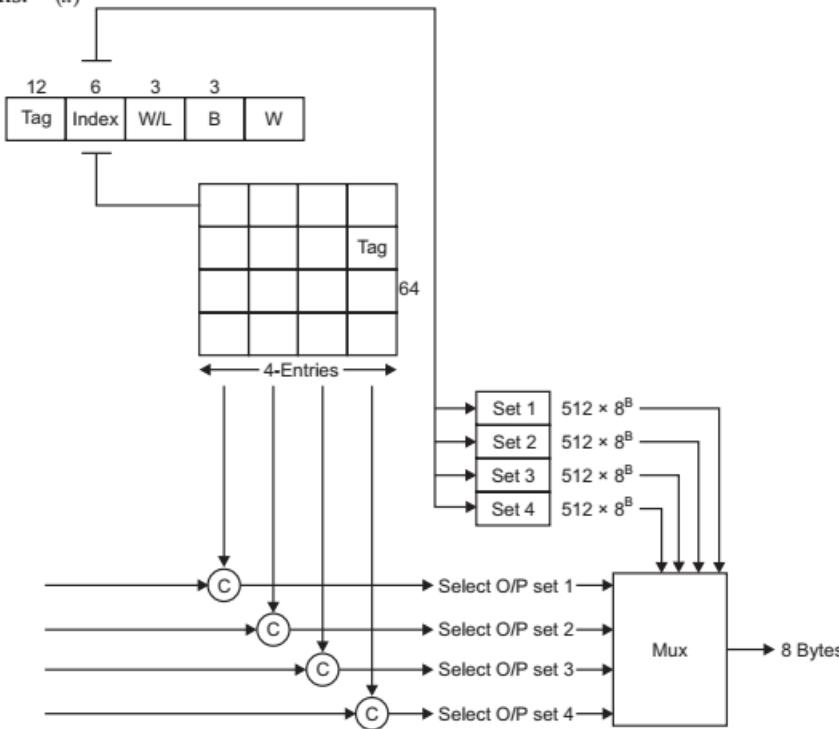
The effect of set number and associativity is here. For a fixed cache size, the two parameters are inversely proportional to each other. When the cache number is small it behaves more like a fully associative cache. When the number of sets is large, its behaviour is close to that of direct mapping. The hit ratio is expected to become lower now.

Cache size effect is also given next. With a larger cache both the hit ratio and the cycle count will increase. This is so because now more data (D) and instructions (I) can be stored in the cache memory.

11. A 128 KB Cache has 64 B lines, 8B physical word, 4k byte pages and is from way set-associative. It uses copy back and LRU replacement. The processor creates 30-bits (byte-addressed) virtual addresses that are translated into 24-bit (byte-addressed) real byte addresses (labelled- $A_{0_1} \dots A_{23}$) from least to most significant.

- (a) Show a complete layout of cache.
- (b) Which address bits are unaffected by translation?
- (c) Which address bits are compared to entries in cache directories?
- (d) Which address bit are used to address the cache directories?
- (e) Which address bits are appended to address bits in (d) to address the cache array?

Ans. (a)



(b) Four (4) address bits.

(c) W/L address bits,

(d) B/W and Index

(e) Index & W/L

12. Suppose two processors (in a Multiprocessor System) make a total of exactly two references to memory, every memory cycle ($T_c = 100\text{ms}$). The memory consists of 8 low-order interleaved memory modules. Find:

(a) Expected waiting time

(b) Total Access Time.

(c) Mean total number of queued requests.

(d) Offered memory bandwidth (references/sec)

(e) Achieved memory bandwidth. [MDU, BE(CSE)-7th Sem., May 2009]

Ans. (a)

$$\begin{aligned} T_w &= \frac{1}{\lambda} \left[\frac{p^2 - p^p}{2(1-p)} \right] \\ &= \frac{1}{\lambda} \left(\frac{1}{\mu} \right) \left[\frac{p - p}{2(1-p)} \right] \\ &= \frac{1}{\lambda} \left[\frac{p - p}{2(1-p)} \right] = \left[\frac{p - p}{2(1-p)} \right] \times T_s \end{aligned}$$

(b)

$$T = T_s + T_w$$

Where T_w - Waiting time T_s - Average Service function.

(c) $N = \lambda T$; items in a queue $d & e$.

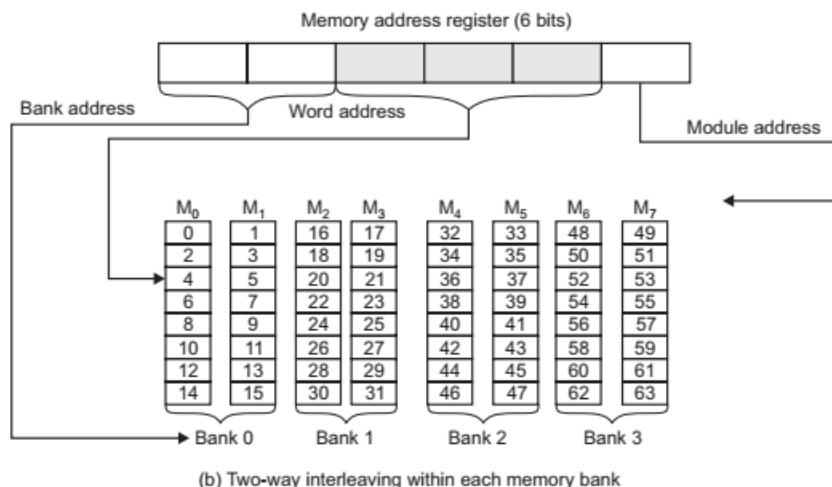
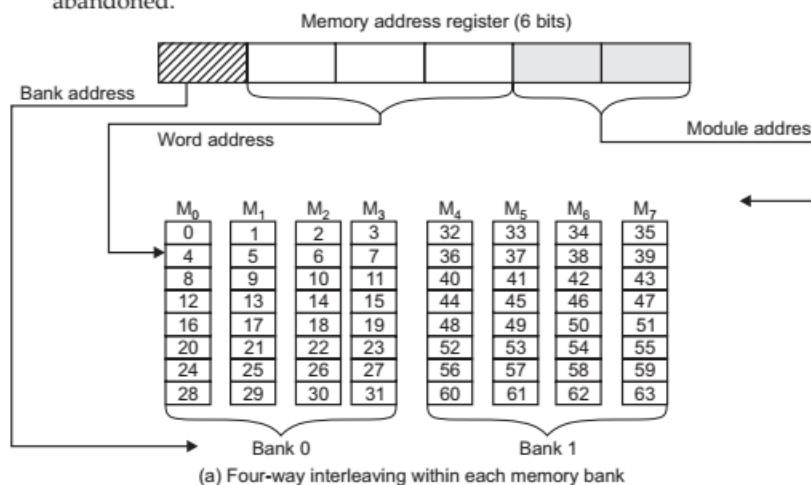
$$B(m, n) = K \left[1 - \left(1 - \frac{1}{K} \right)^t \right]$$

Where $k = \max(m, n)$
and $l = \min(m, n)$.

13. "Higher the degree of interleaving, the higher is the potential memory bandwidth for fault free system." Discuss.

Ans. Two alternative memory addressing schemes showing combination of high and low order interleaving is shown in Figures (a) & (b) Fig. (a) shows four way low order interleaving organized in two memory banks whereas two way low order interleaving organization is divided into four memory banks.

In case of single module failure, the maximum memory bandwidth of the eight way interleaved memory is reduced to zero because the entire memory bank design must be abandoned. In figure (b), the maximum Bandwidth is reduced to four words per memory cycle because only one of the two faulty banks is abandoned.



EXERCISE QUESTIONS

1. Compare the relative merits of the three cache memory organizations :
 - (a) Fully associative cache.
 - (b) Set associative cache.
 - (c) Direct mapping cache.

[UPTU, B. Tech. (CSE) 8th Sem., 2004-05 & GGSIPU, B.Tech. 7th sem; Dec. 2007]
 2. Discuss arbitration, transaction and interrupt w.r.t. backplane bus system :

[GGSIPU, B. Tech. 7th sem; Dec. 2007]
 3. Discuss the cache coherence problem and its solutions.

[Pune Univ., B.Tech. 2nd sem; May 2004]
 4. Consider a cache consisting of 128 blocks of 16 words each for a total of 1024 (1k) words. Assume that the main memory is addressable by a 16-bit address and it consists of 4k blocks. How many bits are there in each Tag, Block/set and word fields for different mapping techniques.
 5. A block set associative cache consists of a table of 64 blocks divided into four block sets. The main memory contains 4096 blocks, each consisting of 128 words.
 - (i) How many memory bits are there in a main memory address?
 - (ii) How many bits are there in each of the Tag, set and word bits.
 - (iii) What is the size of the cache memory?
 6. What is cache? Describe any two methods of cache mapping to main memory

[UPTU, B. Tech. (CSE) 8th Sem., 2005-06]
 7. What is m-way inter leaving? Discuss different types of memory interleaving.

[UPTU, B. Tech. (CSE) 8th Sem., 2006-07]
 8. Describe how cache coherence is maintained in Distributed Shared memory parallel computers

[GGSIPU, B.Tech. 7th sem; Dec. 2011]
 9. Describe associative caches. How they function ?

[GGSIPU, B.Tech. 7th sem; Dec. 2011]

□ □ □

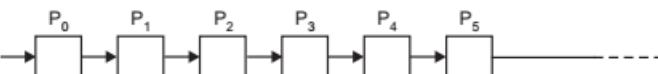
CHAPTER

7

PIPELINING

7.0 INTRODUCTION

Pipelining means realizing temporal parallelism in an economical way. In this technique, the problem is divided into a series of tasks that have to be completed one after the other. Each task is executed by a separate process or processor.



The problem is divided into separate functions that must be performed in succession. Pipelining is very much similar to assembly line in an industry. It means an overlapped parallelism.

7.1 PIPELINE-PRINCIPLES AND IMPLEMENTATION

A linear pipeline processor is a cascade of processing stages which are linearly connected to perform a fixed function over a stream of data flowing from one end to the other.

Linear pipeline is of two types :

- (a) Asynchronous pipelines.
- (b) Synchronous pipelines.

Pipelining involves some conditions. These conditions are :

1. The input task can be divided into sequence of subtasks.
2. Each subtask can be executed by a specialized hardware that operates concurrently with other pipeline stages.
3. There is a constant stream of tasks into the pipe and there is overlapped execution at the subtask level.

7.1.1 Linear Pipeline Processor

7.1.1.1 Asynchronous Model

In an asynchronous pipeline, the data flow between the adjacent stages is controlled by a handshaking protocol.

Principle : When the stage, S_i , is ready to transmit, it sends a ready signal to stage S_{i+1} . After stage, S_{i+1} , receives the incoming data, it returns an acknowledgement signal to S_i .

An asynchronous pipeline model is shown in Fig. 7.1 below.

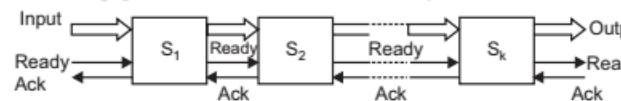


Fig. 7.1. Asynchronous pipeline model.

Applications of Asynchronous pipelines : Asynchronous pipelines are useful in designing the communication channels in the message – passing multicomputers where pipeline worm hole routing is practised.

Please note that these pipelines may have a variable throughput rate. Different amounts of delay may be experienced in different stages.

7.1.1.2 Synchronous Model

A synchronous pipeline is shown in Fig. 7.2

Basic structure of linear-pipelining is as follows :

L : Latch (ICs)

CLK : Clock pulse

S_i : the i^{th} stage

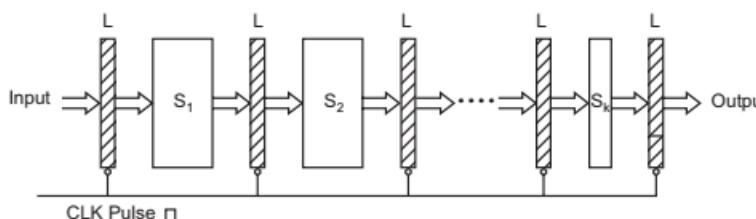


Fig. 7.2. Synchronous pipeline.

Working : The pipeline consists of a cascade of processing stages. The stages are pure combinational circuits performing arithmetic or logic operations over the data stream flowing through the pipe. The stages are separated by high-speed interface latches. The latches are fast-registers for holding the intermediate results between the stages. Information-flows between adjacent stages are under the control of common CLK applied to all the latches simultaneously.

Space-time diagram : One can draw a “space-time diagram” (Fig. 7.3) to show overlapped operations in a linear pipeline processor, having 4-stages only. (We can extend it to n -stages)

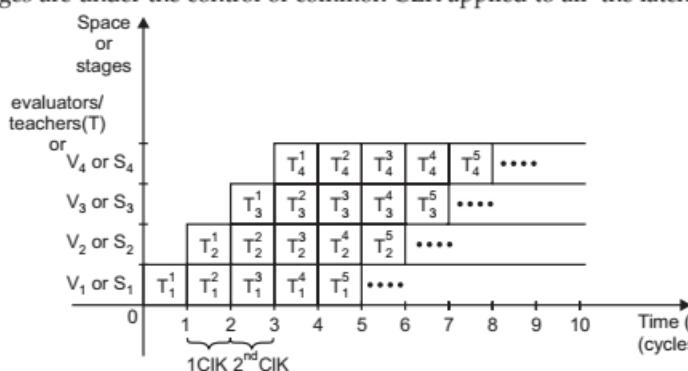


Fig. 7.3. Space-time diagram.

Where, $T_j^i \rightarrow$ is j^{th} -subtask } $T_1^1 \rightarrow$ paper -1, Q 1
 in i^{th} -task. } $T_2^1 \rightarrow$ paper -1, Q 2

Explanation : Once the pipe is filled up, it will output one result / clock period, independent of the number of stages in the pipe.

Ideally, a linear pipeline with k -stages can process n -tasks in $T_k = k + (n - 1)$ clock periods.

Where, k -cycles are used to fill up the pipeline (*i.e.*, no. of subtasks say, k -questions of answer-sheet)

Whereas $T_1 = nk$ when pipelining is not done.

The following are the terms associated with linear pipeline-structure of Parallel computer.

7.1.2 Clock-period (τ) : It is defined by, the formula given below -

$$\begin{aligned} \tau &= \max \{ \tau_i \}_{i=1}^k + \tau_l && i \leftarrow 1 \text{ to } k \\ \therefore \tau &= \tau_m + \tau_l \\ \text{where } \tau_i &\rightarrow \text{is time-delay for stage-}i \\ \text{and } \tau_l &\rightarrow \text{is time-delay of each interface latch.} \end{aligned}$$

and that the reciprocal of CLK-period, (τ), is called the frequency, f , of a pipeline processor *i.e.*,

$$f = \frac{1}{\tau} \text{ Hz.}$$

7.1.3 Speed-up (S_k)

We define speed-up of a k -stage linear-pipeline processor over an equivalent nonpipeline processor as :

$$\begin{array}{c} \text{nonpipelined or Neumann's} \\ \boxed{S_k = \frac{T_1}{T_k} = \frac{n.k}{k + (n - 1)}} \\ \text{pipelined} \end{array}$$

\therefore maximum speed-up is, $S_k \rightarrow k$
 where $n \gg k$

i.e. minimum speed-up that a linear pipeline can provide is k , where k is the number of stages in the pipe. This maximum speed-up is never fully achievable because of "data-dependencies" between instructions, interrupts, program-branches etc. Many pipeline cycles may be wasted on a waiting state caused by out-of-sequence instruction executions.

7.1.4 Efficiency (η)

Efficiency of a linear pipeline is "measured by the percentage of busy time-space spans over the total time-space span".

Now, time-space span = time-interval \times stage space

& Total time-space span = $[k\tau + (n - 1)\tau] \times [k]$

i.e., k -stages are our stage-space

... (1)

Where $n \rightarrow$ number of tasks (instructions)

$k \rightarrow$ number of pipeline stages

$\tau \rightarrow$ CLK period of a linear pipeline.

& Busy time space span = $(n\tau) k$ [from 4th stage all processors are busy] ...(2)

\therefore from (1) & (2) :

$$\eta = \frac{nk\tau}{(k\tau + (n-1)\tau)k}$$

\therefore

$$\eta = \frac{n}{(k\tau + (n-1)\tau)k} = \frac{n}{(k^2\tau + nk\tau - k\tau)/k\tau}$$

or

$$\eta = \frac{n}{k + (n-1)}$$

Note that $\eta \rightarrow 1$ as $n \rightarrow \infty$ $\left[\because \frac{n}{k + (n-1)} = \frac{1}{\frac{k}{n} + \left(1 - \frac{1}{n}\right)} = \frac{1}{\frac{k}{\infty} + \left(1 - \frac{1}{\infty}\right)} = \frac{1}{0+1} = \frac{1}{1} = 1 \right]$

i.e., larger the number of tasks (n) flowing through pipeline, the better is its efficiency.

Also we realize that $\eta = \frac{S_k}{k}$ or $\eta = \frac{S_k}{k} \times 100\%$

i.e., efficiency of a linear pipeline is also defined as the ratio of its actual speed-up to the ideal speed-up k . Since, it is a ratio of two speed ups so it doesn't have any units. However, it is expressed in %.

7.1.5 Throughput (w)

The number of results (tasks) that can be completed by a pipeline per unit time is called its throughput. This rate reflects the computing power of a pipeline. i.e.,

$$\therefore w = \frac{n}{k\tau + (n-1)\tau} = \frac{\eta}{\tau}$$

Where n equals the total number of tasks being processed during an observation period of $k\tau + (n-1)\tau$.

When $\eta \rightarrow 1$, w = throughput = $\frac{1}{\tau} = f$ (Hz).

This means that the maximum throughput of a linear pipeline is equal to its frequency, which corresponds to one output result per clock period.

7.2 NON-LINEAR PIPELINE PROCESSOR

When a pipeline allows feed-forward and feedback connections in addition to the streamline connections then it is called as a **non-linear pipeline**.

A **dynamic pipeline** is sometimes called as a **non-linear pipeline** because of feed forward

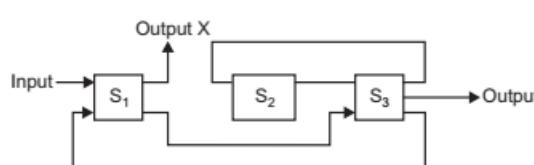


Fig. 7.4. Non-linear pipeline.

and feedback connection. It can be reconfigured to perform different functions at different times. Please note that the linear pipelines discussed earlier are **static pipelines** because they perform fixed functions. A two function non-linear pipeline is shown in Fig. 7.4.

This non-linear pipeline can perform two functions X and Y.

In practice, many of the arithmetic pipeline processes allow non-linear connections to implement **recursion** and **multiple functions**. If feedback and feed-forward connections are not used properly then the inherent advantage of pipelining may be lost. On the other hand, if non-linear data flow is properly sequenced then the pipeline efficiency will enhance.

7.3 CLASSIFICATION OF PIPELINE PROCESSORS

7.3.1 Based on levels of processing—Handler's classification (1977)

Handler has proposed the classification scheme according to the levels of processing as follows.

7.3.1.1 Arithmetic pipelining

In this method, the ALUs of a computer are segmented in various data formats for pipeline operations. This is shown in Fig. 7.5.

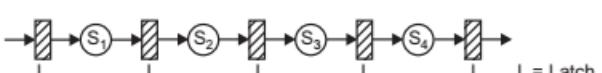


Fig. 7.5. An arithmetic pipeline.

For example : 4 – stage pipes in Star 100,

4 – stage pipes in TI-ASC,

14 – stage pipes in Cray-1,

26 – stage pipes in Cyber-305.

7.3.1.2 Instruction Pipelining

This type of pipelining is also known as **instruction look ahead**. In this the execution of stream of instructions is pipelined by overlapping the execution of current instruction with fetch, decode and operand fetch of subsequent instructions. Almost all high performance computers have instructions execution pipeline. An instructions pipeline is shown in Fig. 7.6 below.

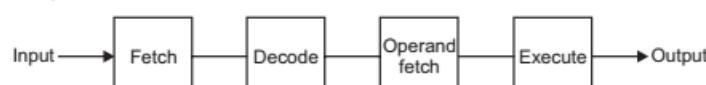


Fig. 7.6. Instruction pipeline.

7.3.1.3 Processor Pipelining

In this pipeline, same data stream is processed by the cascade of processors. Each processor performs a specific task. The data stream passes the first processor with results stored in a memory block which is also accessible by the second processor. The second processor processes this result and passes it to third and so on. This pipeline of multiple processor is not much popular. It is shown in Fig. 7.7 below.

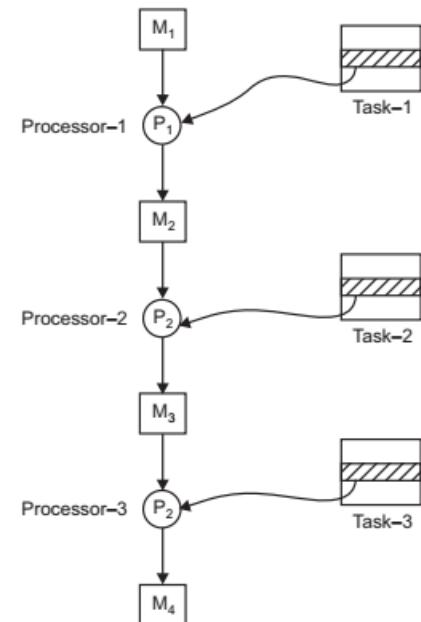


Fig. 7.7. Processor pipelining.

7.3.2 Based on Pipeline Configuration

Ramamoorthy and Li's classification (1977)

7.3.2.1 Unifunction versus Multifunction pipelines

Unifunction Pipeline : A pipeline with fixed and dedicated function is called as a unifunctional pipeline.

For example : Floating-point adder.

Please note that Cray-1 has 12 unifunctional pipeline units for various scalar, vector, fixed point and floating-point operations.

Multifunction Pipeline : A pipeline that performs different functions either at different times or at the same time, by inter connecting different subsets of stages in the pipeline is called as a multifunction pipeline.

For example : TI-ASC has 4 multifunction pipeline processors.

7.3.2.2 Static versus Dynamic pipelines

Static Pipeline : It is a pipeline that assumes only one functional configuration at a time. It can be either unifunctional or multifunctional. The function performed by static pipeline should not change frequently. The frequent change in the function will lead to degradation of performance of static pipeline.

Dynamic Pipeline : Herein, several functional configurations can exist simultaneously. Dynamic pipeline must be multifunctional and uni-functional pipeline must be static. The control in the dynamic pipe is complex as compared to the control in the static pipeline. Please note here that the unifunctional or multifunctional static pipelines are more popular in most of the existing computers.

7.3.2.3 Scalar versus vector pipelines

This classification is based on the instruction or data types.

Scalar pipeline : If a pipeline processes scalar operands under the control of Do-loop then it is called as a **scalar pipeline**. The instructions in a small Do-loop are normally prefetched into the instruction buffer. The scalar operands which are required for repeated scalar instructions are moved into a data cache. This ensures the continuous supply of operands to the pipeline. *For example : IBM-360*

Vector Pipeline : The pipelines that process vector instructions over vector operands are known as **vector pipelines**. Computers having vector instructions are called as vector-processors.

For example : STAR-100, Cyber-250, Cray-1.

Actually, vector pipelines are expanded version of the scalar pipelines. Please note that the scalar pipelines are under software control whereas the vector pipeline uses firmware and hardware controls for processing vector operands.

7.3.3 General Pipelines

In linear pipelines, the inputs and outputs are totally independent. In some cases, it is not so and inputs may depend on previous outputs. Thus, these pipelines will possess feedback or feed forward connections and flow of data is non-linear.

For example : Linear recurrence.

Thus, pipelines in general may have feedback and feed forward connections in addition to the cascade connections. A general pipeline is shown in Fig. 7.8 below.

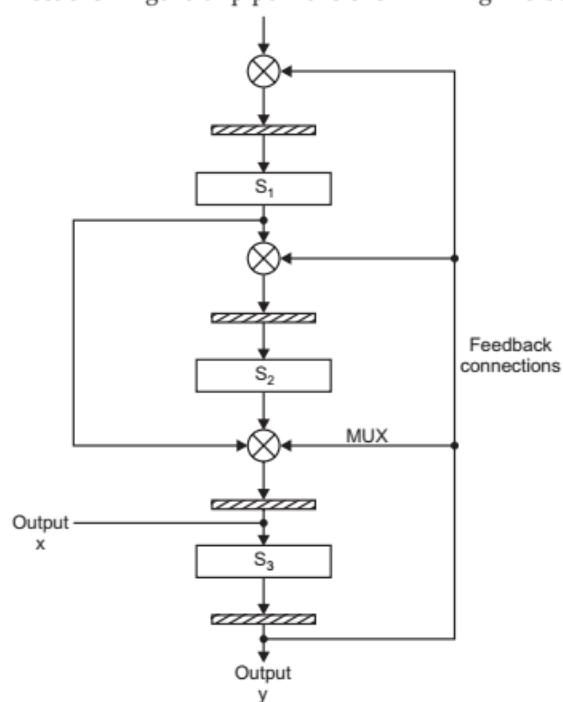


Fig. 7.8. A general pipeline.

As shown in Fig. 7.8, it is a two function pipeline with function-x and function-y. There are three stages S_1 , S_2 and S_3 . The crossed circles are multiplexers. They are used for selecting among multiple connection paths in evaluating different functions.

7.3.3.1 Reservation tables

Reservation table is a 2D chart. It is also known as a Gantt chart. The utilization pattern of successive stages in the pipeline is specified by a reservation table. It was introduced by Davidson in 1971. It shows the utilizations or reservation of successive stages in a pipeline for evaluation of a specific function, in successive pipeline cycles. It is very much similar to the space-time diagram. A reservation table for a linear pipeline is shown in Fig. 7.9 below.

	1	2	3	4	
Stages	S_1	X			
	S_2		X		
	S_3			X	
	S_4				X

Fig. 7.9. Reservation table.

It shows that stage-1 (S_1) is being used at time = 1 units, stage-2 (S_2) is busy at time=2 units and so on. A reservation table consists of rows corresponding to pipeline stages and columns to clock time units. The evaluation time for a function is equal to the total number of clock units in the reservation table. The table represents the flow of data through the pipeline for one complete evaluation of a given function. One symbol like 'X' is sufficient to mark a busy stage in a unifunction pipeline but multiple symbols are used to mark different functions like X and Y. For example,

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
S_1	X			X			X	
S_2		X						X
S_3			X	X	X			

Fig. 7.10. For Function X

	t_0	t_1	t_2	t_3	t_4	t_5	t_6
S_1	Y				Y		
S_2			Y			Y	
S_3		Y		Y			Y

Fig. 7.11. For Function Y.

Please note that the evaluation time of different functions may be different. Like in above Fig. 7.10, the evaluation time for X is 8 clock units and for Fig. 7.11, the evaluation time for Y is 7 clock units.

Also note that when any square of the reservation table is marked, it indicates that the corresponding stage is used at that time. If $(i, j)^{th}$ square is marked then it means that state, S_i , is used j -time units after the initiation of the function evaluation.

In a static, unifunction pipeline, one reservation table is sufficient to describe the data flow pattern. On the other hand, in multifunctional pipeline different reservation tables are used for different functions. Please note that a given reservation table need not correspond to one particular hardware pipeline only. It may be used by several hardware pipelines with different interconnection structures.

These reservation tables describe many interesting features of the pipeline utilization. Some of the features are as follows :

1. **Multiple marks in a Row :** It indicates a repeated usage of a given stage if the marks are in distinct columns. It indicates a prolonged usage of a stage if the marks are in adjacent columns.
2. **Multiple marks in a column :** It indicates that multiple stages are simultaneously used.
3. **Continuous marks in a row/column :** If the marks are continuous, they may represent a slower stage needing more time slots.
4. **Number of columns in a table :** The number of columns in the table stands for the total compute time required by the computation represented by the table.
5. A general pipeline may have multiple paths, parallel usages of multiple stages and non-linear flow of data.

7.3.3.2 Latency Analysis Terminologies

We need to understand some terminologies used during latency analysis. They are discussed below.

1. **Collision :** The number of time units (or clock cycles) between two initiations of a pipeline is the **latency** between them. **Latency values must be positive integers only.** A latency of k means that two initiations are separated by k clock cycles. Any attempt by two or more initiations to use the same pipeline stage at the same time will cause a **collision**.
2. **Forbidden Latencies :** A collision implies resource conflicts between two initiations in the pipeline. Therefore, all collisions must be avoided in scheduling a sequence of pipeline initiations. Some latencies will cause collisions and some will not. **Latencies that cause collisions are called as forbidden latencies. Please note that in order to detect a forbidden latency, one needs to check the distance between any two checkmarks in the same row of the reservation table.**

For example : In the reservation table given below, the latencies are as follows:

	1	2	3	4	5	6	7	8
S ₁	X					X		X
S ₂		X		X				
S ₃			X		X		X	

The distance between the first mark and the second mark in row S_1 above is 5 i.e., it's forbidden latency is 5.

Now, consider another 3 stage pipeline :

	1	2	3	4	5	6	7	8	9	10	11
S ₁	X ₁		X ₂		X ₃	X ₁	X ₄	X ₁ , X ₂		X ₂ , X ₃	
S ₂		X ₁		X ₁ , X ₂		X ₂ , X ₃		X ₃ , X ₄		X ₄	
S ₃			X ₁		X ₁ , X ₂		X ₁ , X ₂ , X ₃		X ₂ , X ₃ , X ₄		

....

Fig. 7.12. Collision with scheduling latency of 2.

We observe from this Fig. 7.12 that the initiations X_1 and X_2 collide in stage-2 at time = 4. At time = 7, these initiations collide in stage 3. Similarly, other collisions are shown in times 5, 6, 8, ... etc.

3. Latency sequence : It is defined as a sequence of permissible non forbidden latencies between successive task initiations.

For example : In Fig. 7.12, we observe that 2, 4, 5, 7 are forbidden latencies. So, 1, 3, 6, and 8 are some non-forbidden latencies.

4. Latency cycle : It is defined as a latency sequence which repeats the same subsequence or cycle indefinitely.

For example : Consider the following reservation table in Fig. 7.13

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
S ₁	X ₁	X ₂			X ₁	X ₂	X ₁	X ₂	X ₃	X ₄				X ₃	X ₄	X ₃	X ₄	X ₅	X ₆	
S ₂		X ₁	X ₂	X ₁	X ₂					X ₃	X ₄	X ₃	X ₄					X ₅	
S ₃			X ₁	X ₂	X ₁		X ₁	X ₂			X ₃	X ₄	X ₃		X ₃					

Fig. 7.13. Shows latency cycle (1, 8) = 1, 8, 1, 8, 1, 8....., with an average latency of 4.5.

Here in, the latency cycle (1, 8) represents the infinite latency sequence 1, 8, 1, 8, 1, 8This implies that the successive initiations of new tasks are separated by one cycle and eight cycles alternately.

5. Average Latency : The average latency of a latency cycle is obtained by dividing the sum of all latencies by the number of latencies along the cycle.

For example : The latency cycle (1,8) has an average latency of $(1+8)/2= 4.5$

6. Constant Cycle : A Constant cycle is a latency cycle which contains only one latency value.

For example :

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
S ₁	X ₁				X ₁	X ₂	X ₁			X ₂	X ₃	X ₂			X ₃	X ₄	X ₃			
S ₂		X ₁		X ₁			X ₂		X ₂			X ₃		X ₃			X ₄		
S ₃			X ₁		X ₁		X ₁	X ₂		X ₂		X ₃		X ₃	X ₃	X ₃		X ₃		

Here, Latency cycle (6) = 6, 6, 6, 6.....

So, (6) is a constant cycle. The average latency of constant cycle is simply the latency itself.

7.3.3.3 Collision-free scheduling-steps to follow

Davidson provides an elegant method for identifying collision-free issue latencies and determining the possible issue sequences that are collision free. These sequences are called as state transitions Davidson method involves four steps :

Step-1 : Compile a list of the issue latencies that will cause collisions for each row (stage) of the reservation table.

	1	2	3	4	5	6	7	8	9	0
1	X								X	
2		X	X					X		
3			X							
4				X	X					
5						X	X			

Fig. 7.14. A Reservation table.

For example : Consider the following reservation table shown in Fig. 7.14

Note here that row-1 will have a collision with an issue latency of 8. That, is we need to check the distances between any two checkmarks in the same row of the reservation table.

∴
 Row-1 : 8
 Row-2 : 1, 5, 6
 Row-3 : 0
 Row-4 : 1
 Row-5 : 1

Step-2 : Form the for bidden list for the reservation table that is the set union of all the non-null row forbidden issue latencies :

Forbidden list, $F = 1, 5, 6, 8$

The forbidden list tells us that new issues can be made without collisions at latencies of 2, 3, 4, 7, 9^+ (or at $t_3, t_4, t_5, t_8, t_{10}^+$). Note that the '+' means any latency greater than the one indicated.

Step-3 : A collision vector for the initial state is constructed from the forbidden list. This vector reads from right to left (by latency) using '1' for collision and '0' otherwise. Please note that the last non-forbidden latency of 9 can be ignored in the collision vector.

For example : In Fig. 7.14, the collision vector, V , that is formed is as follows :

Collision vector, $V = [1, 0, 1, 1, 0, 0, 0, 1]$

That, is since 1 is a forbidden state in list, F , so we put a '1' as '1' shows a collision. As 2 is not a forbidden state (from step-2) so we put a '0' under it in the above collision vector. V . This goes on. We ignore the last non-forbidden latency of 9 in this collision vector, V .

Step-4 : A state transition diagram is constructed from the collision vector.

The initial state collision vector is $V = [10110001]$. The transitions are 2, 3, 4, 7 and 9^+ . To find the collision vector of the next state, the collision vector of the present state is shifted right for each transition. The shifted collision vector is ORed with the initial state collision vector to form a new state. For example :

Now, $V = [10110001]$

Right-shifting V by 1-bit we get :

$$V' = [1011000]$$

Again, right-shifting V by 1-bit we get :

$$V'' = [101100].$$

We have right shifted 2 times because one of the out-transition is 2. So, now we OR this result (V'') by the original collision vector V and we get :

$$\begin{array}{r} 10110001 \\ + \quad 101100 \\ \hline \underline{10111101} \end{array}$$

So, we can now draw this in our state transition diagram. We repeat this process for 3, 4, 7 and 9^+ transitions also. When the out-transition is 3 we get V'' as [10110111].

When out-transition is 4, we get :

$$V'' = [10111011]$$

Similarly when transition is 7, we get :

$$V'' = V = [10110001]$$

Also, for out-transition of 9, we get :

$$V'' = V = [10110001]$$

So, we draw a state transition diagram (STD) now :

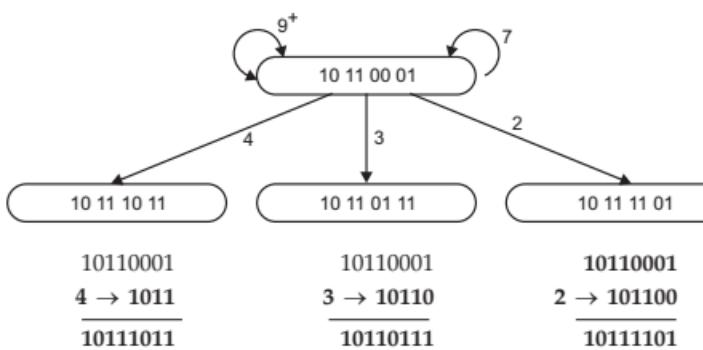


Fig. 7.15. An initial state diagram.

Observe here that there are three new states and two returns to the initial state. The latency used for the new state is shown by a number on the arrow connecting the states. Please note that the latency of 7 returns to the original state. Any latency equal to or greater than 9 will also return to the original state – an effect that is indicated by 9^+ . Please note that the process of shifting continues by taking the collision vector of each new state, shifting and ORing with the initial state collision vector. So, now we get a complete state transition diagram shown in Fig. 7.16 below :

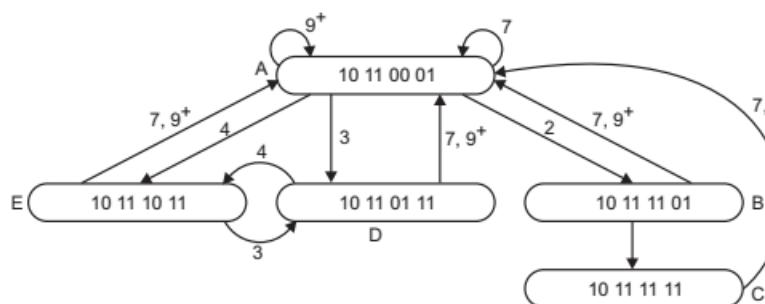


Fig. 7.16. Complete STD. Showing states – A, B, C, D, and E.

We can identify various issue cycles from Fig. 7.16 above. They are shown in table below :

Cycle	Path	Issue Latencies	Average Latency	Acceptance Ratio
1.	A, A, A, A	7,7	7	0.14
2.	A, D, A, D,	3, 7, 3, 7	5	0.2
3.	A, D, E, D, E	3, 4, 3, 4	3.5	0.28
4.	A, D, E, A, D, E	3, 4, 7, 3, 4, 7	4.66	0.21
5.	A, E, D, E, D	4, 3, 4, 3	3.5	0.28
6.	A, E, A, E	4, 7, 4, 7	5.5	0.18
7.	A, B, A, B	2, 7, 2, 7	4.5	0.22
8.	A, B, C, A, B, C	2, 2, 7, 2, 2, 7	3.66	0.27

Please note that all cycles start with an issue into an empty pipeline-state A. We also find that in the 4th column, the minimum of average latency is 3.5.

So, we say that the minimum average latency (MAL) for the cycles 3 and 5 is 3.5. Cycles that permit issues at a constant latency can have a minimum, constant latency (MCL). For example, the only MCLs for this pipeline are 7 and of course 9^+ . The MCL is the smallest integer that is not a factor of any of the forbidden latencies. Like, in our example, the smallest latency that is not a factor of (1, 5, 6, 8) is 7. A latency of 2 will collide with 6 and 8, a latency of 3 will collide with 6 and so on. Constant latencies are, however, of interest because then the control is easier to implement.

Now, we need to find the greedy cycles on the state diagram. Greedy cycles are defined as those simple cycles in which each latency is the minimal latency (outgoing arc) from a state in the cycle. Please note that simple cycle is the one in which each state appears only once.

To determine greedy cycles we follow a procedure :

1. Start from each node of the state diagram.
2. Select arc with smallest latency label until a closed simple cycle is formed.
3. The latencies on the above corresponds to the greedy cycle.

The average latency of any greedy cycle is less than or equal to the number of 1s in the collision vector. It is lower than those of simple cycles. Also, the average latency of greedy cycle is always greater than or equal to MAL. At least one of the greedy cycles will lead to MAL. Thus, the collision free scheduling of pipeline is nothing but finding greedy cycles from the set of simple cycles. The greedy cycles yielding MAL is the final choice to get the maximum throughput. So, **Overall summary** is as shown below :

```

From Reservation table given
↓
Issue latencies are found out
↓ then
Forbidden list/latencies are formed
↓
Collision Vector is formed
↓
State diagram is drawn
↓
Greedy cycles are found
↓
MAL is calculated

```

We are in a position to solve some problems now.

7.3.3.4 Problems based on Reservation tables

Example 1. Consider the following reservation table :

	t_1	t_2	t_3	t_4	t_5	t_6	t_7
S_1	X					X	
S_2			X				X
S_3		X		X			
S_4			X		X		

- (a) Determine latencies in the forbidden list, F and the collision vector, C.
- (b) Draw the state transition diagram.
- (c) List all simple cycles and greedy cycles.
- (d) Determine MAL.
- (e) If clock period, $T = 20\text{ns}$, determine the maximum throughput of the pipeline.

[Pune Univ., B.Tech. 2nd sem; May 2004]

Solution : (a) Forbidden latencies = {5, 4, 2}

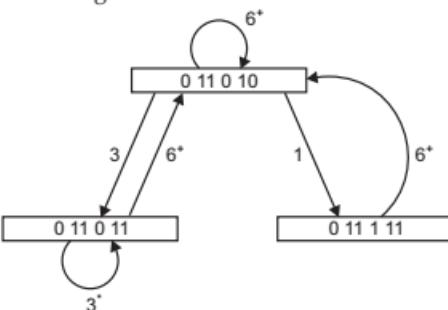
∴ Permissible latencies = {1, 3, 6}

∴ Collision Vector, $V = \{0 \ 1 \ 1 \ 0 \ 1 \ 0\}$

Where 1 – Collision

0 – No collision

(b) The state transition diagram is :



\therefore Initial collision vector (ICV) = 011010
 & Permissible latencies are [1, 3, 6]

$$\therefore \begin{array}{r} \text{ICV} = 011010 \\ 1 \rightarrow 1101 (+) \\ \hline 011111 \end{array} \quad (\text{Right shift by 1})$$

Similarly,

$$\begin{array}{r} \text{ICV} = 011010 \\ 3 \rightarrow 000011 (+) \\ \hline 011011 \end{array} \quad (\text{Right shift by 3})$$

Also,

$$\begin{array}{r} \text{ICV} = 011010 \\ 6 \rightarrow 000000 (+) \\ \hline 011010 \end{array} \quad (\text{Right shift by 6})$$

So, a state transition diagram is drawn above.

(c) List of Latency Cycles is : (6), (3, 6) (1, 6), (3), (3, 3, 6), (3, 6, 6) and (6, 1, 6)

List of Simple Cycles is : (6), (3), (3, 6) and (1, 6)

List of Greedy Cycles is : (3), (1, 6)

(d) MAL = 3

$$(e) \text{ Throughput, } W = \frac{\eta}{t} \times \frac{1}{\text{MAL}}$$

$$\begin{array}{l} \text{Assuming efficiency, } \eta = 1 \\ \text{ans } t = 20 \text{ ns} \end{array}$$

$$W = \frac{1}{20 \times 10^{-9}} \times \frac{1}{3} = 16.67 \text{ MIPS}$$

Example 2. Consider the following reservation table (RT) :

	t_i	1	2	3	4	5	6	7
s_i	X		X					X
s_2				X		X		
s_3			X	X				

Calculate the following :

- Find all forbidden latencies.
- Draw a state transition diagram.
- Find all simple and greedy cycles.
- Find optimal constant latency cycle and minimal average latency.
- If pipeline clock period is 20 ns. Find the throughput of the pipeline.

Solution.

(a) ROW 1 : 2, 4, 6

ROW 2 : 2

ROW 3 : 6

\therefore Set F (of forbidden latencies) = {2, 4, 6}

(b) Permissible latencies are {1, 3, 5, 7⁺}

Now, collision vector, V =

Now,

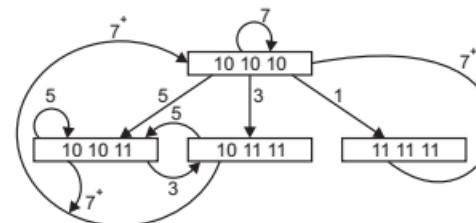
$$I \ C \ V = 1 \ 0 \ 1 \ 0 \ 1 \ 0$$

$$\begin{array}{r} 1 \\ \xrightarrow{\quad} 1 \ 0 \ 1 \ 0 \ 1 \\ \hline 1 \ 1 \ 1 \ 1 \ 1 \end{array}$$

	1	2	3	4	5	6	7
S ₁	X		X				X
S ₂				X		X	
S ₃			X		X		

[Right shifting by 1]

Similarly for states 3, 5, 7 and 7⁺ we can find the next states. Continuing in the same fashion we get the following STD :



(c) Now, we find out the simple, greedy cycles and the average latencies :

Simple Cycles	Average Latency
5	5
7	7
(1, 7)-Greedy	4*
(3, 7)	5
(5, 7)	6
(5, 3)-Greedy	4*
3, 5, 7	5

(d) \therefore Minimum average latency (MAL) = 4.

(e) \therefore Maximum throughput = $\frac{1}{\text{MAL} \times \tau}$
Where τ - clock period

Now,

$$\text{MAL} = 4$$

\therefore and

$$\tau = 20 \text{ ns} \text{ (given)}$$

$$\begin{aligned}\therefore \text{Maximum throughput} &= \frac{1}{4 \times 20 \times 10^{-9}} = \frac{10^6}{80 \times 10^{-13}} \\ &= \frac{10^6}{0.008} = 12.5 \text{ MIPS}\end{aligned}$$

Example 3. Consider the execution of a program of 15000 instructions by linear pipeline processor. The clock rate of pipeline is 25 MHz. Pipeline has five stages and one instruction is issued per clock cycle. Neglect penalties due to branch instructions and out of sequence execution:

- (a) Calculate the speed up program execution by pipeline as compared with that by non-pipelined processor.
- (b) What are the efficiency and throughput of the pipeline processor?

[GGSIPU, M.Tech (CSE/IT)-1st sem. Dec. 2011]

Soluton. Given :

$$f = 25 \text{ MHz}$$

$$\text{Number of instructions} = n = 15000$$

$$\text{Number of pipeline stages} = k = 5$$

$$(a) \quad \text{Speed up} = \frac{nk}{k + (n - 1)} = \frac{(15000)5}{5 + (15000 - 1)} = 4.99$$

$$(b) \quad \text{Efficiency, } \eta = \frac{n}{k + (n - 1)} = \frac{15000}{5 + 14999} = 0.99$$

$$\& \text{ Throughput, } W = \frac{nf}{k + (n - 1)} = \frac{15000 \times 25 \times 10^{-6}}{5 + 14999} = 24.99 \text{ MIPS}$$

Example 4. A processor (P_1) is non-pipelined and has a clock rate of 25 MHz. It has average CPI = 4. Another processor (P_2) has clock rate of 20MHz. P_2 is designed with five stages. P_2 is improved successor of P_1

- (a) If a program of 100 instructions is to be executed on both procesors, what is the speed up of P_2 compared to that of P_1 ?
- (b) Find MIPS rate of P_1 as well as P_2 during execution of the program?

[GGSIPU, M.Tech. (CSE/IT)-1st sem. Dec-2011]

Solution.

$$(a) \quad \text{CPI} = 4$$

$$f_1 = 25 \text{ MHz} \quad f_2 = 20 \text{ MHz}$$

$$n = 100, \quad k = 5$$

$$\therefore \text{Speed-up, } S = \frac{T_1}{T_2}$$

$$\text{But} \quad T_1 = \frac{n \times \text{CPI}}{f_1} = \frac{100 \times 4}{25 \times 10^6} = 16 \mu\text{sec}$$

$$\begin{aligned}T_2 &= [k + (n - 1)] \tau_2 \\ &= \frac{[5 + 99].1}{20 \times 10^6} = \frac{104}{20 \times 10^6} = 5.2 \mu\text{sec.}\end{aligned}$$

$$\therefore S = \frac{T_1}{T_2} = \frac{16}{5.2} = 3.08$$

(b) Throughput, W = Number of instructions executed/second

For pipelined processor, P_2 ,

$$W_{P_2} = \frac{n\tau}{k + (n-1)} / 10^6 = \frac{100 \times 20 \times 10^6}{(5 + 99) \times 10^6} = \frac{2000}{104}$$

= 19.23 MIPS

For Non-pipelined processor, P_1 ,

$$W_{P_1} = \frac{1}{4\tau} = \frac{f}{4} = \frac{1}{4} \times 25 \times 10^6 = 6.25 \times 10^6 \text{ instruction sec.}$$

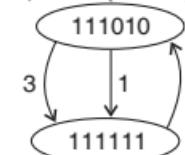
= 6.25 MIPS

Example 5. Determine MAL and throughput of the following instruction pipelining shown in the reservation table

	t_1	t_2	t_3	t_4	t_5	t_6	t_7
s_1	x				x		
s_2		x					x
s_3		x			x		x
s_4		x					x
s_5	x		x				x

[UPTU, B. Tech. (CSE/IT) 8th sem, 2005-08]

Solution. Forbidden list, $F = \{2, 4, 5, 6\}$
Collision vector, $C = \{010111\}$



Simple cycle

1,6

3,6

\therefore MAL = 3.5

$$\text{& throughput} = \frac{1}{3.5} \times 100 \% = 33.3 \%$$

Average latency

$$7/2 = 3.5$$

$$9/2 = 4.5$$

7.3.4 Arithmetic Pipeline Design

Pipelining techniques can be applied to speed-up numerical computations.

7.3.4.1 Computer Arithmetic Principles

In our computers, arithmetic is performed with **finite precision** due to the use of fixed-size memory words or registers. This fixed-point or integer arithmetic offers a fixed range of numbers that can be operated upon. Floating-point arithmetic operates over much increased dynamic range of numbers. In modern computers, fixed-point and floating-point arithmetic operations are performed by separate hardware. Advanced RISC microprocessors usually have the hardware for both operations on the same processor chip. Please note that the finite precision means that the numbers exceeding the limit

must be truncated or rounded to provide a precision within the number of significant bits allowed. In case of floating-point numbers, exceeding the exponent range means error conditions called as **overflow** or **underflow**.

7.3.4.1.1 Fixed-Point Operations

There are three notations followed for fixed-point operations :

- (a) Signed magnitude.
- (b) One's complement.
- (c) Two's complement.

Out of these, two's complement notation is more popular in most of the computers because of its unique representation of all numbers including zero.

Add, subtract, multiply and divide are the four basic arithmetic operations.

7.3.4.1.2 Floating-Point Operations

There are four basic arithmetic operations for a pair of floating point numbers.

Say, $x = (m_x, e_x)$

and $y = (m_y, e_y)$ are two floating point numbers.

Where m_x and m_y are their mantissa and e_x and e_y are exponents.

For simplicity and clarity, say $e_x \leq e_y$ and base $r = 2$. Then the four basic operations on x and y will be :

1. $x + y = (m_x \times 2^{e_x - e_y} + m_y) \times x^{e_y}$
2. $x - y = (m_x \times 2^{e_x - e_y} - m_y) \times x^{e_y}$
3. $x \times y = (m_x \times m_y) \times 2^{e_x + e_y}$
4. $x \div y = (m_x + m_y) \times 2^{e_x - e_y}$

From these equations, it is clear that we can divide these operations into two groups—one for the exponent operations and the other for mantissa operations. These two groups of operation demand double hardware for floating point unit as compared to that required in a fixed point unit.

7.3.4.1.3 Elementary Functions

Elementary functions include trigonometric, exponential, logarithmic and other transcendental functions. Truncated polynomials or power series can be used to evaluate the elementary functions such as $\sin x$, $\ln x$, e^x , $\cosh x$, $\tan^{-1} y$, x^3 etc. Please note that the computer arithmetic can be implemented by hardwired random logic circuitry as well as by table lookup using ROM or RAM in the memory. Table look ups can be used to see the values of frequently used constants. Hashing will provide a faster access to these tables.

7.3.5 Arithmetic Pipeline Stages

Depending on the arithmetic operation (like add, subtract, multiply, divide squaring, square roots, logarithm etc.) which is to be implemented, different pipeline stages in an arithmetic unit require different hardware logic. Basically, we need some hardware for **add and shift operations only** since all operations involve these two operations only. **Arithmetic and logical shifts can be easily implemented with shift registers.** **High speed addition requires the use of carry-propagation adder (CPA) which adds two numbers** and produces an arithmetic sum or it requires a **carry-save adder (CSA)** to add three input numbers and produce one sum output and a carry output. Let us now tabulate the differences between CSA and CPA.

CPA	CSA
<ol style="list-style-type: none"> 1. It adds two numbers and produces its sum. 2. When CPA is used to built a cascade of full adder then the carry-out (C_{out}) of the lower stage is connected to the carry-in (C_{in}) of a higher stage. 	<ol style="list-style-type: none"> 1. It adds three numbers and produces one sum output and a carry output. 2. When CSA is used to built a cascade of full adder then the carryout terminals of all stages are taken as the output lines for a carry-vector, C.

7.3.6 Different pipeline designs

CPAs and CSAs can be used to design a multiply pipeline and a divide pipeline. We discuss these one by one in the next subsection.

7.3.6.1 Multiply pipeline design

Let us now study, how to use a number of CSAs for multiple-number addition. The same will serve the purpose of pipeline multiplication. We shall consider an example of multiplication of two 6-bit numbers. This is shown below in Fig. 7.17.

As is clear from the Fig. 7.17. that this 6-bit, 2 number multiplier pipeline has five stages :

Stage 1 : In this stage, we generate all $6 \times 6 = 36$ immediate product terms i.e., these terms form the six rows of the shifted multiplicands, W_i where $i = 1, 2, \dots, 6$.

These 6 numbers are passed to stage-2.

Stage 2 : The six numbers are then added by forming two groups each groups having three numbers to get two sum vectors and two carry vectors. Two CSAs are used for this addition.

Stage 3 : The four results generated by stage-2 are added in this stage by using another CSA. Thus, three of the above results are added and the fourth (Sum Vector) is forwarded to stage-4. This CSA also generates two results. In all, three results are fed to stage-4.

Stage 4 : This stage consists of a CSA that adds all the three vectors received and generates one sum vector and one carry vector. They are passed to stage-5.

Stage 5 : It adds the sum vector and carry vector received from stage 4, using a CPA and the result is one number which is the final output, the Product $P = X \times Y$.

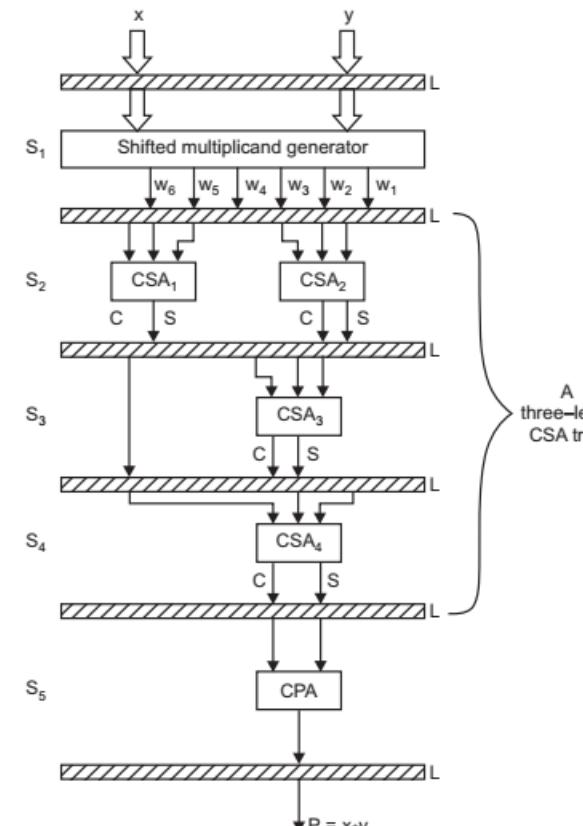


Fig. 7.17. A pipelined 6-bit Multiplier.

Please note that there are totally four CSAs that are interconnected to form a three-level CSA tree. This CSA tree merges six numbers into two : The sum vector, S and the carry vector, C . The final stage is a CPA that adds sum vector, S and a carry vector, C to produce the product, P . The above CSA tree that adds only multiple single bit numbers is known as a bit-slice Wallace tree.

However, the cost of the hardware of this multiplier is quite high. So, some **trade-off is required** between cost and evaluation time. So, we have an **alternative approach** also and it is called as an **iterative CSA tree for a pipelined multiplier**. In this case, a CSA tree pipeline of Fig. 7.17 is modified to have a feedback so that it allows the multiple pass usage of CSAs. Such an iterative multiplier is shown in Fig. 7.18. below.

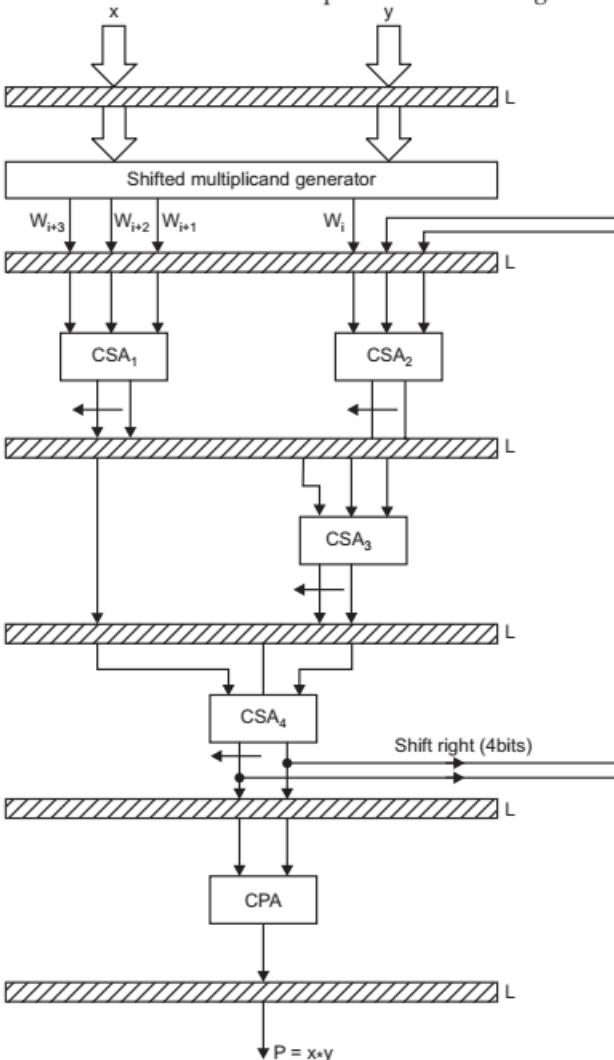


Fig. 7.18. Iterative CSA tree for pipelined Multiplication

In this approach, two input ports of CSA tree are connected with the feedback carry vector and sum vector. CPA is needed to be expanded to handle increased operand width. This pipeline can be used to merge four additional multiplicands per iteration. For instance, to multiply two 32-bit numbers, only eight iterations will be required with this CSA tree with feedback.

Now, one pass 32-input CSA tree pipeline requires the use of 30 CSAs in eight pipeline stages. On the other hand, the iterative CSA tree requires 4 CSAs totally. Thus, there is saving of cost of 26 additional CSA (*i.e.*, $30 - 4 = 26$ CSAs). Thus, the iterative approach saves the hardware cost significantly as compared to single pass approach. Even though the hardware cost reduces yet the evaluation time increases.

7.3.6.2 Division pipeline design

Many models of high performance computers *e.g.*, IBM 360/370, CDC-6600/7600, share the same hardware. This is possible because of application of convergence division method.

Convergence Division Method

Assume that we want to compute $Q = M/D$

Where Q is quotient

M is dividend

D is divisor.

We will consider normalized binary numbers.

$\therefore 0.5 \leq M < D < 1$ to avoid overflow.

$$\text{Now, } Q = \frac{M}{D} = M \times \frac{1}{D} = M \times D^{-1}$$

i.e., multiplication of M and inverse of D .

$$\text{Let } \delta = 1 - D \text{ and } 0 < \delta < 0.5 \quad \dots(1)$$

$$\text{and } R_i = 1 + \delta^{2^i} ; \text{ for } i = 1, 2, \dots k \quad \dots(2)$$

be the successive converging factors.

$$\begin{aligned} \therefore Q &= \frac{M}{D} = \frac{M \cdot R_1 \cdot R_2 \dots R_k}{D \cdot R_1 \cdot R_2 \dots R_k} \\ &= \frac{M \cdot R_1 \cdot R_2 \dots R_k}{(1 - \delta) \cdot R_1 \cdot R_2 \dots R_k} \quad (\text{From equation-1}) \end{aligned}$$

$$\begin{aligned} \therefore R_1 &= 1 + \delta \\ R_2 &= 1 + \delta^2 \\ R_3 &= 1 + \delta^4 \\ &\vdots \\ R_k &= 1 + \delta^{2^{k-1}} \quad [\text{From equation-2}] \end{aligned}$$

$$\therefore Q = \frac{M(1 + \delta)(1 + \delta^2)(1 + \delta^4)\dots(1 + \delta^{2^{k-1}})}{(1 - \delta)(1 + \delta)(1 + \delta^2)(1 + \delta^4)\dots(1 + \delta^{2^{k-1}})}$$

$$\begin{aligned} \text{Let } D_i &= (1 - \delta)(1 + \delta)(1 - \delta^2)(1 + \delta^2)\dots(1 + \delta^{2^{k-1}}) \\ &= (1 - \delta^2)(1 + \delta^2)\dots(1 + \delta^{2^{k-1}}) \end{aligned}$$

$$\begin{aligned}
 &= (1 - \delta^4)(1 + \delta^4) \dots (1 + \delta^{2^{k-1}}) \\
 \therefore D_i &= (1 - \delta^{2^i}) \\
 \text{As } O < \delta < 0.5, \delta^{2^i} &\rightarrow 0 \text{ as } i \rightarrow \infty \\
 \therefore D_i &= 1 \\
 \therefore Q &= M \cdot R_1 R_2 \dots R_k \quad \dots(3)
 \end{aligned}$$

Thus, division can be carried out by repeated multiplication as given by equation-3. Please note here that the smaller is the fraction δ , the faster convergence is possible. In IBM 360 machine, same hardware is shared for floating point multiply and divide operation. For speeding up multiplication operations multiplier recoding techniques are used. After recoding, six multiplicand multiples are obtained. A pipelined multiply/divide unit of IBM 360 machine is shown in Fig. 7.19 below.

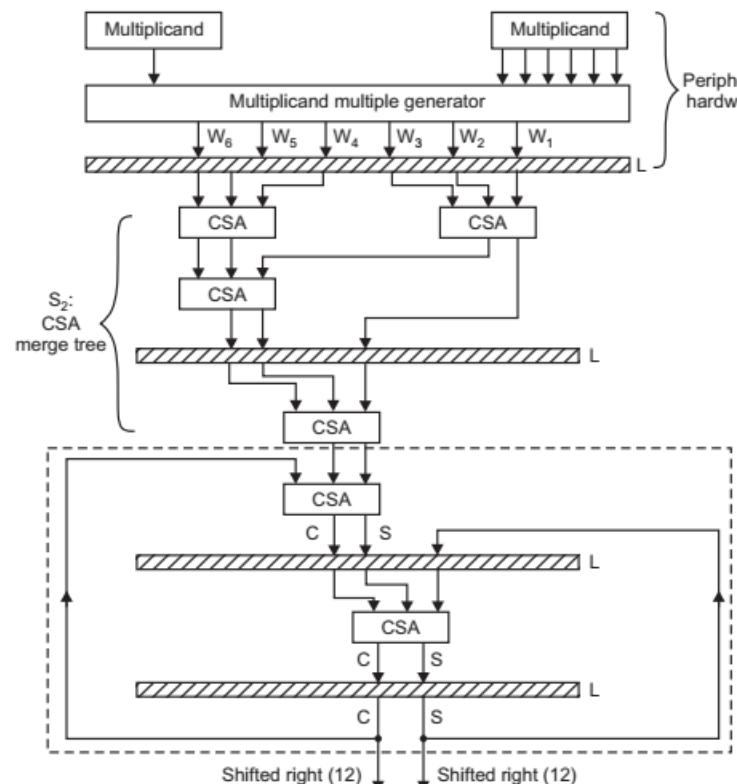


Fig. 7.19. Pipelined Multiply/Divide unit of IBM-360.

As shown in Fig. 7.19, there are two parts of multiply/divide unit :

Part I : The iterative hardware for multiple multiplicand addition as shown inside the dashed-line box.

Part II : The peripheral hardware for input reservation, prenormalization, multiplier recoding, exponent arithmetic, carry propagation and output storage.

A convergence division method is used for division. Therefore, above iterative-multiply unit can also perform division without additional facilities. Herein, there are 4 areas where parallelism has been found :

- (a) Parallel operations of ADD unit and multiply/divide unit within floating point unit.
- (b) Pipelining in ADD and Multiply/Divide unit.
- (c) Parallel execution within the iterative multiply hardware.
- (d) Parallel operations of fixed point unit and floating-point unit.

7.3.7 Instruction Pipeling Design

A stream of instructions can be executed by a pipeline in an overlapped manner.

7.3.7.1 Phases of instruction execution

A typical instruction execution consists of a sequence of operations namely :

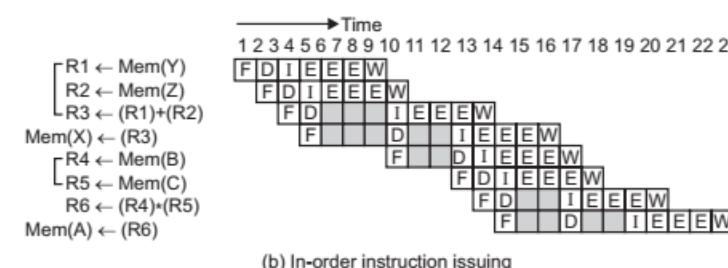
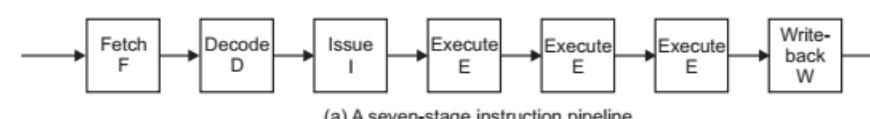


Fig. 7.20. Instruction Execution Phases.

These phases are ideal for overlapped execution on a linear pipeline. Each phase may require one or more clock cycles to execute, depending on the instruction type and processor/memory architecture used.

7.3.7.2 Pipelined Instruction Processing

Pipelined Instruction Processing. A typical instruction pipeline is depicted in Fig. 7.21. The **fetch stage** (F) fetches instructions from a cache memory, presumably one per cycle. The **decode stage** (D) reveals the instruction function to be performed and identifies the resources needed. Resources include general-purpose registers, buses, and functional units. The **issue stage** (I) reserves resources. Pipeline control interlocks are maintained at this stage. The operands are also read from registers during the issue stage.



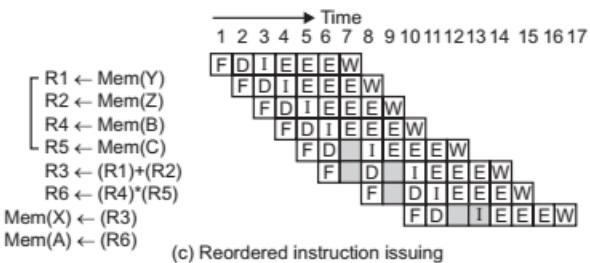


Fig. 7.21. Pipelined execution of $X = Y + Z$ and $A = B \times C$.

The instructions are executed in one or several *execute stages* (E). Three execute stages are shown in Fig. 7.21 (a). The last **writeback stage** (W) is used to write results into the registers. Memory load or store operations are treated as part of execution. Figure 7.21 shows the flow of machine instructions through a typical pipeline. These eight instructions are for pipelined execution of the high-level language statements $X = Y + Z$ and $A = B \times C$. Assume **load and store** instructions take four execution clock cycles, while floating-point **add and multiply** operations take three cycles.

Figure 7.21(b) illustrates the issue of instructions following the original program order. The shaded boxes correspond to idle cycles when instruction issues are blocked due to resources latency or conflicts or due to data dependences. The first two load instructions issue on consecutive cycles. The add is dependent on both loads and must wait three cycles before the data (Y and Z) are loaded in.

Similarly, the **store** of the sum to memory location X must wait three cycles for the **add** to finish due to a flow dependence. There are similar blockages during the calculation of A. The total time required is 17 clock cycles. This time is measured beginning at cycle 4 when the first instruction starts execution until cycle 20 the last instruction starts execution. This timing measure eliminates the unduly effects of the pipeline “**startup**” or “**draining**” delays.

Figure 7.21(c) shows an improved timing after the instruction issuing order is changed to eliminate unnecessary delays due to dependence. The idea is to issue all four load operations in the beginning. Both the add and multiply instructions are blocked fewer cycles due to this data prefetching. The reordering should not change the end results. The time required is being reduced to 11 cycles, measured from cycle 4 to cycle 14.

7.3.8 Mechanisms for Instruction Pipelining

There are different mechanisms to smoothen the pipeline flow and to remove bottlenecks and unnecessary memory-access operations.

7.3.8.1 Prefetch Buffers

In one memory-access time, a block of consecutive instructions are fetched into a prefetch buffer. There are 3 types of buffers :

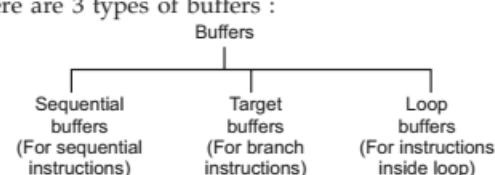


Fig. 7.22. Types of buffers.

Please note here that each buffer has its own role as shown in Fig. 7.22. Also note that both sequential and target buffers operate in a FIFO order. These buffers become part of the pipeline as additional stages. A conditional branch instruction causes both sequential and target buffers to fill with instructions. These two buffers work alternately to prevent a collision between the instructions flowing into and out of the pipeline.

A third type of buffer called as a **loop buffer** is maintained by the **fetch stage of the pipeline**. All these three buffers are shown in Fig. 7.23. below.

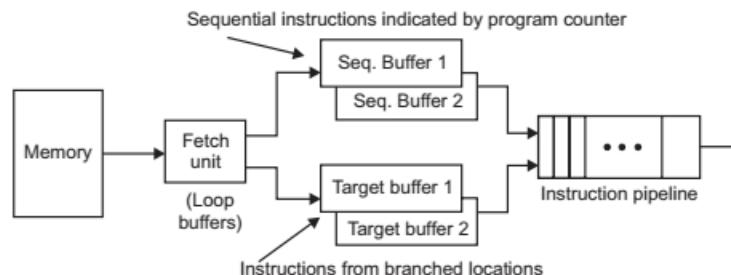


Fig. 7.23. Three prefetch buffers.

The loop buffer operates in two steps :

1. It contains instructions sequentially ahead of the current instruction. This saves the instruction fetch time from memory.
2. It recognizes when the target of a branch falls within the loop boundary

The advantages of this scheme is that unnecessary memory accesses can be avoided.

For Example : CDC-6600 and Cray-1 have used loop buffers.

7.3.8.2 Multiple functional units

Multiple Functional Units.

Sometimes a certain pipeline stage becomes the bottleneck. This stage corresponds to the row with the maximum number of checkmarks in the reservation table. This bottleneck problems can be alleviated by using multiple copies of the same stage simultaneously. This leads to the use of multiple execution units in a pipelined processor design (Fig. 7.24.)

Sohi (1990) used a model architecture for a pipelined scalar processor containing multiple functional units (Fig. 7.24.) In order to resolve data or resources dependences among the successive instructions entering the pipeline, the **reservation stations (RS)** are used with each

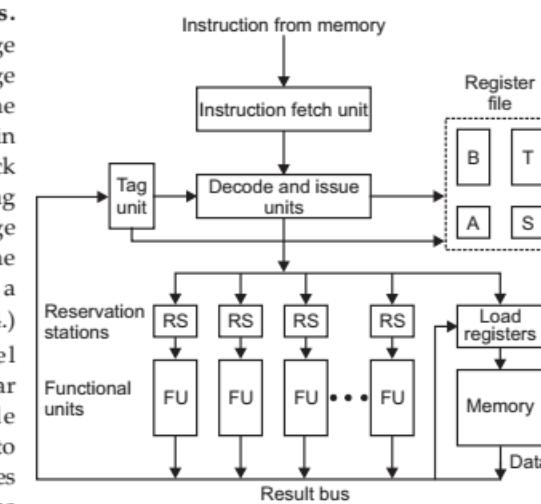


Fig. 7.24. Pipeline Processor with many functional units.

functional unit. Operands can wait in the RS until its data dependences have been resolved. Each RS is uniquely identified by a **tag**, which is monitored by a **tag unit**.

The tag unit keeps checking the tags from all currently used registers or RSs. This register tagging technique allows the hardware to resolve conflicts between source and destination registers assigned for multiple instructions. Besides resolving conflicts, the RSs also serve as buffers to interface the pipelined functional units with the decode and issue units. The multiple functional units are supposed to operate in parallel, once the dependences are resolved. This will alleviate the bottleneck in the execution stages of the instruction pipeline.

7.3.8.3 Internal data forwarding and Register tagging

There are 2-techniques that are used for instruction pipelining. They are discussed below.

I. Internal forwarding : It refers to a “short-circuit” technique for replacing unnecessary memory accesses by register-to-register transfers in a sequence of fetch-arithmetic-store operations. (∴ Register access is faster than memory access).

It is also of 3 types :

(a) **Store-fetch forwarding** : The following sequence of the 2-operations store then-fetch can be replaced by 2-parallel operations, one store and one register transfer :

$$\begin{aligned} M_i &\leftarrow (R_1) \text{ (store)} \\ M_2 &\leftarrow (M_i) \text{ (fetch)} \end{aligned} \quad \left. \right\} 2 \text{ memory accesses}$$

being replaced by :

$$\begin{aligned} M_i &\leftarrow (R_1) \text{ (store)} \\ R_2 &\leftarrow (R_1) \text{ (reg. transfer)} \end{aligned} \quad \left. \right\} 1 \text{ memory access}$$

e.g.,

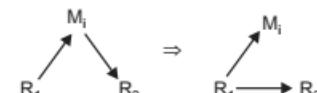


Fig. (a)

(b) **Fetch-fetch forwarding** : The following 2-fetch operations can be replaced by 1-fetch & 1-register transfer. (Fig. (b)) Again 1-memory access has been eliminated:

$$\begin{aligned} R_i &\leftarrow (M_i) \text{ (fetch)} \\ R_2 &\leftarrow (M_i) \text{ (fetch)} \end{aligned} \quad \left. \right\} 2 \text{ memory accesses}$$

being replaced by :

$$\begin{aligned} R_1 &\leftarrow (M_i) \text{ (fetch)} \\ R_2 &\leftarrow (R_1) \text{ (reg. transfer)} \end{aligned} \quad \left. \right\} 1 \text{ memory access}$$

e.g.,



Fig. (b)

(c) **Store-store overwriting** : Following 2-memory updates (stores) of the same word can be combined into one, since the second store overwrites the first :

$$\begin{aligned} M_i &\leftarrow (R_1) \text{ (store)} \\ M_i &\leftarrow (R_2) \text{ (store)} \end{aligned} \quad \left. \right\} 2 \text{ memory accesses}$$

being replaced by :

$$M_i \leftarrow (R_2) \text{ (store)} \quad \} 1 \text{ memory access}$$

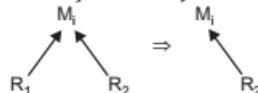


Fig. (c)

II. Register tagging : "Refers to the use of tagged registers, buffers & reservation stations for exploiting concurrent activities among multiple arithmetic units."

Application : This technique may be used in almost any computer that has multiple functional pipelines & accumulators.

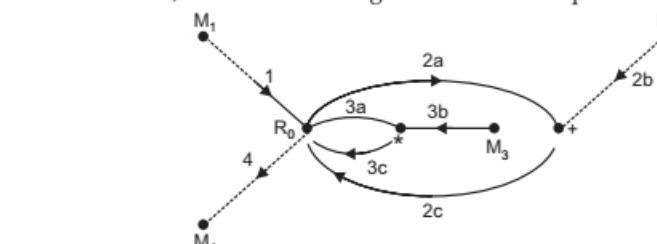
We are in a position to solve the problem now.

Example 1. The inner loop of a certain program is completed to perform the following operations in a sequence :

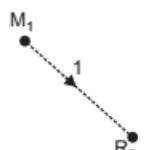
1. $R_0 \leftarrow (M_1)$ (fetch)
2. $R_0 \leftarrow (R_0) + (M_2)$ (add)
3. $R_0 \leftarrow (R_0) \times (M_3)$ (mul)
4. $M_4 \leftarrow R_0$ (store)

Apply your internal-forwarding technique to simplify of sequence of such 4 executions, so that minimum number of memory references are done ? Explain.

Solution. Now, we first draw original data flow-sequence for the given problem :



i.e., 1st instruction is $R_0 \leftarrow (M_1)$



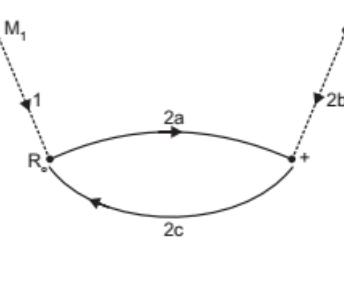
Where '1' represents instruction 1 above.

Similarly, 2nd instruction is :

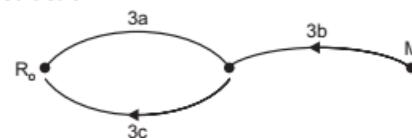
$$R_0 \leftarrow (R_0) + (M_2)$$

i.e., Contents of R_0 are added to contents of M_2 and result in again brought to R_0 so we can draw :

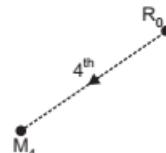
i.e., add (R_0) to (M_2) by using '+' or "adder unit" & then bring that result to R_0 (2c step). i.e., our 2nd instruction need 3-substeps



Similarly, for 3rd instruction.



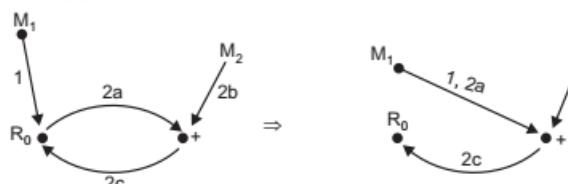
Similarly, for 4th instruction.



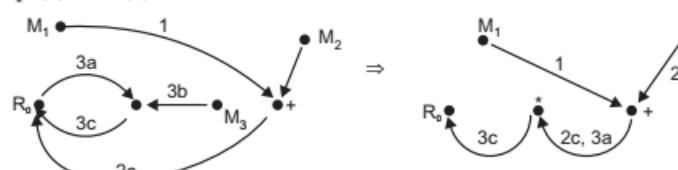
So, we get the above original data flow diagram/sequence.

Now, we apply our internal-forwarding technique :

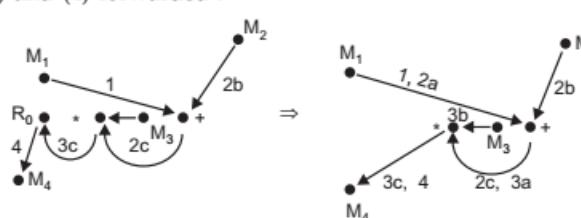
1. Step (1) and (2) forwarded :



2. Step (2) and (3) forwarded :



3. Step (3) and (4) forwarded :



Note : 1. Nodes in the graph correspond to the memory cells, registers, an adder, or a multiplier.

2. Memory access are shown by thick arrows and register transfers by thin arrows.

3. We have, thus, enhanced rate of execution of these 4-instructions.

Application : Both internal-forwarding & resource tagging have been practiced in IBM model 91 floating-point execution unit.

7.3.9 Pipeline Hazards

Pipeline hazards :

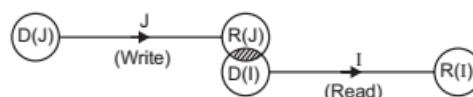
1. Pipeline hazards are caused by “resource-usage conflicts” among various instructions in the pipeline. Such hazards are triggered by interinstruction dependencies.
2. When successive instructions overlap their fetch, decode and execution through a pipeline-processor, inter instruction dependencies may arise to prevent the sequential data flow in the pipeline.

For example, an instruction may depend on the results of a previous instruction. Until the completion of the previous instruction, the present instruction cannot be initiated into the pipeline. In other instances, two (2) stages of a pipeline may need to update the same memory location. Hazards of this sort, if not properly detected and resolved, could result in an “interlock-situation”, in the pipeline or produce unreliable results by overwriting.

7.3.9.1 Data Hazards

There are 3-classes of data dependent hazards, according to various data update patterns :

- (a) **Write after read (WAR) hazards** : It may occur when J-attempts to modify some data object that is read by I. i.e.,



[∵ it is modified by J, So, J writes]

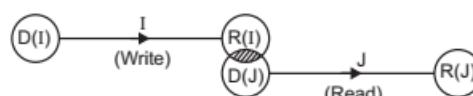
∴ The necessary condition for this hazard are :

$$D(I) \cap R(J) \neq \emptyset \text{ for WAR} \quad \dots(1a)$$

↑

f_i or null

- (b) **Read-after-write (RAW) hazards** : A RAW hazard between the two (2) instructions I & J may occur when J attempts to read some data object that has been modified by I. i.e.,



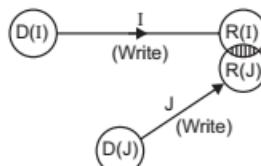
[∵ modified by I, ∵ I writes]

∴ The necessary condition for this hazard is :

$$R(I) \cap D(J) \neq \emptyset \text{ for RAW} \quad \dots(1b)$$

(null)

- (c) **Write-after-write (WAW) hazards** : A WAW hazard may occur if both I and J attempt to modify the same data object. i.e.,



[\because both I & J modify. So, both write]

\therefore The necessary condition for this hazard is :

Note : RAR is not there because when we do read-after-read nothing is changed, so RAR is no problem.

$$\begin{aligned} R(I) \cap R(J) &\neq \emptyset \text{ for WAR} \\ &\uparrow \\ &(f_i) \text{ or null} \end{aligned} \quad \dots(1c)$$

\therefore Improper timing & data dependencies may create some hazardous situations.

7.3.9.2 Control Hazard

“Resource-objects” are used to refer to **working-registers**, **memory-locations** & special flags.

$$\begin{aligned} \therefore \quad (\text{resource-object}) &= \text{data-object} \\ &\uparrow \\ &\text{content of} \end{aligned}$$

\therefore Each instruction can be considered as a mapping from set of data objects to a set of other data objects.

Domain, $D(I)$, of an instruction I is defined as the set of ‘resource-objects’ whose contents may affect the execution of I (*i.e.*, operands only).

Range, $R(I)$, of an instruction, I is the set of resource objects whose data objects may be modified by execution of instruction I (*i.e.*, results of execution of instruction)

Obviously, the operands to be used in an instruction execution are retrieved (read) from its domain and the results will be stored (written) in its range.

Consider, execution of 2 instruction I & J in a program. Instruction J appears after instruction I in the program. Instruction J appears after instruction I in the program. There may be none or other instructions between instruction I & J . Instruction J may enter the execution pipe before or after the completion of the execution of instruction I . The improper timing & data dependencies may create some hazards situations.

How to do Hazard detection ?

1. It can be done in the **instruction fetch stage** of a pipeline by comparing the domain and range of the incoming instruction with those of the instruction being processed. When any one of the condition of equations, 1a, 1b, & 1c is detected, a warning signal can be generated to prevent the hazard from taking place.
2. Another approach is to allow the incoming instruction through the pipeline and **distribute the detection to all potential pipeline stages**. It offers better flexibility at the **expense of increased hardware control**.

7.3.9.3 Hazard Resolution

How to resolve such hazards ?

1. Once a hazard is detected, the system should resolve the interlock situation. Consider the instruction sequence { ... I, I + 1, ... J, J + 1, ... } in which a hazard has been detected between the current instruction, J and a previous instruction I. A straight forward approach is to stop the pipe & to suspend the execution of instruction J, J + 1, J + 2 ... , until the instruction, I has passed the point of resource conflict.

In order to avoid RAW-type of hazards, IBM engineers developed a **short-circuiting approach, also called as data-forwarding**, which forwards multiple copies of the data to as many waiting instruction as may wish to read it.

Note : Another type of hazards is due to the job-scheduling problem.

Once a task is initiated in a static pipeline, its, flow pattern is fixed. An 'initiation' refers to the start of a single function evaluation when 2 or more initiations attempt to use the same stage at the same time, a collision results. Thus, the job sequencing problem is "To properly schedule queued tasks awaiting initiation in order to avoid collisions and to achieve high throughput."

7.3.9.4 Problems based on pipeline Hazards

Example 1. Consider the following instructions ($I_1 - I_5$) and identify various hazards :

```

 $I_1 : R_1 \leftarrow (M(R_4)) ; Load$ 
 $I_2 : R_2 \leftarrow R_1 + R_2 ; Add$ 
 $I_3 : M[R] \leftarrow R_2 ; Store$ 
 $I_4 : R_2 \leftarrow R_6 \text{ and } R_3 ; AND \text{ (logical)}$ 
 $I_5 : R_6 \leftarrow R_6 + 1 ; Increment ?$ 

```

Solution. The instruction $\langle I_1, I_2 \rangle$ pose a RAW hazard.

$\langle I_2, I_4 \rangle$ pose WAW hazard and $\langle I_4, I_5 \rangle$ pose a WAR hazard.

Example 2. Identify various hazards in the following instruction stream :

```

 $I_1 : R_2 \leftarrow R_2 + R_3$ 
 $I_2 : \text{if zero } (R_2) \text{ then goto L3 ?}$ 

```

Solution. $\langle I_1, I_2 \rangle$ show RAW Hazard.

Data dependency hazards are problematic. They may prevent simultaneous processing of two instructions in the pipeline. As a result, they disrupt and reduce throughput.

Example 3. Identify the hazards in the following instruction stream :

```

 $I_1 : R_2 \leftarrow R_2 + R_3$ 
 $I_2 : \text{if } R_2 = 0 \text{ then goto L3}$ 
 $I_3 : M[500] \leftarrow R_2$ 
 $I_4 : L3 : M[600] \leftarrow M[500] + R_5 ?$ 

```

Solution. $\langle I_1, I_2, I_3 \rangle$ show RAW hazard. $\langle I_2, I_3 \rangle$ show control hazard.

Here, I_3 is the constituent of control dependency hazard.

Example 4. Identify all of the RAW, WAR, WAW and control hazards in the following instruction sequence :

```

DIV r2, r5, r8
SUB r9, r2, r7
ASH r5, r14, r6

```

*MUL r11, r9, r5
BEQ r10, 0, r12
OR r8, r15, r2 ?*

[Pune Univ., B. Tech. 2nd sem., Dec.2003]

Solution. Let us denote all these instructions by statement numbers – S1 to S6 as follows :

*S1 : r2 ← r5/r8
S2 : r9 ← r2/r7
S3 : r5 ← r14 << r6
S4 : r11 ← r9 × r5
S5 : r10 ← r12
S6 : r8 ← r15 OR r2.*

- (i) Now, flow dependencies in the program are :
 - (a) S1 → S2
 - (b) S3 → S4
 - (c) S1 → S6
 - (d) S2 → S4
 ∴ The program has 4 RAW hazards.
- (ii) No antidependencies in the program.
 ∴ No WAR hazards.
- (iii) No output dependencies in the program.
 ∴ No WAW hazards.

7.3.10 Dynamic Instruction Scheduling

There are three approaches used here. They are as follows :

- (a) Static Scheduling Approach.
- (b) Register Scoreboardig Approach.
- (c) Tomasulo's Approach.

7.3.10.1 Static Scheduling

In this approach, hazard detection is conducted to a fixed point in the pipeline usually at the instruction decode stage. But it's main disadvantage is that all dependencies or collisions cannot be resolved in advance and hence potential hazards have to be treated as actual hazards.

7.3.10.2 Register Scoreboarding

A separate register that has one bit to identify each register of the register file is called as a **scoreboard**. This bit is set if the corresponding register is in use. A computer with 32 registers will have a 32-bit score board to indicate the busy/free status of registers. When an instruction uses a set of registers, the bits corresponding to those registers are set to 1 in the scoreboard. They are reset to 0 as soon as the instruction using that register completes execution. Thus, the hardware will examine the scoreboard to see if the registers specified in the instruction are free before executing it.

7.3.10.3 Tomasulo's Approach

Let us assume that there is a usage counter, C_i , associated with each distinct store, S_i . The store includes local registers and main memory. If the memory is interleaved

then each distinct memory module constitutes a distinct store. C_i may take values in the range $\{1, 0, -1, -2, \dots -n_i\}$ where n_i is the maximum number of concurrently executable instructions having S_i as a source.

- If $C_i = 1$, it indicates that some instructions with store, S_p as a destination is being processed by the pipeline at that time. i.e., S_i is being used as a source as well as a destination by more than one instruction.
- If $C_i = 0$, it implies that the store is neither a source nor a destination for any instruction.
- If $C_i = -m$, for some $m \leq n_i$ it indicates that m instructions in the pipeline are using S_i as a source.

Let us also assume that there is a binary busy flag, B_n , associated with each distinct functional unit, FU_n . At any given time, B_n can be 1 or 0.

If $B_n = 1$, it indicates that FU_n is being used by a pipeline stage.

If $B_n = 0$, it indicates that FU_n is not busy.

It is assumed that the hazard detection is carried out in the DECODE stage of the pipeline. If the hazard is absent, DECODE stage completes decoding and pass the result to EADDR stage. If, however, hazard is detected, the DECODE stage will not issue the instruction to the next stage.

DECODE will hold the instruction in a latch until the hazard condition disappears.

Let SOURCE (I_k), DEST (I_k) and UNIT (I_k) represent operand source, result destination set and functional set units set of an instruction, I_k .

There are three (3) independent procedures involved in Tomasulo's algorithm :

Procedure I : DECODE stage will issue I_k when the following conditions are satisfied:

$$\left. \begin{array}{l} C_i = 0 \text{ for all } S_i \in \text{DEST}(I_k) \\ C_j \leq 0 \text{ for all } S_i \in \text{SOURCE}(I_k) \\ B_n = 0 \text{ for all } FU_n \in \text{UNIT}(I_k) \end{array} \right\}$$

These conditions confirm that :

- None of the destination stores of the instructions are being used either as a source or as a destination by any other preceding instruction in the pipeline.
- Each of the source store of the instruction is only being used as a source for other preceding instructions in the pipeline.
- Each of the functional units required to process the instruction is free.

Procedure II : DECODE stage will change usage counters and busy flags, after issuing instruction, I_k to the next stage, as follows :

$$\left. \begin{array}{l} \text{new } C_i = 1 \text{ for all } S_i \in \text{DEST}(I_k) \\ \text{new } C_j = C_j - 1 \text{ for all } S_i \in \text{SOURCE}(I_k) \\ \text{new } B_n = 1 \text{ for all } FU_n \in \text{UNIT}(I_k) \end{array} \right\}$$

Procedure III : After complete processing of the instruction, usage counters and busy flags will be updated as follows :

$$\left. \begin{array}{l} \text{new } C_i = 0 \text{ for all } S_i \in \text{DEST}(I_k) \\ \text{new } C_j = C_j + 1 \text{ for } C_i \in \text{SOURCE}(I_k) \\ \text{new } B_n = 0 \text{ for all } FU_n \in \text{UNIT}(I_k) \end{array} \right\}$$

Action to be taken after hazard detection : Once the hazard is found, it must be resolved. There are two approaches commonly followed for resolution of hazards. They are as follows :

1. Stop the pipeline initiation at the point of hazard unit the instruction has passed the point of resource conflict. Thus, if a hazard is detected between the current instruction I_j and a previous instruction I_i , pipeline is stopped and execution of instructions $I_j, I_{j+1}, I_{j+2} \dots$ is suspended.
2. Second approach is to suspend only instruction I_j and allow instructions $I_{j+1}, I_{j+2} \dots$ to flow down the pipe.

First approach is simple to implement. But it will degrade the performance by losing the advantage of pipelining.

The **second approach** is more sophisticated and hence attractive, but more complex.

Resolution of Collision/Structural Hazard

When a hazard is detected between an instruction I_j and some earlier instruction, issue of I_j to the next stage is delayed until the hazard condition vanishes. As a result, I_j will remain in one of the buffers of DECODE stage. DECODE stage will continuously test the values of C_i and B_n until condition in (1) are satisfied. If it is decided to always have an in order execution of instruction stream then delaying issue of I_j means the instruction $I_{j+1}, I_{j+2} \dots$ following I_j will also be delayed irrespective of any hazard conditions with those instructions. These succeeding instruction are delayed only because of delay of I_j . This problem can be resolved to some extent by introduction of virtual functional units. Till now, we have associated real functional units $FU_n \in \text{UNITS}(I_j)$. Now we associate a set of virtual functional units ($VFUN_1, VFUN_2, \dots, VFUN_q$) with each such FU_n . The number of these virtual function units q can be made as large as required. The main aim behind this idea is to avoid delayed issue of I_j because of unavailability of functional unit. Instead, I_j will be passed to next stage by assuming the availability of virtual functional unit capable of processing it, provided there are no data dependency hazards. Thus, only condition that will stop I_j issue to next stage is a data dependency hazard. Of course, the assumption of virtual functional unit will not help in completing the instruction I_j . It will always need the availability of a real functional unit. But this will remove the hurdle in issue of succeeding instructions provided no data-dependency hazards are detected. Virtual functional units can save the time required to test $B_n = 0$. It is avoided altogether. A simple realization of q virtual functional units is by associating a queue of q latches with a FU_n . Assigning an instruction I_j to a virtual functional unit then means passing I_j to the queue. The instruction will be actually processed by real function unit FU_n by servicing the contents of the queue on first come first serve basis. The concept is shown in Fig. 7.25

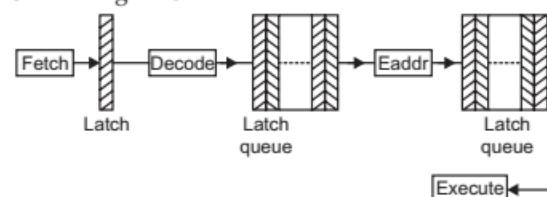


Fig. 7.25. Realization of Virtual Functional Units by Latch Queue

Forwarding : Resolution of Data Dependencies

Assumption of virtual function units will avoid the delay of instructions issue due to collisions or structural hazards. Hence, the only reason remains that will stop an instruction I_j to be issued to next stage is data dependency hazard. This delay of issue

to instruction due to RAW type data dependency hazards can be further reduced by a technique known as **forwarding**. This technique is also introduced by Tomasulo in 1967. This is similar to internal forwarding technique studied in section 7.3.8.3. It can be called as **data forwarding**. In this technique each distinct store S_i is associated with a tag T_i . This tag T_i is an indicator of functional unit FU_n , that is used by some instruction in the pipeline and the output of that FU_n is to be written into S_i . A tag T_i is attached to store S_i whenever S_i is being used as a destination of an instruction and the instruction has been passed to DECODE stage. If for some $S_i \in \text{SOURCE}(I_k)$ (where I_k is the instruction stream processed by pipeline), there is a tag T_i attached to S_i , then a potential RAW hazard is posed. The store S_i has to receive the value, that is produced by a functional unit FU_n and T_i tag indicates this FU_n . In this situation, I_k is conditionally issued to the next stage (EADDR) along with the tag T_i . When FU_n processes I_k and generates a value to be written to S_i , the pipeline control checks all the stores and latches for the presence of tag indicating FU_n . As S_i is the store where such match will be found, the FU_n will write the result to S_i will be removed. When such a match is found with a latch holding an instruction, then the output of FU_n is forwarded to the conditionally issued instruction I_k or the tag associated with I_k may be reset. This indicates that the hazard is eliminated. It is possible that I_k may have to wait for several forwarded values before its RAW hazards are fully resolved (depending on particular type of I_k). Only after the elimination of hazards I_k is processed by the next stage.

Finally note that the forwarding scheme discussed above will avoid the delaying of instructions issued by DECODE stage due to RAW hazards. However, the issued instruction cannot be completed, till the data dependencies have been resolved.

Effect of branching on pipeline performance is described below by a linear instructions pipeline consisting of 5-segments.

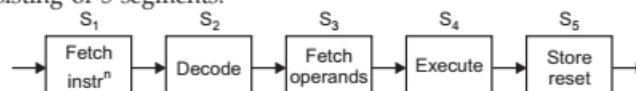


Fig.25(a)

Case I : Now, consider the space-time diagram; When no branch instructions are there:

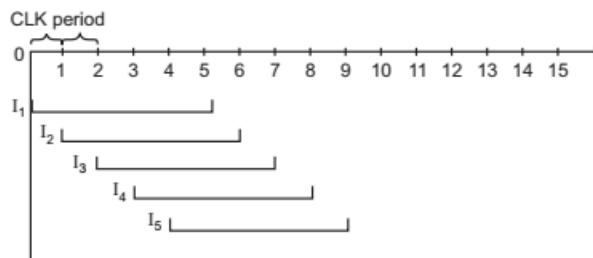


Fig.25(b)

i.e., to execute I_1 , 5-clock periods are required, similarly, for $I_2, I_3, I_4, I_5 \dots I_n$. These instructions are executed by our instruction pipeline in an "overlapped fashion", if branch-type of instructions do not appear. Under such circumstances, once the pipeline is filled up with sequential instructions (non-branch type), the pipeline completes the execution of one instruction per a fixed latency (usually, 1 or 2 clk periods). (Shown in Fig. (b))

Case II : When branching occurs : Now, space-time diagram is

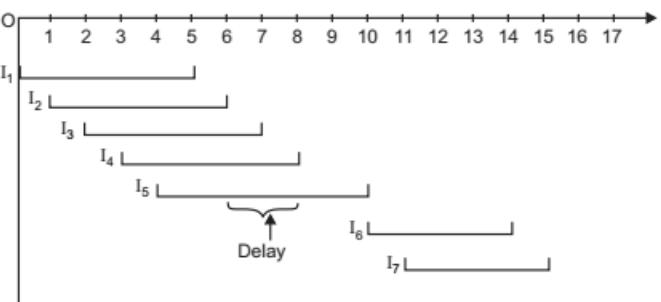


Fig.25(c). Instruction I_5 is a branch instruction.

Whereas, a "branch instruction" entering the instruction pipeline may be halfway down the pipe (such as a "successful" conditional branch instruction) before a branch decision is made. This will cause the PC to be loaded with the new address to which the program should be directed, making all prefetched instructions useless. The next instruction cannot be initiated until the completion of the current branch instruction cycle. This causes extra time delays in order to drain the pipeline. The "Overlapped-action" is **suspended** and the pipeline must be drained at end of the branch cycle. The continuous flow of instruction into the pipeline is thus temporarily interrupted because of presence of a branch instruction. In general, the higher the % of branch-type instruction in a program, the slower a program will run on a pipeline processor. This certainly does not merit the concept of pipelining.

An analytical estimation or Mathematical representation of effect of branching on an n-segment instruction pipeline is given below :

Let $n \rightarrow$ segment pipeline (*i.e.*, pipeline stages) & each instruction require $\rightarrow n$ -pipeline cycles or n/clk periods.

If $m \rightarrow$ instructions (with no branching) then time required $\equiv n + (m - 1)$... (1)

Now, if $p \rightarrow$ probability of branch-instructions and $q \rightarrow$ probability of successful branch instructions.

or

Probability that branch is successful & $m \rightarrow$ instructions to be executed

Then total no. of branch-instructions = mna

Extra time required = $(n - 1)$ Clk cycles

(2)

From (1) & (2) we get :

∴ Total no. of cycles

$$\begin{aligned} \text{Total No. of cycles required to execute } m\text{-instructions} &= (n + (m - 1)) + mpq(n - 1) \\ &\quad (\text{no branch}) \quad (\text{branched}) \end{aligned}$$

m-instructions.
(in branched case)

(in branched case)

$$= \frac{(n + (m - 1)) + mpq(n - 1)}{n}$$

Special Cases :

- As m becomes large, 'performance' of our instruction pipeline is measured by average no. of instructions executed per instruction cycle :

$$\begin{aligned}\text{Performance} &= \frac{Lt.}{m \rightarrow \infty} \frac{m}{(n + (m - 1)) + mpq(n - 1) / n} \\ &= \frac{n}{1 + pq(n - 1)}\end{aligned}$$

- When $p = 0$, (i.e., no branch instructions encountered) then performance reduces to n -instructions per n pipeline clocks, which is ideal i.e.,

$$\text{Performance} = \frac{n}{1+0} \quad [\because p = \text{probability of branch instructions} = 0]$$

For e.g., if

$n = 5$
$p = 20\% = 0.20$
$q = 60\% = 0.60$

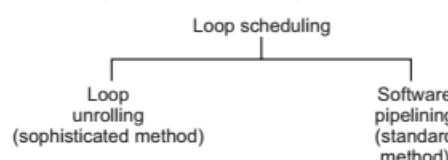
Then Performance = $\frac{5}{1 + (0.20)(0.60)(4)}$
 $= \frac{5}{1.48}$
 $= 3.378 \text{ instructions/instruction cycle}$

Which is less than the ideal execution rate of 5 instructions/5 pipeline cycles.
In other words, an average of **35.2% cycles may be wasted because of branching.**

7.3.11 Advanced Pipelining

Loops are the most fundamental source of parallelism for instruction level pipeline processors (ILP). Here, the regularity of the control structure is used to speed up the computation. Please note that loop scheduling is a focal point of instruction schedulers which have been developed for highly parallel ILP processors, such as VLIWS. Also note that it is a standard feature of the emerging global schedulers.

Loop scheduling is normally done in two ways :



We shall be discussing these two techniques one by one.

7.3.11.1 Loop Scheduling—Unrolling

Loop unrolling means repeating the loop body a number of times and thus the need of inter-iteration code, such as decrementing loop count, testing for loop ends and branching back conditionally between the iteration is eliminated. This reduces

the execution time. Loop unrolling can be implemented quite easily when the number of iterations is already fixed at the compile time. This normally happens with 'do' and 'for' loops. Let us consider an example now :

```
For j = 1 to 4
do
{
    B (j) = 2 × A (j)
}
```

The above loop is replaced by the following code, by the optimizing compiler :

```
B(1) = 2 * A(1)
B(2) = 2 * A(2)
B(3) = 2 * A(3)
B(4) = 2 * A(4)
```

Thus, loop unrolling reduces the execution time at the cost of code length. After loop unrolling, a larger basic blocks is obtained which can be scheduled to execute the instructions in an overlapped manner to improve the performance of the pipeline.

The resulting speed depends on the size of the loop. Please note that when the loop size is large, the relative gain is less whereas if the loop size is small, the gain is more significant.

7.3.11.2 Out of order Execution

Let us consider a k -stage pipeline. This pipeline can be imagined as a system that accepts unprocessed task in some order and generates the same set of tasks but processed in some order. There are two possibilities related to order of one processed tasks :

1. The task may merge out in the same order that they enter. This is known as in-order execution. This is shown in Fig. 7.26. below :

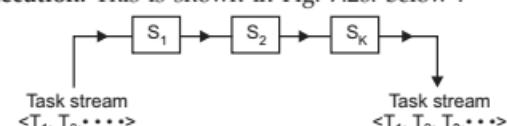


Fig. 7.26. In-order execution.

2. The task may merge out in different order. The necessity of preserving initiation order is relaxed. This is known as out of order execution.

There are some problems with in-order execution. They are as follows :

1. Different tasks may require different processing time.
2. If a task, T_i , is delayed at any stage then all task $T_{i+1}, T_{i+2} \dots$ following T_i will also be delayed. That is, all subsequent instructions would be held up.

Thus, second approach is more efficient. In this out-of-order execution, the necessity of preserving initiation order is relaxed. If any task, T_i , is delayed at a given stage and it is possible that T_{i+1} can proceed down the pipeline, then it is allowed. This technique is also known as forwarding. It may lead to a situation where the output task stream order is some permutation of the initiation order. A k -stage pipeline that performs out-of-order execution is shown in Fig. 7.27.

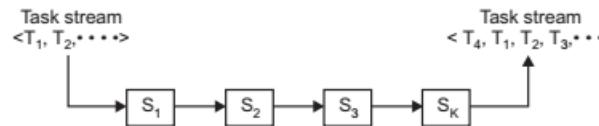


Fig. 7.27. Out-of-order execution (k -stage pipeline)

The out-of-order technique, may increase the overlapping in processing of the instructions and also the throughput. But it needs additional machinery both logical and physical. The machinery should ensure that the throughput is increased and the intended function of the original task stream is preserved.

7.3.11.3 Instruction Scheduling

It is usually advantageous to reorder instructions on pipelined super-computers. Reordering of instructions will lead to minimization of delays due to instruction stalls (wait) because of the dependencies. In case an instruction, 'I' and $(I + 1)$ are dependent, i.e., $(I + 1)$ cannot be completed until the result of 'I' is available then it is desirable to place other independent instructions between I and $(I + 1)$ so that 'I' can be complete before $(I + 1)$ is attempted. This process of arranging sequences of straight line code so that stall will be eliminated or reduced is known as **Instruction Scheduling**. Usually, instruction scheduling is performed on the basic blocks. Basic blocks are the straight line code of which only the first instruction can be the target of branch instruction and only the last instruction can be a branch instruction.

7.3.11.4 Trace Scheduling

A technique that was initially developed for horizontally micro-coded machine. Then it is a reimplement for VLIW (Very Large Instruction Word) machines as an instruction scheduling technique. VLIW machines demand more parallelism than is typically available within a basic block. Hence, additional instruction are found in the branches to execute. Trace scheduling is based on the point that branch decisions can be predicted at the compilation time with some reliability. Branch decisions can be taken based on software heuristics or on using profiles of the previous program execution.

Trace scheduling involves a concept of a trace. **Trace is possible execution path within a loop.** A trace may include conditional branches and joint points. This is shown below in Fig. 7.28.

A **trace scheduler** identifies the traces in a program initially and then selects the most likely trace and schedules it as an entity. Then, the next likely trace is selected and the process is repeated till the whole program is scheduled. Inner loops of programs are unrolled so that maximum parallelism can be

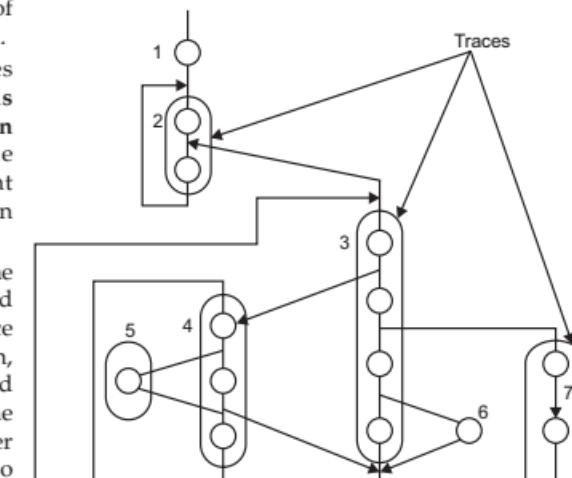


Fig. 7.28. Flow graph and traces.

exposed for trace scheduling. A trace can have multiple entry and exit points. Sometimes it becomes necessary to insert a compensation code to preserve correctness.

For example : Consider a piece of code given below :

```
x = y  
if      a = 1 go to L1  
z = 10  
:  
L1:      .....
```

Now, if an assignment statement is moved from its position to a position after the conditional branch, then it becomes necessary to insert a compensation code. It becomes necessary to repeat the assignment statement at the target of the conditional branch, so that it is always executed regardless of the value of 'a' as shown below.

```
if      a = 1  go to L1  
x = y  
z = 10  
:  
L1:      x = y
```

Drawbacks of Trace Scheduling :

1. The compilation time is more.
2. The generated block size is more.

Trace scheduling was developed as a strategy for analysing VLIW architecture and it is proved to be very effective in generating highly parallel code for wide-word machines.

For processors with a large number of issues per cycle, in which conditional or predicted execution is not appropriate or not supported, trace scheduling is very useful.

In such processors, simply loop unrolling may not be possible and may not be even sufficient to uncover enough ILP so that the processor may be kept busy.

There are two steps in trace scheduling :

Step 1 : Trace Selection : In this step, a likely sequence of basic block is found, whose operations will be put together into a smaller number of instructions *i.e., traces*. Long traces are generated by loop unrolling. Using static branch prediction, other conditional branches are also selected. As a result, traces are produced as a straight line sequence formed by joining many such basic blocks.

Step 2 : Trace Compaction : This process makes the trace compact by squeezing the trace into small number of wide instructions. This is nothing else but code scheduling. The process of trace compaction also attempts to move operations as early as possible. It packs the operation into a few wide instructions.

Advantage of Trace Scheduling : It simplified the decision related to global code motion.

7.3.11.5 Software Pipelining

Software pipelining is the process of pipelining the successive iterations of a loop in the source code. The term is analogous to hardware pipelining. The operations of single iteration are divided into ' k ' stages and a single iteration performs stage-1 from iteration- i , stage-2 from iteration ($i - 1$) and so on. The concept is shown with an

example now. Consider the following loop :

```
For      i = 1 to 5
do
{
    A(I) = A(I) * 2 + C
}
```

This is an example of DO-all loop. All iterations are independent in this loop. Let us assume that cycle time of **memory access** is one and that of **arithmetic operation** is two. If there is no pipelining, then each iteration will take six cycles to execute. That is,

Cycle	Instruction	Comments
1.	Read	Fetch A[I]
2.	MUL	Multiply by 2
3.		
4.	ADD	Add to C
5.		
6.	Write	Store A[A]

Thus, N iterations will need $6N$ cycles. Now, in our code, $N = 5$ (for loop runs 5-times)
 \therefore Cycles required = $6N = 6 \times 5 = 30$.

Please note that we are neglecting the loop control overhead.

Now, say, we want to execute the same code with software pipelining. Let the processor (PE) be a two-issue PE. Now, how many cycles are needed ?

Cycle			Iteration		
	1	2	3	4	5
1.	Read				
2.	MUL				
3.		Read			
4.		MUL			
5.	Add		Read		
6.			MUL		
7.		Add		Read	
8.	Write			MUL	
9.			Add		Read
10.		Write			MUL
11.				Add	
12.			Write		
13.					Add
14.				Write	
15.				NOP	
16.					Write

c. The software pipelined code requires only 16 cycles to execute.

$$\text{Speed-up} = \frac{30}{16} = 1.8$$

It is also possible to parallelize recurrence with software pipelining. The main feature of software pipelining is it minimizes the interval at which iterations are initiated.

Thus, the advantages of software pipelining are :

1. It reduces initiation latency and thus maximizes throughput.
 2. It produces small and compact code.

SUMMARY

Practical pipelines are not always simple because of various reasons. Some of the reasons are — Real pipelines are non-linear, different stages of a pipeline need different time periods, feedback and feed-forward connections exist and so on. A flow pattern of static pipeline is always fixed once a task is initiated. When the same stage is demanded by two or more inputs, there is a collision. So, proper job scheduling is to be done. For this some assumptions (like execution time for all pipeline stages are multiple of some basic block) are made. In case of multifunction and dynamic pipelines, path followed through the pipeline may change dramatically with each input. So, the task of defining optimal schedules becomes very difficult. This chapter solves all such problems.

MULTIPLE CHOICE QUESTIONS (MCQS) WITH ANSWERS

7. The average latency of a constant cycle is :

(a) Constant	(b) Latency itself
(c) Not known	(d) None of the above.
8. CPA stands for :

(a) Carry power adder.	(b) Carry positive addition.
(c) Carry propagation adder.	(d) None of the above.
9. CSA stands for :

(a) Carry save adder	(b) Carry storage adder
(c) Carry simple adder	(d) None of the above.
10. A separate register that has one bit to identify each register of the register file is called as a :

(a) Black board	(b) White board
(c) Score board	(d) None of the above.
11. Loop scheduling includes :

(a) Loop unrolling	(b) Software Pipelining
(c) Both (a) and (b)	(d) None of the above.
12. In out-of-order execution, the task are :

(a) in different order	(b) in same order
(c) no order	(d) None of the above.
13. A possible execution path within a loop is known as :

(a) Control path	(b) Flow path
(c) Trace	(d) None of the above.
14. During loop scheduling, when the loop size is large, the relative gain is :

(a) Large	(b) Less
(c) No change	(d) None of the above.
15. Real pipelines are :

(a) Linear	(b) Non-linear
(c) Exponential	(d) None of the above.

:: ANSWERS ::

- | | | | | |
|---------|---------|---------|---------|---------|
| 1. (b) | 2. (a) | 3. (a) | 4. (c) | 5. (b) |
| 6. (a) | 7. (b) | 8. (c) | 9. (a) | 10. (c) |
| 11. (c) | 12. (a) | 13. (c) | 14. (b) | 15. (b) |

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

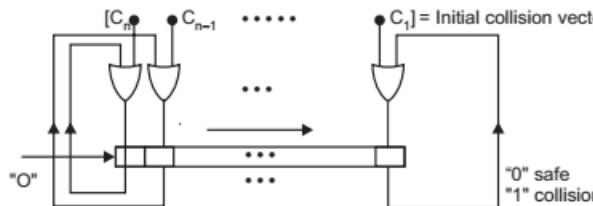
1. Suggest a possible circuitry which helps us in finding the next state of the pipeline using collision vector, C as the input. The output is either '0' meaning safe or '1' meaning collision ?

Ans. We need an n -bit right shift register where n is the maximum forbidden latency. The circuitry is shown below :

Herein, the initial collision vector (C) is initially loaded into the register. The register is then shifted to the right. Each 1-bit shift corresponds to an increase in the latency by 1. When a '0' bit emerges from the right end after P shifts, it means P is a permissible latency. Likewise, a 1-bit being shifted out means a collision and

thus the corresponding latency should be forbidden. Logical 0 enters from the left end of the shift register. The next step after P shift is thus obtained by bitwise-ORing the initial collision vector with the shifted register contents.

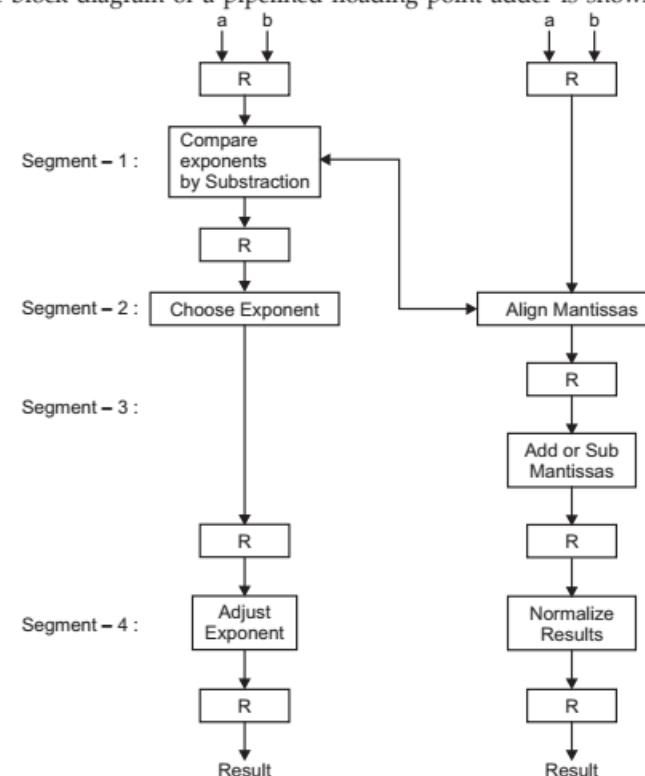
For example : From the initial state, $C_x = (1\ 0\ 1\ 1\ 0\ 1\ 0)$, the next state of $(1\ 1\ 1\ 1)$ is reached after 3 shifts or 6 shifts.



- Give the block diagram for a pipelined floating point adder. Assume that exponent matching takes 0.1 ms, mantissa alignment 0.2 ns, adding mantissas 1.0 ns and normalizing result 0.2 n sec. What will be the highest clock speed which can be used to drive the adder. If two vectors of 100 components are to be added using this adder what will be the time of addition.

[IPTU, B.Tech (CSE)-8th sem., 2007-08]

Ans. The block diagram of a pipelined floating point adder is shown next.



Given: Time of each segment is as follows:

$$\begin{aligned} t_1 &= 0.1 \text{ ns} \\ t_2 &= 0.2 \text{ ns} \end{aligned}$$

$$t_3 = 1.0 \text{ ns}$$

$$t_4 = 0.2 \text{ ns}$$

Herein, the slowest stage is the 3rd stage (of adding mantissa part). So, it will be the clock speed of adder.

Clock speed (T) = 1.0 ns

$$\begin{aligned} \therefore \text{Total time} &= t_1 + t_2 + t_3 + t_4 \\ &= 1.5 \text{ n secs.} \end{aligned}$$

and Time to execute 100 tasks

$$\begin{aligned} &= [k + (n - 1)] T \\ &= [4 + (100 - 1)] 1 \\ &= [4 + 99] \\ &= 103 \text{ ns.} \end{aligned}$$

3. How will you design a pipeline floating point adder?

[UPTU, B. Tech (CSE) 8th Sem.; 2003-04]

Ans. Let us consider the design of a pipeline floating point adder. The inputs to the pipeline are two floating point numbers in normalized form :

$$\begin{aligned} A &= a \times 2^p \\ B &= b \times 2^q \end{aligned}$$

Where a, b are two fractions and p, q are their exponents.

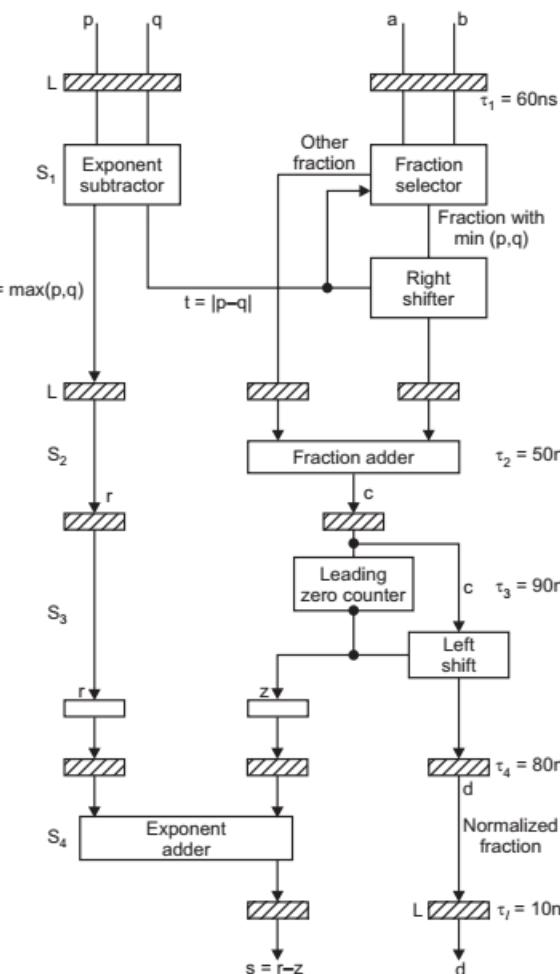
Aim is to compute sum,

$$\begin{aligned} C &= A + B \\ &= c \times 2^r \\ &= d \times 2^s \end{aligned}$$

Where $r = \max(p, q)$ and $0.5 \leq d < 1$.

This pipeline has four stages— S_1, S_2, S_3, S_4 and d is normalized (mantissa) fraction.

Operation performed by S_1 : Comparing p and q . Compare the two exponents, p and q . Get the larger exponent, $r = \max(p, q)$ and determine the difference, $t = |p - q|$.



Operation performed by S_2 : Right shifting of fraction i.e., shift right the fraction associated with lower exponent by t -bits to equalize the two exponent.

Operation performed by S_3 : Adding two fractions—Add the preshifted fraction with the other fraction and produce the intermediate sum fraction, C .

Operation performed by S_4 : Normalize the result—count the number of leading zeros, say u , in fraction C and shift left C by u -bits. The output is normalized fraction, sum, $d = c \times 2^r$. The exponent S is also modified as $(r - u)$ to produce the output exponent.

All these stages can be implemented with combinational logic circuits.

Also shown in the above Figure is that :

$$\tau_1 \text{ (delay at } S_1) = 60 \text{ ns}$$

$$\tau_2 \text{ (delay at } S_2) = 50 \text{ ns}$$

$$\tau_3 \text{ (delay at } S_3) = 90 \text{ ns}$$

$$\tau_4 \text{ (delay at } S_4) = 80 \text{ ns}$$

and τ_l (latch delay) = 10 ns

∴ The period of clock driving this pipeline will be equal to :

$$\therefore \tau = \tau_i \text{ (max)} + \tau_l$$

$$\therefore \tau = (90 + 10) \text{ ns}$$

$$\therefore \tau = 100 \text{ ns}$$

$$\Rightarrow f = \frac{1}{\tau} = \frac{1}{100 \text{ ns}} = 10 \text{ MHz.}$$

Therefore, the time taken by the pipeline to complete a task will be $T_p = 100 \text{ ns}$ and time taken by non-pipelined processor to complete a task will be :

$$\begin{aligned} T_S &= \tau_1 + \tau_2 + \tau_3 + \tau_4 + \tau_l + \tau_l \\ &= 60 + 50 + 90 + 80 + 10 + 10 \end{aligned}$$

$$\therefore T_S = 300 \text{ ns}$$

$$\begin{aligned} \therefore \text{Speed-up with pipeline adder} &= S = \frac{T_s}{T_p} \\ &= \frac{300}{100} = 3 \end{aligned}$$

Please note that in order to achieve an ideal speed up equal to the number of stages i.e., 4, all stages should possess a uniform delay of 75 ns.

$$\therefore \text{Maximum speed-up} = \frac{300}{75} = 4 \text{ (Ideal)}$$

4. Write a short note on Multifunction Arithmetic pipelines.

Ans. The pipelines like floating point adder, fixed point multiplication or division etc were unifunction pipelines as they carry out one predefined operation. However, a multi-function pipeline is designed to carry out a set of multiple operations. The basic idea is to reduce the cost of arithmetic unit by the hardware sharing. It is implemented by connecting required number of basic functional modules. These basic modules are commonly required in several different operations. For example, normalization operation is common for all floating point operations. Figure below shows a multi-function pipeline in TI-ASC computer.

The pipeline of TI-ASC computer consists of eight stages. The figure shows all the interconnections of routes among eight stages. This pipeline can perform following functions :

- (a) Fixed point arithmetic.
- (b) Floating point arithmetic.
- (c) Many logical shifting operations over scalar and vector operands.

The clock period is 60 ns. Different connecting paths are used based on different arithmetic and logic instructions.

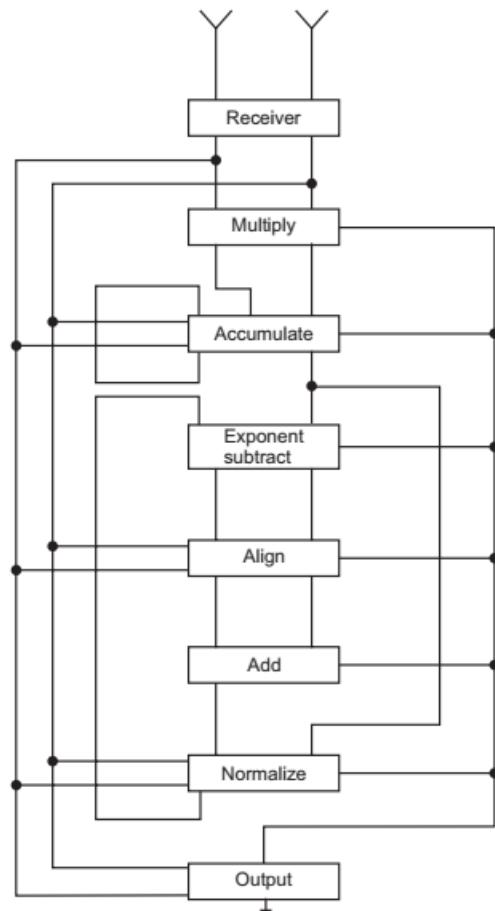


Fig. Multifunction pipeline in TI-ASC system.

5. Explain the performance of pipelines with stalls ?

Ans. A stall (wait) causes the pipeline performance to degrade from the ideal performance. Now, the speed-up from pipelining is given by a formula :

$$\begin{aligned}\text{Speed-up from pipelining} &= \frac{\text{Average instruction time pipelined}}{\text{Average instruction time unpipelined}} \\ &= \frac{\text{CPI unpipelined} \times \text{Clock period unpipelined}}{\text{CPI pipelined} \times \text{Clock period pipelined}}\end{aligned}$$

The ideal CPI on pipelined machine is always 1.

$$\begin{aligned}\therefore \text{Pipelined CPI} &= \text{Ideal CPI} + \text{Pipelined stall clock cycles/instruction} \\ &= 1 + \text{Pipeline stall clock cycles/instruction.}\end{aligned}$$

Neglecting the cycle time overhead of pipelining and assuming the stages are perfectly balanced, the cycle time of two machines are equal and

$$\text{Speed-up} = \frac{\text{CPI unpipelined}}{1 + \text{pipeline stall cycles per instruction}}$$

If all instructions take the same number of cycles, which must also equal to the number of pipeline stages (*i.e.*, the depth of pipeline) then unpipelined CPI is equal to the depth of the pipeline.

$$\therefore \text{Speed-up} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

From the above equation it is clear that, if there are no stalls (waits) then the speed up is equal to the pipeline depth *i.e.*, the number of pipeline stages.

6. Implement the following dot-product operation using internal data forwarding concept :

$$S = \sum_{i=1}^n a_i \times b_i \quad ?$$

$$\begin{aligned}\text{Ans. Now, } S &= \sum_{i=1}^n a_i b_i \\ &= a_1 b_1 + a_2 b_2 + a_3 b_3 + \dots + a_n b_n\end{aligned}$$

We consider two cases :

Case I : Without data forwarding

Without internal data forwarding between the two functional units, the three instructions must be executed sequentially in a loop structure. This is shown in Fig. (a) below.

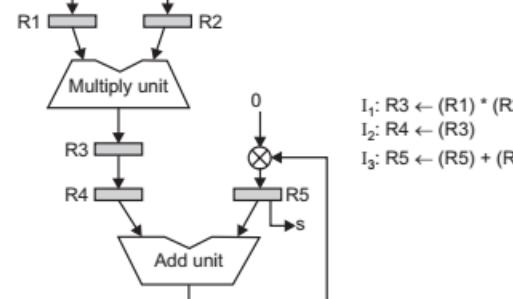


Fig. (a) Without data forwarding.

Case II : With data forwarding

With internal data forwarding, the output of the multiplier is fed directly into the input register, R4, of the adder. At same time, the output of the multiplier is also routed to register, R3. Thus, we can say that **internal data forwarding between the two functional units reduces the total execution time through the pipelined processor**. This is shown below in Fig. (b).

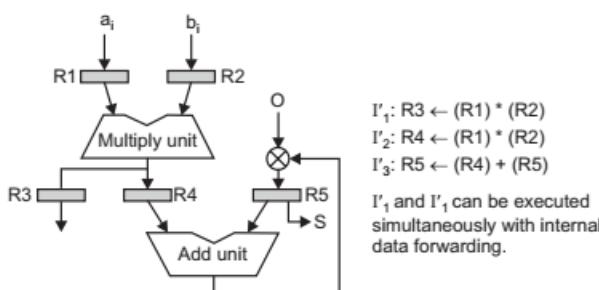


Fig. (b) With internal data forwarding.

7. Compare the performance of the following :
- Superscalar
 - Superpipelined
 - Superpipelined superscalar ? [UPTU, B. Tech (CSE) 8th Sem.; 2007-08]

Ans. Let us assume that :

N — Independent instructions through the pipeline.

n — Degree of super pipelined processor

K — Stages in pipeline

(m, n) — Degree of superpipelined-superscalar machine

Then, we compare the three in the tabular form as follows :

Superscalar	Super pipelined	Super pipelined Superscalar
<p>1. The time required by the scalar base machine is $T(1, 1) = (K + N - 1) \dots (1)$</p> <p>But the ideal execution time required by an m-issue superscalar machine is :</p> $T(m, 1) = K + \frac{N-m}{m} \dots (2)$ <p style="text-align: center;">(basecycles)</p>	<p>1. The time required by the scalar base machine is $T(1, 1) = (K + N - 1) \dots (1)$</p> <p>And the minimum time required to execute N instructions for a superpipelined machine of degree, n with k stages in the pipeline is :</p> $T(1, n) = K + \frac{1}{n}(N-1) \dots (2)$	<p>1. The time required by the scalar base machine is $T(1, 1) = (K + N - 1) \dots (1)$</p> <p>(base cycles)</p> <p>And the minimum time needed to execute N independent instructions on a super pipelined superscalar machine of degree (m, n) is</p> $T(m, n) = K + \frac{N-m}{mn} \dots (2)$

<p>Where k is the time required to execute first m instructions through the m pipelines simultaneously. The second term shows the time required to execute remaining $(N - m)$ instructions, m per cycle.</p> <p>\therefore Ideal speed-up, $S(m, 1)$</p> $\begin{aligned} &= \frac{T(1,1)}{T(m,1)} \\ &= \frac{N+k-1}{N/m+k-1} \\ &= \frac{m(N+k-1)}{N+m(k-1)} \end{aligned}$ <p>(From equation (1) and (2)) As $N \rightarrow \infty$, the speed-up limit, $S(m, 1) \rightarrow m$</p>	<p>\therefore Speed-up,</p> $\begin{aligned} S(1, n) &= \frac{T(1,1)}{T(1,n)} \\ &= \frac{K+N-1}{K+(N-1)/n} \\ &= \frac{n(K+N-1)}{nK+N-1} \end{aligned}$ <p>As $N \rightarrow \infty$, $S(1, n) \rightarrow n$</p>	<p>\therefore Speed-up, over base machine is :</p> $\begin{aligned} S(m, n) &= \frac{T(1,1)}{T(m,n)} \\ &= \frac{K+N-1}{K+(N-m)/mn} \\ &= \frac{mn(K+N-1)}{mnK+N-m} \end{aligned}$ <p>As $N \rightarrow \infty$, $S(m, n) \rightarrow mn$</p>
2. They use spatial transistors.	2. They use temporal parallelism.	2. They use combined parallelism.
3. It requires more transistors.	3. It requires faster transistors.	3. It requires more and faster transistors.
4. It has more performance than superpipelined machine.	4. It has lower performance than superscalar machine.	4. Design trade offs do exist in regard to choices of (m, n) .

8. Consider the following reservation table for four stage pipeline with a clock cycle, $\tau = 20$ ns.

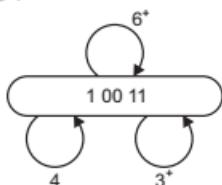
- (a) What are the forbidden latencies and initial collision vector ?
- (b) Draw the state transition diagram for scheduling the pipeline.
- (c) Determine MAL.
- (d) Determine pipeline throughout.
- (e) Determine the lower bound on the MAL for this pipeline. Have you obtained the optimal latency from the above state diagram ?

Ans. (a) Forbidden latencies are 1, 2 and 5.

\therefore Initial collision vector (ICV) = 10011

(b) Its STD is as follows :

	1	2	3	4	5	6
S_1	X					X
S_2		X		X		
S_3			X			
S_4				X	X	



9. Consider the following pipeline reservation table :

- What are the forbidden latencies ?
- Draw a STD.
- List all the simple and greedy cycles.
- Determine the optimal constant latency cycle and the minimal average latency.
- Let the pipeline clock period be $\tau = 20$ ns. Determine the throughput of this pipeline.

Ans. (a) Forbidden latency : 3

Collision Vector : (100)

(b) State transition diagram (STD) is given below :

(c) Simple cycles are :

(2), (4), (1, 4), (1, 1, 4) and (2, 4)

Greedy cycles are :

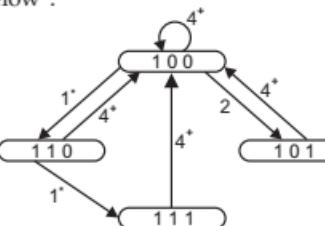
(2) and (1, 1, 4)

(d) Optimal constant latency cycle : (2)

MAL = 2

$$(e) \text{Throughput} = \frac{1}{2\tau} = \frac{1}{2 \times 20 \text{ ns}} = 25 \text{ MOPS}$$

	1	2	3	4
S ₁	X			X
S ₂		X		
S ₃			X	



10. The time delays in the four segments in a pipeline are as follows:

$$t_1 = 50 \text{ ns}$$

$$t_2 = 30 \text{ ns}$$

$$t_3 = 95 \text{ ns}$$

$$t_4 = 45 \text{ ns}$$

The interface registers delay time, $t_r = 5$ ns.

(a) How long would it take to add 100 pairs of numbers in the pipeline?

(b) How can we reduce the total time to about one-half of the time calculated in part-a?

Ans. (a) Time taken to add n -pairs of numbers in k -segment pipeline

$$= (k + n - 1) t_p \text{ a} \quad \text{Now, } k = 4 \text{ and } n = 100 \text{ (given)}$$

Let us now calculate the clock cycle, t_p :

$$\text{Total time delay in a segment-}i \ (TD_i) = \begin{pmatrix} \text{time delay} \\ \text{in segment} \\ \text{circuit} \end{pmatrix} + \begin{pmatrix} \text{time delay} \\ \text{of interface} \\ \text{register} \end{pmatrix}$$

= Maximum time delay of all segments.

$$\therefore TD_{\max} = \begin{pmatrix} \text{Maximum time} \\ \text{delay in segment} \\ \text{circuits} \end{pmatrix} + \begin{pmatrix} \text{time delay} \\ \text{of the interface} \\ \text{register} \end{pmatrix}$$

$$= 95 \text{ ns} + 5 \text{ ns}$$

$$= 100 \text{ ns.}$$

But time delay in each segment is different. So, the clock cycle must be larger than or equal to the maximum delay.

Clock cycle, $t_p = 100$ ns

Time taken to add 100 pairs of numbers in the 4-segment

$$\begin{aligned}\text{pipeline} &= (k + n - 1) t_p \\ &= (4 + 100 - 1) 100 \text{ ns} \\ &= 10.3 \text{ ns.}\end{aligned}$$

- (b) To reduce the time calculated in part-a to one half, we have to reduce the clock cycles, t_p . This can be achieved if we could reduce the maximum time delay, TD_{\max} to 50 ns. This can be reached by reducing 1t from 50 ns to 45 ns and 3t from 95 ns to 45 ns.

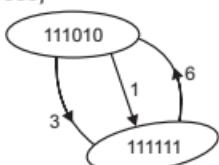
11. Determine the Minimum Average Latency (MAL) and through put of the following Instruction pipelining shown in the reservation table.

Reservation Table

S_1	X			X		
S_2		X				X
S_3		X		X		X
S_4		X				X
S_5	X		X			X
	t_1	t_2	t_3	t_4	t_5	t_6
						t_7

Ans. Forbidden list $F = \{2, 4, 5, 6\}$

Collision vector, $C = \{010111\}$



Simple Cycle Average Latency

$$1,6 \quad \frac{7}{2} = 3.5$$

$$3, 6 \quad \frac{9}{2} = 4.5$$

Therefore, minimum avg. latency = 3.5. Ans.

$$\text{Through put} = \frac{1}{3.5} \times 100 = 33.3\%. \text{ Ans.}$$

12. Discuss the hazard and hazards detentions in the memory system used in pipelines with example.

Ans. HAZARD: Pipeline hazard means the situation that prevents the next instruction in the instruction stream from executing during its clock cycle. Three types of hazards can occur:

1. Data Hazard: This arises when an instruction depends on the results of a previous instruction in a way that it is exposed by the overlapping of instructions in the pipeline. The data hazard may be classified into three more categories:

- (a) RAW Hazard (Read After Write): When the instruction j attempts to read some object that is being modified by the instruction.

- (b) **WAR Hazard (Write After Read):** When the instruction j attempts to write into object that is read by i .
- (c) **WAW Hazard (Write After Write Hazard):** When j attempts to write onto some object that it modified by i .

The data hazard can be resolved by *register lagging, data forward*.

2. **Structural Hazard:** This type of hazard occur when the hardware could not support all possible combinations of the instructions simultaneously.

Two Techniques to Resolve:

- Duplicacy of resources
- Proper pipelining.

3. **Control Hazard:** Control hazard arise for the pipelining of branches and other instructions that change the program counter status control hazard can cause a greater performance loss in pipelining. The simplest technique to resolve this is to *freeze and flesh* the pipeline.

Four Methods to Resolve:

- (1) Branch Elimination. (2) Branch Speed up.
- (3) Branch Prediction (4) Branch Target capture.

13. (a) Show the parallel addition in a hypercube and 2D mesh.

(UPTU, B.Tech (CSE) 8th sem; 2005-06)

- Ans.** Assume A and B are the input matrices and C be a result matrix. The algorithm for matrix multiplication is:

DO $I = I, N$

DO $J = I, N$

DO $K = I, N$

$$C(I, J) = C(I, J) + A(I, K) * B(K, J)$$

The desired parallel algorithm can be written as, For all PE's

$C = 0$.

$P_O I$ ($K = 1, 4$)

$C = C + A * B$

Call Rotate Left (A)

Call Rotate up (B)

Continue

End.

Step A : (a) Inner Product :

a_{11}	b_{11}	a_{21}	b_{12}	a_{31}	b_{13}	a_{41}	b_{14}
a_{12}	b_{21}	a_{22}	b_{22}	a_{32}	b_{23}	a_{42}	b_{24}
a_{13}	b_{31}	a_{23}	b_{32}	a_{33}	b_{33}	a_{43}	b_{34}
a_{14}	b_{41}	a_{24}	b_{42}	a_{34}	b_{43}	a_{44}	b_{44}
Result C_{11}		C_{22}		C_{33}		C_{44}	

Step B (1): Rotate Columns of B left by 1 position:

$a_{11} b_{12}$	$a_{21} b_{13}$	$a_{31} b_{14}$	$a_{41} b_{11}$
$a_{12} b_{22}$	$a_{22} b_{23}$	$a_{32} b_{24}$	$a_{42} b_{21}$
$a_{13} b_{32}$	$a_{23} b_{33}$	$a_{33} b_{34}$	$a_{43} b_{31}$
$a_{14} b_{42}$	$a_{24} b_{43}$	$a_{34} b_{44}$	$a_{44} b_{41}$
Result C_{12}		C_{23}	C_{34}
			C_{41}

Step B (2): Rotate Columns of B left by 1 position:

$a_{11} b_{13}$	$a_{21} b_{14}$	$a_{31} b_{11}$	$a_{41} b_{12}$
$a_{12} b_{23}$	$a_{22} b_{24}$	$a_{32} b_{21}$	$a_{42} b_{22}$
$a_{13} b_{33}$	$a_{23} b_{34}$	$a_{33} b_{31}$	$a_{43} b_{32}$
$a_{14} b_{43}$	$a_{24} b_{44}$	$a_{34} b_{41}$	$a_{44} b_{44}$
Result C_{13}		C_{24}	C_{31}
			C_{42}

Step B (3): Rotate Columns of B left by 1 position:

$a_{11} b_{14}$	$a_{21} b_{11}$	$a_{31} b_{12}$	$a_{41} b_{13}$
$a_{12} b_{24}$	$a_{22} b_{21}$	$a_{32} b_{22}$	$a_{42} b_{23}$
$a_{13} b_{34}$	$a_{23} b_{31}$	$a_{33} b_{32}$	$a_{43} b_{33}$
$a_{14} b_{44}$	$a_{24} b_{41}$	$a_{34} b_{42}$	$a_{44} b_{43}$
Result C_{14}		C_{23}	C_{32}
			C_{43}

Complexity $O(n^3)$ **Hypercube SIMD Model**

If the PRAM processor interaction pattern form a graph that embed with dilation 1 in a target SIMD architecture, then there is a natural translation from the PRAM algorithm combine values in a binomial tree pattern. The hypercube processor array version follows directly from the PRAM algorithm. The only significant difference is that the hypercube processor array model has no shared memory : processors interact by passing data. The resulting algorithm appears in figure. Figure illustrates this algorithm.

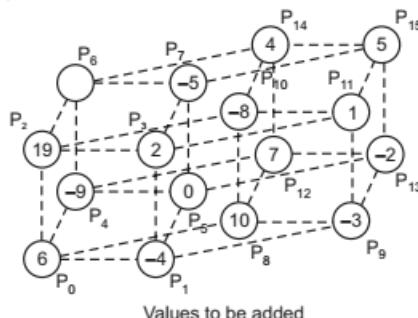
To computer the worst-case time complexity of this algorithm, we determine the number of computation steps and the number of communication steps. Each processing element adds at most $[n/p]$ values to find its local sum. Processor 0, which iterates the second inner for loop more than any other processor, performs $\log p$ communications steps and $\log p$ addition steps. The complexity of finding the sum of n values is $\Theta(n/p + \log p)$ using the hypercube processor array model with p processors.

What if we want every processing element to have a copy of the global sum? One way to accomplish this would be to add a broadcast phase to the end of the algorithm we have already developed once processing element 0 has the global sum, the value can be transmitted to the other processors in $\log p$ communication steps by reversing the direction of the edges in the binomial tree illustrated in figure.

Summation (Hypercube SIMD):

```

Parameter   n  (Number of elements to add)
            p  (Number of processing elements)
Global     j
Local      local.set.size, local.value[1...[n/p]], sum, tmp.
begin
    for all  $P_p$  where  $0 \leq i \leq p - 1$  do
        if  $i \leftarrow (n \text{ modulo } p)$  then
            local.set.size  $\leftarrow [n/p]$ 
        else
            local.set.size  $\leftarrow [n/p]$ 
        else
            local.set.size  $\leftarrow [n/p]$ 
        endif
        sum  $\leftarrow 0$ 
    endfor
    for  $j \leftarrow 1$  to  $[n/p]$  do
        for all  $P_p$  where  $0 \leq i \leq p - 1$  do
            if local.set.size  $\geq j$  then
                sum  $\leftarrow sum + local.value[j]$ 
            endif
        endfor
        for  $j \leftarrow log p - 1$  down to 0 do
            for all  $P_p$  where  $0 \leq i \leq p - 1$  do
                if  $i < 2^j$  then
                    tmp  $\leftarrow [i + 2^j]sum$ 
                    sum  $\leftarrow sum + tmp$ 
                endif
            endfor
        endfor
    endfor
end.
```



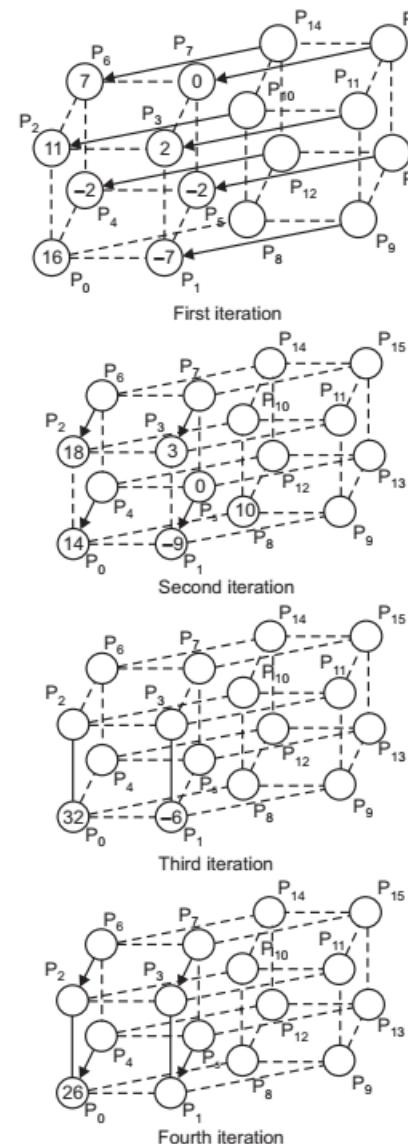


Fig. Parallel summation on the hypercube SIMD model.

Summation (2-D Mesh SIMD):

```

Parameter    $l$   (Mesh has size  $l \times l$ )
Global       $i$ 
Local        $tmp, sum$ 
begin

```

```

[Each processor finds sum of its local values – code not shown]
for all  $i \leftarrow l - 1$  downto 1 do
    for all  $P_{j,i}$ , where  $1 \leq j \leq i$  do
        [Processing elements in column  $i$  active]
         $tmp \Leftarrow east\ (sum)$ 
         $sum \leftarrow sum + temp$ 
    endfor
endfor
for  $i \leftarrow l - 1$  downto 1 do
    for all  $P_{i,1}$ , do
        [Only a single processing element active]
         $tmp \Leftarrow south\ (sum)$ 
         $sum \leftarrow sum + tmp$ 
    endfor
endfor
end

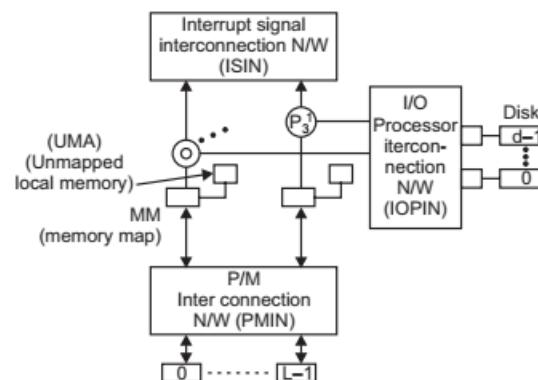
```

Fig. Parallel summation on a processor array organized as a 2-D mesh. Upon termination of algorithm, variable [1, 1] sum contains the global sum.

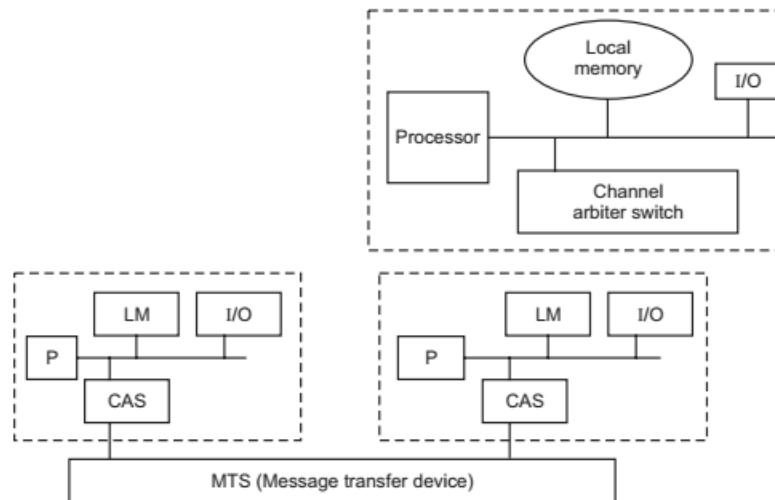
14. (b) Discuss the tightly coupled and loosely coupled multiprocessor architecture with block diagrams.

Ans. (b) **Tightly Coupled and Loosely Coupled Multiprocessor :** A multiprocessor system is controlled by one operating system which provide interaction between processor and their program at the process, data set and data element levels.

1. **Tightly Coupled Multiprocessors:** These multiprocessor communicate through a shared memory. It is consist of P processor, l memory modules and k input output channel. Three units connected through interconnection network PMIN, IOPIN, ISIN.



- 2. Loosely Coupled Multiprocessors:** In such systems each processor has a set of input output devices and a large local memory where it executes most of the data and instructions.



EXERCISE QUESTIONS

1. Consider a five stage pipelined processor specified by following reservation table:

	1	2	3	4	5	6
S ₁	X					X
S ₂		X			X	
S ₃			X			
S ₄				X		
S ₅		X				X

- (a) List the set of forbidden latencies and the collision vector.
- (b) Draw the state transition diagram.
- (c) List all simple cycles from state diagram.
- (d) Identify the greedy cycles among the simple cycles.
- (e) Find out minimum average latency ?
- (f) Find out maximum throughout of this pipeline ?

[UPTU, B. Tech (CSE) 8th Sem.; 2007-08]

[GGSIPU, M.Tech 1st sem., Dec. 2003 and Dec., 2004]

and [GGSIPU, M-Tech. (CSE/IT) -1st sem., Dec. 2011]

[Hint : Forbidden Latency for $s_1 = \{5\}$

Forbidden Latency for $s_2 = \{3\}$

Forbidden Latency for $s_3 = \{\phi\}$

Forbidden Latency for $s_4 = \{\phi\}$

Forbidden Latency for $s_5 = \{4\}$

\therefore Forbidden Latency for set = {3,4,5}

\therefore Collision vector, C = $[d_5 \ d_4 \ d_3 \ d_2 \ d_1] = [11100]$

Transition diagram is given

\therefore Simple cycle for path 1 is

$$s_1 = \{1, 2, 3, 9, 10, 11, \\ 17, \dots\}$$

Greedy cycle = {1,1,6}

$$\therefore \text{MAL} = 8/3 = 2.66$$

Also, simple cycle for path 2 is

$$s_2 = \{1, 7, 16, 19\}$$

Greedy cycle for path 2 is

$$= \{6\}$$

$$\therefore \text{MAL} = 6/1 = 6$$

Similarly, simple cycle for path 3 is

$$s_3 = \{1, 2, 8, 9, 15, \dots\}$$

Greedy cycle for path 3 is {1,6}

$$\therefore \text{MAL} = 7/2 = 3.5$$

$$\therefore \text{MAL is min}(2.66, 6, 3.5) = \\ 2.66$$

$$\& \text{Max. put through} = \frac{1}{\text{MAL}}$$

$$= 3/8$$

2. Three functional pipelines f_1, f_2 and f_3 are characterised by the following reservation tables.

$f_1:$

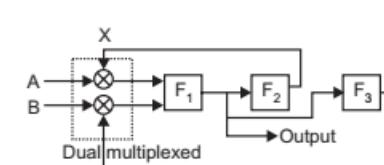
	1	2	3	4
S_1	X			
S_2		X		
S_3			X	X

$f_2:$

	1	2	3	4
T_1	X			X
T_2		X		
T_3			X	

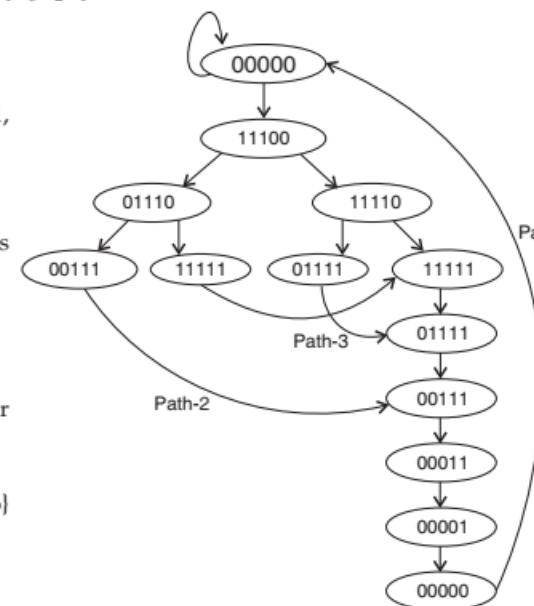
$f_3:$

	1	2	3	4
U_1	X		X	
U_2				X
U_3	X			



- (a) Complete the composite reservation table for the composite pipelines with 9 rows ($S_1, S_2, S_3, T_1, T_2, T_3, U_1, U_2, U_3$) and 12 columns (1, 2, ..., 12).

- (b) Write the forbidden list and the initial collision vector.



- (c) Draw a state diagram clearly showing all latency cycles.
 - (d) List all simple cycles and greedy cycles.
 - (e) Calculate MAL and maximal throughout of this composite pipeline

[Hints]

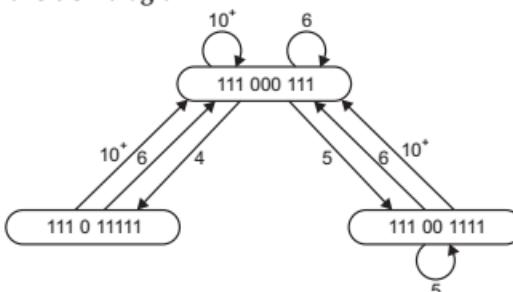
- (a) The complete reservation table is as follows :

	1	2	3	4	5	6	7	8	9	10	11	12
S1	X								X			
S2		X								X		
S3			X	X							X	X
T1				X				X				
T2					X							
T3						X						
U1				X		X						
U2							X					
U3						X						

- (b) **Forbidden latencies** : 8, 1, 7, 9, 3, 2.

So, collision vector : (111000111).

- (c) State transition diagram :**



- (d) Simple cycles are

(5), (6), (10), (4, 6), (4, 10), (5, 6) and (5, 10).

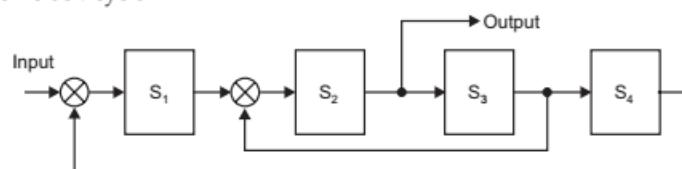
Greedy cycles are found from simple cycles :

(5) and (4, 6)

(e) MAL = 5

(f) Maximum throughput = $1/(5\tau)$

3. Consider the following pipeline processor with four stages. This pipeline has a total evaluation time of six clock cycles. All successor stages must be used after each clock cycle :



- (a) Draw its reservation table.
- (b) List set of forbidden latencies.
- (c) Draw the state transition diagrams (STD).
- (d) List all greedy cycles from STD.
- (e) What is the value of the minimal average latency ?
- (f) What is the maximal throughput of this pipeline ?

[GGSIPU, M.Tech., 1st sem., 1st minor test – 2004]

[Hints :

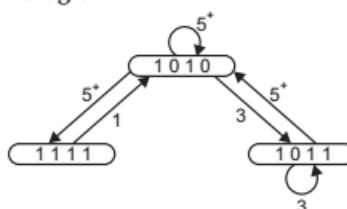
- (a) Reservation table :

	1	2	3	4	5	6
S ₁	X				X	
S ₂		X		X		X
S ₃			X		X	
S ₄				X		

- (b) Forbidden latency : 2, 4

Collision vector : (1010)

- (c) State transition diagram :



- (d) Simple cycles are :

(3), (5), (1, 5) and (3, 5)

- (e) Greedy cycles are :

(1, 5) and (3).

- (f) MAL = 3

(g) Maximum throughput = $1/(3\tau)$.

4. For synchronous model (pipelines) derive the formulas for calculating clock skewing, speed-up, efficiency, throughput and optimal number of pipeline stages. What is performance/cost ratio ? [GGSIPU, B.Tech., 7th sem., Dec. 2007]
5. (a) Explain how the throughput in a pipelined processor increases compared to a non-pipelined processor.
 (b) Prove that a k -stage linear pipeline can be at most k -times faster than that of a non-pipelined serial processor. [KUD, B.E. (CSE), 8th sem., 1996 & 1997]
6. Show that the maximum speed-up of a pipeline is equal to its stages. [KUD, B.E. (CSE), 8th sem., 1997]
7. Given a task that can be divided into m subtasks, each requiring 1 unit of time, how much time is required for an m -stage pipeline to process n -tasks ?

[KUD, B.E. (CSE), 8th sem., 1995]

8. (a) Draw a five-stage multiply pipeline for multiplying binary integers. The first stage gives partial results. The last stage is a 36-bits carry-lookahead adder. The middle three stages are made of 16 CSAs.
(b) If $\tau_1 = \tau_2 = \tau_3 = \tau_4 = 70$ ns and $\tau_5 = 45$ ns
Also $\tau_l = 20$ ns (latch delay). Find the maximal clock rate of the pipeline.
(c) What is the maximal throughput of this pipeline in terms of the number of 36-bit results generated per second ?

[UPTU, B. Tech. (CSE) 8th Sem.; 2003-04]

9. How scoreboarding technique improves the performance of a pipeline ?
[Pune Univ., B.E. 2nd sem., Dec. 2006]
10. Discuss the following terms associated with pipelining :
(a) Register tagging
(b) Internal forwarding. [Pune Univ., B.E. 2nd sem., Dec. 2006]
11. Explain the principle of pipeline with the help of pipeline for floating point adder.
[Pune Univ., B.E. 2nd sem., May, 2005]
12. Write a short note on – Hazards in pipeline processor.

[UPTU, B. Tech (CSE) 8th Sem.; 2004-05][Pune Univ., B.E. 2nd sem., Dec. 2003, Dec. 2005]

13. What is meant by pipeline, superscalar and superpipeline processor ? What are the various factors placing constraints on new start of the pipeline processor ?
[Pune Univ., B.E. 2nd sem., May, 2006]
14. Draw and explain pipeline unit for fixed-point multiplication of 8 bit integers.
[Pune Univ., B.E. 2nd sem., May, 2006]
15. For a k -stage pipelined processor, let ' p ' be the probability of a conditional branch instruction in typical instruction stream of ' N ' instructions and ' a ' is the probability of a successfully executed conditional branch instruction. Let ' b ' be the delay slot and ' t ' is the clock period. Find out effective pipeline throughput and performance degradation factors.
[Pune Univ., B.E. 2nd sem., May, 2002]

16. Derive speed-up, efficiency and throughput for k -stages and n -tasks in a linear pipeline. Show the variation of PCR with different stages.
[UPTU, B. Tech. (CSE) 8th Sem.; 2004-05]
17. What do you understand by pipelining? Explain it. What are the hazards that occur in pipelining in your opinion. Explain it.
[UPTU, B. Tech. (CSE) 8th Sem.; 2008-09]

18. What do you understand by linear and non-linear pipeline processes. Explain them
[UPTU, B. Tech. (CSE) 8th Sem.; 2008-09]
19. Define a trace. How is it used in a VLIW processor? List the advantages and disadvantages of VLIW processor.
[GGSIPU, B.Tech (CSE) -7th sem., Dec. 2011]
20. Draw the space diagram for a 6 segment pipeline showing the time it takes to process 8-tasks.
[GGSIPU, B.Tech (CSE) -7th sem., Dec. 2011]



8 VECTOR PROCESSING

8.0 INTRODUCTION

If we recall our convolution problem which is stated as—"Given a sequence of weights and sequence of input values as $\{W_1, W_2, W_3, \dots, W_k\}$ and $\{X_1, X_2, X_3, \dots, X_n\}$ respectively, to compute an output sequence $\{Y_1, Y_2, \dots, Y_{n+1-k}\}$ " and is defined as :

$$Y_1 = W_1 X_i + W_2 X_{i+1} + \dots + W_k X_{i+k-1}$$

Herein, the two different vectors are :

$$W = \{W_1, W_2, W_3, \dots, W_k\}$$

$$\& \quad X = \{X_1, X_2, X_3, \dots, X_n\}$$

Now, this calculation can be executed in different ways as given below :

Way 1 : In an ordinary uniprocessor, this computation would have to be performed in a sequential manner.

For example : it's machine code equivalent program fragment can be :

```

y := 0
j := 1
for i := 1 to k
    y := y + W[i] * x[j++]
```

Herein, the inner product computation exhibits natural parallelism. This means that the products $W_i \times X_j$ can be calculated independently and in parallel before they are summed.

Please note that the summation of all products exhibits natural parallelism and can take the advantage of its intrinsic parallelism. That is,

$$W_1 X_1 + W_2 X_2 + W_3 X_3 + \dots$$

Here, adding of the product pairs is done in parallel. The basic characteristics of the inner product and other similar computations is that some sort of an identical operation (like multiplication, division, addition etc) is required to be performed repeatedly on independent set of operands.

For example : pairs of vector elements $W[i], X[j]$ for $i = 1, \dots, k$ and $j = 1, \dots, n$.

Also note that if the repetitive operations can be decomposed and organized linearly as a set of tasks or phases, then pipelining may be used to overlap the processing of the independent sets of operands.

Way 2 : Now consider a pipeline drawn in Fig. 8.1 below. It consists on n-stages.

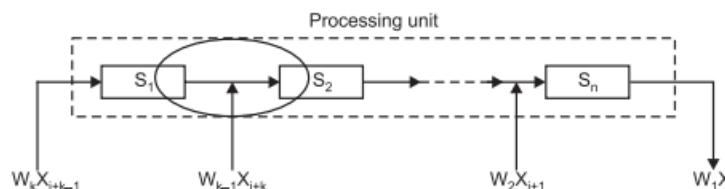


Fig. 8.1. Pipeline vector Processing.

This pipeline would be fed at the input side with the operand sets after fixed interval of time. At any given point of time, the different stages would be executing different phases of the operation on different sets of operands. Computers working on this principle are known as pipelined vector processors or pipelined array processors.

Way 3 : When repetitive operations can be performed by independent processing units which operate on an independent set of operands concurrently. Such a computer which works on this principle, is known as a parallel array processor or SIMD machines and is shown in Fig. 8.2 below :

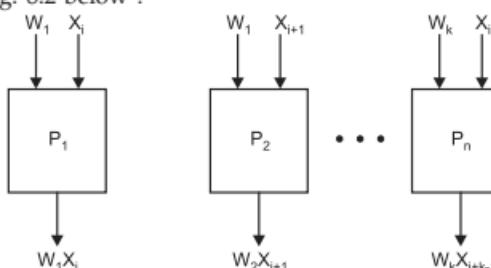


Fig. 8.2. Parallel Array processing.

Where : $P_1, P_2 \dots P_n$ are Processing units.

So, we can now say, that in this chapter we will deal with second-way while the next chapter discusses way-3.

A supercomputer is a computer that has a sustained performance rate, of at least 100 M Flops and a maximum rate of 1000 M Flops or more.

A vector processor is used for high performance scientific computing where the matrix and vector operations are common.

Some examples are :

- | | |
|--------------------------|-------------------------------|
| (a) Weather forecasting. | (b) Space flight simulations. |
| (c) Image processing. | (d) Remote sensing. |

For example : Cray Y-MP and Convex-C380 are vector processors used today.

8.1 COMPARISON OF VECTOR AND ARRAY PROCESSORS

We discuss some similarities first. They are as follows :

1. The **basic principle** of vector and array processor is that, only one instruction stream is there, that is monitored by a single control unit. And also a single program counter (PC) is processed by the system.
2. They are both member groups of the supercomputer class and parallelism in them is of inter-instruction type.

3. They are both designed to execute a common class of computation problems, involving arrays of data or vectors.
 4. Both use data parallelism.
- Let us see some differences now.

Vector Processor	Array Processor
<ol style="list-style-type: none"> 1. Parallelism is achieved by a multiplicity of concurrently executable, distinct and specialized functional units that collectively constitute the overall ALU in the processor. 2. Parallelism is also achieved by pipelining the individual functional units. 3. Data communication is achieved through shared memory and shared registers. 4. The functional units do not communicate with one another. 5. Only single stream of data traffic at the processor-memory interface. 	<ol style="list-style-type: none"> 1. Parallelism is achieved by a multiplicity of processing elements (PEs), each of which constitutes a distinct ALU in its own right. 2. Several concurrent streams of data traffic are at the processor–memory interface. So, parallelism is achieved. 3. Data communication is achieved through shared memory or through direct transmission between the PEs. 4. The PEs can communicate with one another directly. 5. Several concurrent streams of data traffic are at the processor–memory interface.

8.2 BASIC VECTOR ARCHITECTURE AND IT'S CLASSIFICATION

Architecturally, a typical vector processor consists of multiple functional pipes. A parallel task-scheduling model is presented for multi-pipeline vector processors. The functional diagram of modern multiple-pipeline vector computer is shown in Fig. 8.3

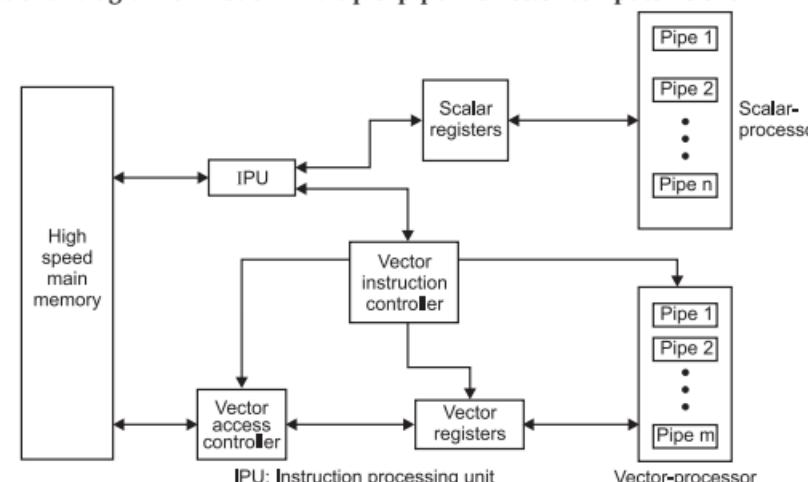


Fig. 8.3. A typical vector processor block diagram.

It's main features are as follows :

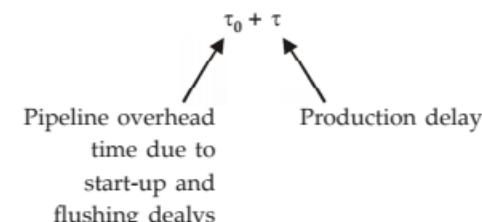
1. On the extreme left-hand side, we find that there is a main memory block which is often interleaved to minimize access time of vector operands.

2. The IPU fetches and decodes instructions. The instructions and data may be in vector or scalar format.
3. All scalar instructions are transferred to the scalar processor for execution where scalar processor itself contains multiple scalar pipelines.
4. All vector instructions are transferred to vector instructions controller by IPU, after recognizing it.
5. A set of vector instructions are placed in a task system with a precedence relation determined by data dependencies.
6. A long vector task can be partitioned into many subvectors to be processed by different pipelines concurrently.
7. The functions of vector instruction controller are :
 - Decoding vector instructions.
 - Calculating effective vector-operand addresses.
 - Setting up the vector processor.
 - Monitoring the execution of vector instructions.
8. Vector instruction controller can partition a vector task and schedule different instructions to different functional pipelines
9. Vector access controller fetches vector operands by a series of main memory accesses.
10. The vector registers are used to fill up the gap between the main memory and the vector processors.

How vector tasks are scheduled ?

The answer is here. The vector instructions are scheduled by the vector instruction controller. It is capable of scheduling several vector instructions simultaneously.

\therefore The time required to complete the execution of single vector task is measured by the expression :



Here, $\tau = t_1 \cdot L$

Where t_1 : Average latency between two successive operand pairs.

L : Vector length

Note :

1. The **start-up time** is measured from the initiation of the vector instruction to the entrance of the first operand pair into the pipeline.
2. With different vector instructions, the t_0 and t_1 value changes.

The overhead time, t_0 , may vary from tens to several hundreds of pipeline cycles. The latency, t_1 is usually one or two pipeline cycles. Thus, it is assumed that $t_0 \gg t_1$.

Vector processor classification : Depending on the vector formation and from where these are available for processing by the pipeline, we classify vector processors as :

1. Memory-to-memory architecture.
2. Register-to-register architecture.

I. Memory-to-memory architecture

Some main features of this architecture are as follows :

- (a) The source operands, intermediate and final results are retrieved directly from the main memory.
- (b) The information regarding the base address, the offset, the increment and vector length must be specified in the vector-instructions in order to enable streams of data transfer between the main memory and pipelines.
- (c) A vector stream unit replaces the vector registers. (Vector registers are in fact used in second category discussed after this).
- (d) In cyber-205 vector machine, vector operands of size 512 bits (super words) are directly retrieved from the main memory to fill pipelines and vice versa.
- (e) The start up time for vector pipelines is more because it takes more time to start a vector operation due to increased memory access time.

Examples of vector super computers of this category are : TI-ASC, CDC, Cyber 205 etc.

II. Register-to Register architecture

Some main features of this architecture are as follows :

- (a) In this class, operands and results are retrieved indirectly from the main memory through the use of a large number of vector or scalar registers.
- (b) The vector functional pipelines retrieve operands from and puts the results into the vector registers.
- (c) All vector registers are programmable in user instructions.
- (d) The length of each vector register is usually fixed. Say, for example, 64-bit in Cray supercomputer.
- (e) This architecture is very popular in vector supercomputer area.

Examples of vector supercomputers of this category are : Cray-1, Fujitsu VP-2000 series.

Which of the two classes mentioned above is better ?

Register-to-register vector processors (class-2) are usually more efficient than class-1 vector processors. This is so because in class-2 vector processors, there is reduced vector access time and there is an overlap between store vector and other operations. On the other hand, memory-to-memory vector processors (class-1) are useful if the vectors are extremely long.

What is the difference between scalar and vector processing ?

Consider the following C-code-

```
for (i = 0 ; i < n ; i++)
    A [i] = B[i] + C[i];
```

Two cases arise :

Case 1 : Scalar processor implementation of above code.

The scalar processing of the above code is as follows :

Initialize i = 1

```

L1 : Read B[i] , C[i]
Add B[i] + C[i]
A [i] ← B [i] + C[i] // Store
    i ← i + 1
If (i <= n) go to L1
else
    break;

```

In this case, the scalar loop is to be repeated N-times.

Case 2 : Vector processor implementation of the above code

The for-loop operations can be vectorized as follows :

$$A[1:n] = B[1:n] + C[1:n]$$

Where 'n' is array size.

In this case, the instruction is to be fetched only once.

Please note that an optimized object code need to be produced in order to maximize the pipeline utilization. Various methods of enhancing vector processing capability are as follows :

1. Enrich the vector instruction set to avoid excessive memory accesses. For example: Use compress vector instruction.
2. Combine or group the scalar instructions of same type together as a batch rather than interleaving them.
3. Choose suitable algorithms. Please note here that faster algorithms used in conventional serial processors may not be at all effective here.

For example : A merge-sort algorithm will be a better choice for pipelining as the machine can merge two ordered vectors in one pass only.

4. Use a **vectorizing compiler**. This compiler must be developed to detect the concurrency among vector instructions. These compilers would regenerate parallelism lost in the use of sequential languages. The process to replace a block of sequential code by the vector instructions is called as vectorization and the system software which does this parallelism is called as a vectorizing compiler.

8.3 VECTOR PROCESSING RELATED TERMINOLOGY

Some terms are very important here and are thus given below :

1. **Vector** : A vector is a set of scalar data items, all of the same type, stored in memory.
For example : A vector array, A is shown here. $A[1:N] = B[1:N] + C[1:N]$
2. **Stride (or skew distance)** : The vector elements are ordered to have a fixed addressing increment between successive elements called as stride or skew distance. The value of stride could be different for different variables.
For example : When a vector is loaded into a vector register, then the stride/ skew distance is 1 (one) i.e., all the elements of a vector are adjacent.
3. **Vector processor** : It is an ensemble of hardware resources including vector registers, functional pipelines, PEs and register counters, for performing vector operations.
4. **Vector processing** : It occurs when arithmetic or logical operations are applied to vectors.

5. Vectorization : The process of conversion from the scalar code to the vector code is called as vectorization.
6. Vectorizing compiler (or Vectorizer) : A compiler capable of vectorization is called as a vectorizing compiler or simply a vectorizer.
7. Flushing time : The time between the decoding of a vector instruction and the exit of the first result from the pipeline is called as flushing time.

8.4 VECTOR INSTRUCTION TYPES

Vector instructions are of various types. We shall discuss them one by one now.

I. Vector-vector instructions

As shown in Fig. 8.4 (a) below, one or two vector operands are fetched from the respective vector register. Then they enter through a functional pipeline unit to the vector register, V_i . That is,

$$f_1 : V_i \rightarrow V_j$$

$$f_2 : V_j \times V_k \rightarrow V_i$$

For mapping f_1 , the example can be like,

$$B(I) \leftarrow \sqrt{A(I)} \text{ (vector-square root)}$$

For mapping f_2 , the example can be like,

$$V_3 = V_1 + V_2$$

[V_i for $i = 1, 2, 3$ are vector registers]

II. Vector-scalar instructions

As shown in Fig. 8.4 (b) below, is a vector-scalar combination i.e., s_j and V_k register respectively. This operand is fetched and sent to the vector register, v_j , via a function pipeline.

That is,

$$f_3 : S \times V_i \rightarrow V_j$$

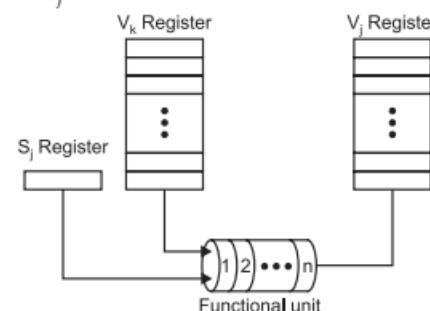


Fig. 8.4. (a) Vector-vector instruction.

Fig. 8.4. (b) Vector-scalar instruction.

For example : A scalar product $S \times V_1 = V_2$ Herein, the elements of V_1 are each multiplied by a scalar, S , to produce vector, V_2 , of equal length.

III. Vector memory instructions

As shown in Fig. 8.4 (c) below, there is a vector load or vector store, element by element, between vector register (V) and memory (M).

That is,

$$\begin{aligned} f_4 : M \rightarrow V & \text{ (Vector load)} \\ f_5 : V \rightarrow M & \text{ (Vector store)} \end{aligned}$$

IV. Vector reduction instructions

These corresponds to the following mappings :

$$\begin{aligned} f_6 : V_i \rightarrow S_j \\ f_7 : V_i \times V_j \rightarrow S_k \end{aligned}$$

For example : For mapping f_6 , finding maximum, minimum, sum and mean value of all elements in a vector is a good example.

For mapping f_7 above, a good example is the dot-product which performs :

$$S = \sum_{i=1}^n a_i \times b_i \text{ (from two vectors)}$$

V. Gather and scatter instructions

These instructions use two vector registers to gather or to scatter vector elements randomly throughout the memory. That is,

$$\begin{aligned} f_8 : M \rightarrow V_1 \times V_0 & \text{ (Gather)} \\ f_9 : V_1 \times V_0 \rightarrow M & \text{ (Scatter)} \end{aligned}$$

Please note here that 'Gather' is an operation that fetches from memory, the non-zero elements of a sparse vector using indices that themselves are indexed.

On the other hand, scatter operation is a reverse operation. It stores into memory, a vector in a sparse vector whose non-zero entries are indexed. (A matrix in which most of the entries are zeros is called as a **sparse matrix**).

VI. Masking instructions

These types of instructions use a **mask** (Vector mask) to compress or to expand a vector to a shorter or longer index vector, respectively.

That is,

$$f_{10} : V_0 \times V_m \rightarrow V_1$$

VII. Chaining

It is a technique used in some vector processors in which the output of one pipeline is directly released into another pipeline. This enhance the performance of processors. Also, the time delays due to storing of intermediate results can be eliminated.

An **intelligent** compiler should have a capability of detecting sequences of operations that can be chained together.

For example :

$$\begin{aligned} \text{Say, } C(I) &= [A(I) \times B(I)] + [X(I) \times Y(I)] \\ &\text{for } I \leftarrow 1, 2, \dots, N \end{aligned}$$

Chaining can help here. This is how it helps. Say, we have two **load-store pipelines**

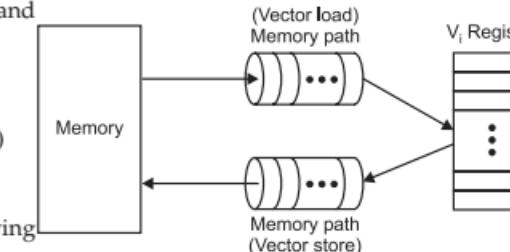


Fig. 8.4. (c) Vector-memory instructions.

(as we need to load two pairs of vector load operations – (A, B) and (X, Y) . Say, it starts at time, t_1 . Then, another **multiply pipeline** is present in which two vector multiplication operations i.e., $A \times B$ and $X \times Y$ are done at time t_2 . At time t_3 , however, two addition operations are performed in a **add pipeline**. The final result is available at time t_4 . With minor pipeline reconfiguration delay, the store operation begins at t_5 . This is shown in the timing diagram of figure 8.5 below.

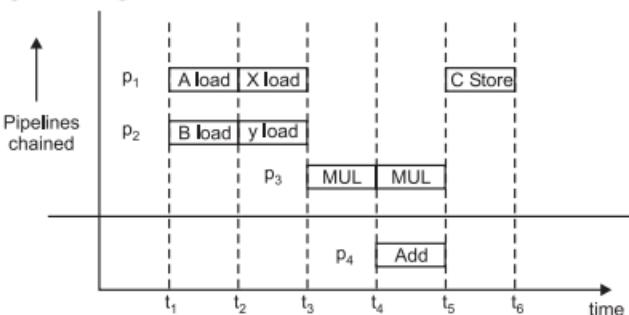


Fig. 8.5. Pipeline chaining of given expression.

∴ The entire operation requires :

$$t_6 - t_1 = 4N + \Delta_1 \text{ time.}$$

Where $\Delta_1 = t_2 - t_1$ = delay of one pipeline.

VIII. Some special instructions

- (a) A **Boolean vector** can be generated as a result of comparing two vectors.

For example :

Say, $X = (2, 5, 8, 7)$

& $Y = (9, 3, 6, 4)$

When the compare instruction,

$Z = X > Y$ is executed, then the Boolean vector,

$Z = (0, 1, 1, 1)$ is generated.

That is, all the vector elements are compared one by one. If the condition is true (1) then 1 is generated in the Boolean vector (Z) else 0 is added to Z .

- (b) A **compress instruction** will shorten a vector under the control of masking vector.

For example :

Let $X = (1, 2, 3, 4, 5, 6, 7, 8, 9)$

& $Y = (0, 0, 0, 0, 0, 0, 0, 0, 1)$ (a mask)

When compress instructions $Z = X(Y)$ is executed then the compressed vector, $Z = (9)$ is generated.

- (c) A **merge instruction** combines two vectors under the control of masking vector.

For example :

Let $X = (1, 3, 5, 6)$

& $Y = (2, 4)$

When the merge instruction, $Z = X, Y(M)$ is executed then the merged vector, Z is $(1, 2, 3, 4, 5, 6)$ is generated.

Please note here that in vector processing, vector instructions perform the same operation on different data sets repeatedly whereas in scalar processing, instructions need to perform the operation on single pair of operands.

8.5 VECTOR PERFORMANCE MODELING

The performance of a pipelined vector processor is measured on the basis of the following parameters :

1. Speed-up (S_k)
2. Utilization efficiency (η)
3. Throughput (W)
4. Performance-to-cost ratio (PCR)

To explain these parameters, let us assume certain notations that will be used again and again.

k : The number of pipeline stages.

T : The total pipeline delay in one instruction execution.

n : The number of instruction contained in a task.

N_i : The length of vector operands used in the i^{th} instruction ($1 \leq i \leq n$).

W : The throughput of a pipeline computer.

T_i : The time required to finish the i^{th} instruction in a pipeline computer ($1 \leq i \leq n$).

T_p : The total time required to finish a task consisting of n -instructions.

S_k : The speed-up of a pipeline computer with k stages. [over a serial computer].

η : The efficiency of a pipeline computer.

Here in, if $N_i = 1$ means scalar instruction.

& $N_i > 1$ means vector instruction.

We will now discuss the parameters one by one.

I. Speed-up (S_k)

The time required per stage,

$$\tau = \frac{T}{K}$$

For a single vector instruction with vector length, N_i , the delay is :

$$\begin{aligned} T_i &= (k - 1) \cdot \tau + N_i \cdot \tau \\ &= (N_i + k - 1) \cdot \frac{T}{K} \end{aligned}$$

Where $(k - 1)\tau$ is the time required to fill up the pipe when N_i becomes long.

When k is small,

$$T_i = N_i \cdot \tau$$

Now, say there is a sequence of n -vector instructions. Suppose that the execution of different types of vector instructions takes the same amount of time, if they have same vector length.

$$\therefore \text{Total execution time required in a pipeline processor} = T_p = \sum_{i=1}^n T_i$$

$$= \frac{T}{K} \left[(K-1) \cdot n + \sum_{i=1}^n N_i \right] \quad \dots(1)$$

If a sequential machine is used then

$$T_s = T \cdot \sum_{i=1}^n N_i \quad \dots(2)$$

$$\therefore \text{Speed-up } (S_k) = \frac{T_s}{T_p}$$

$$= \frac{T \sum_{i=1}^n N_i}{\left[T \left[(K-1)n + \sum_{i=1}^n N_i \right] \right] / K} \quad [\text{From (1) and (2)}]$$

or

$$S_k = \frac{K \cdot \sum_{i=1}^n N_i}{(K-1)n + \sum_{i=1}^n N_i}$$

So, we can achieve a maximum possible speed-up if long vectors are processed in the pipeline.

II. Utilization Efficiency (η)

The efficiency of the pipeline computer is defined as the total space-time product required by the job divided by a total available space-time product.

$$\therefore \eta = \frac{S_k}{K}$$

So, efficiency (η) of a pipeline is the ratio of actual speed-up to the maximum possible speed-up, K .

Please note that the efficiency of the pipeline depends on the vector length distribution i.e., too many scalar operations of different types will downgrade the performance.

III. Throughput (W)

When we say, maximum throughput we mean to say that the results that can be generated per unit time. The throughput rate reflects the processing capability of a pipeline processor. For vector processor what is a throughput? To understand this, let us assume certain notations :

Let T_s : Total time required to process a job in a non-pipelined, serial processor.
 K : pipeline stages.

For a pipelined processor with K -stages, the clock period, τ should be :

$$\tau = \frac{T_s}{K} + \theta \quad \dots(1)$$

where τ : Clock period for pipeline
 θ : Latch delay

Now, if the pipeline filling time is neglected then we can assume that one instruction will flow out of the pipe at every τ units of time.

\therefore Total time required for processing n -instruction $= n \cdot \tau$

$$= T_s + n \cdot \theta$$

$$\therefore \text{Throughput}, \quad W = \frac{n}{n\tau} = \frac{n}{n\left(\frac{T_s}{K} + \theta\right)} \quad [\text{From (1)}]$$

$$\therefore \boxed{W = \frac{1}{\left(\frac{T_s}{K} + \theta\right)}}$$

So, a higher throughput may be obtained at the cost of more hardware.

$$\therefore \eta = \frac{n \sum_{i=1}^k C_i \cdot \tau_i}{\sum_{i=1}^k C_i \cdot \left[\sum_{i=1}^k \tau_i + (n-1)\tau \right]}$$

Where C_i = Cost of i^{th} stage.

τ_i = Delay of i^{th} stage.

k = stages

n = Total number of jobs streamed into the pipeline during the period of measurement.

IV. Performance-to-cost ratio (PCR)

$$\text{In case of vector processors, } PCR = \frac{W}{C + kd} = \frac{1}{\left(\frac{T_s}{N} + \theta\right)(C + KD)} \quad \dots(1)$$

where d = Average latch cost

$C + kd$ = cost of entire pipeline.

Now, the optimal design of a static linear pipeline processor require k_0 stages such that PCR is maximized.

$$\text{i.e.,} \quad \frac{\partial(PCR)}{\partial k} = \frac{T_s \cdot \frac{C}{k^2} - \theta d}{\left(\frac{T_s}{k} + \theta\right)^2 \cdot (C + kd)^2} \quad \dots(2)$$

$$\text{when} \quad \frac{\partial(PCR)}{\partial k} = 0$$

then $T_s \cdot \frac{C}{k^2} - \theta d = 0$; [From equation-2]

At $k = k_0$,

$$T_s \cdot \frac{C}{k^2} - \theta d = T_s \cdot \frac{C}{k_0^2} - \theta d = 0$$

or $T_s \cdot \frac{C}{k^2} - \theta d = 0$

or $k_0 = \sqrt{\frac{T_s \cdot C}{\theta d}}$

Which is **maximum**.

8.6 VECTORIZATION

For vectorization process, the vectorizing compilers/vectorizers are used.

A vectorizing compiler analyzes Do-Loop in a high level language, that can be executed in parallel and generates object codes with vector instructions. **Please note that the performance will be proportional to the vectorization ratio. It means that the higher is the vectorization ratio, the higher is the performance.** To achieve this, the compiler replaces a block of complicated data access and restructures the program sequence. However, some barriers to vectorization do exist in conditional and branch statements, non-linear and indirect indexing and subroutine calls within loops, sequential dependencies.

Actually, in vector processing, we identify the Do-Loops or the segments of the program which are SIMD in nature and we allocate them to various functional units simultaneously. **This process is called as vectorization.** While the efficient allocation is called as **optimization**. The performance of vector processors depends on this **vectorization and optimization**.

8.7 DESIGN OF A VECTORIZING COMPILER

A vectorizing compiler (or vectorizer) has the capability of detecting parallelism in serially coded programs. It recognizes the constructs that can be executed in parallel. This vectorizer (s) basically performs precedence analysis and code generation. **Please note here that an ideal vectorizer performs analysis of data dependencies and determines the possibility of vectorization.**

An effective vectorizer should be able to :

1. Determine the flow pattern between subprograms or Do-Loops.
2. Check the precedence relationship among the subprograms.
3. Check the locality of variables.
4. Determine the loop variables.
5. Check the independence of variables.
6. Replace the inner loop with vector instructions.

Thus, vectorizer is a post compilation software which after going through the

compiled code of the user program, detects the independent Do-Loops with the help of various dependent analysis algorithms and generate strip-mining code with the allocation of vector registers. Consider the following for-loop of C :

```
For    (i = 0 ; i < n ; i++)
      C[i] = A[i] * B[i]
```

Herein, the size of vectors A, B and C is not predefined as the size will depend on the value of n, in above for-loop. So, vectorization becomes a problem. The solution to this problem can be –

- (a) **Vector's length can be controlled** : We can create a vector length register (VLR) which will control the length of vector operation. If we define values of VLR such that $VLR < \text{length of the vector}$ then such Do-Loops / For-Loops can be vectorized until the real vector length is less than the maximum vector length. **This is called as strip-mining.** But it has some problems like large start-ups which degrades the performance.
- (b) **Vector strides** : This technique can be used when the elements of vectors are not adjacent. As explained earlier also, **the stride is the distance which separates the elements.** The value of stride could be different for different variables. **The stride is also known as a skew distance.** When a vector is loaded into a vector-register then the stride is one meaning that all the elements of vector are adjacent.

We are in a position to solve some examples now.

Example 1. Convert the following code into its vector form :

- (a) DO 10 I = 1, 100, 1

$$10 C(I) = A(I + 2) + B(I + 3)$$

- (b) DO 20 I = 1, N

$$20 \text{ IF } (L(1) \cdot NE \cdot 0) A(I) = A(I) + 1 ? \quad [\text{KUD, BE (CSE) 8th Sem; Dec. 1996}]$$

Solution. (a) It's vector equivalent is :

$$c(1 : 100 : 1) = A(3 : 102 : 3) + B(4 : 103 : 3)$$

- (b) It's vector equivalent is :

$$(L(I) \cdot NE \cdot 0) A(1 : N) = A(1 : N) + 1$$

Example 2. Vectorize the following FORTAN code :

DO 100 I = 1, N

A(I) = B(I) + C(I)

100 B(I) = 2 × A(I + 2) ?

Solution. It's vector form is :

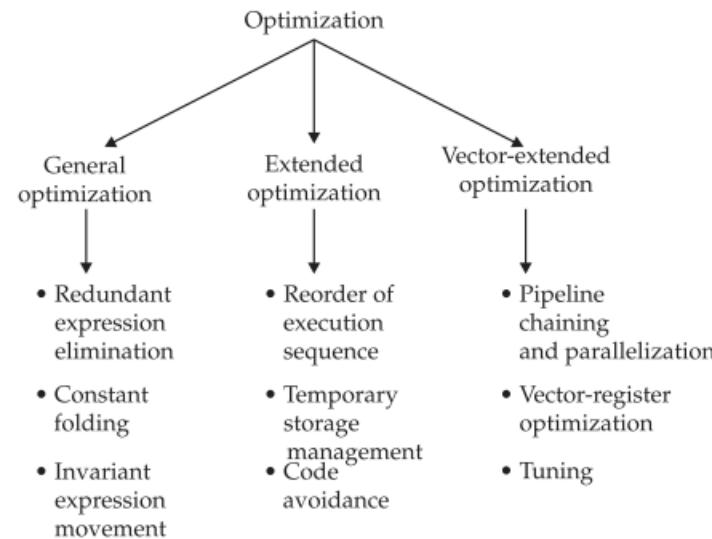
$$A(1 : N) = B(1 : N) + C(1 : N)$$

$$\text{TEMP } (1 : N) = A(3 : N + 2)$$

$$B(1, N) = 2 \times \text{TEMP } (1 : N).$$

8.8 OPTIMIZATION OF VECTOR FUNCTIONS

The process of efficient allocation of the vectors is known as optimization. Optimization is a tool which is used for efficient scalar and vector code generation. There are 3 main ways of optimization :



We shall discuss these nine optimization techniques one by one.

I. Redundant expression elimination

We try to remove redundant expressions from the code. This reduces the number of memory accesses and the execution time.

For. e.g. : Say, a vectorized output is :

$$C(1 : 20) = A(1 : 99 : 2) \times B(1 : 20) + A(1 : 99 : 2) \times C(1 : 20)$$

Then, a possible set of intermediate code is :

$$\begin{aligned}
 V_1 &\leftarrow A(1 : 99 : 2) \\
 V_2 &\leftarrow B(1 : 20) \\
 V_3 &\leftarrow V_1 \times V_2 \\
 V_4 &\leftarrow A(1 : 99 : 2) \\
 V_5 &\leftarrow C(1 : 20) \\
 V_6 &\leftarrow V_4 \times V_5 \\
 V_7 &\leftarrow V_6 \\
 C(1 : 20) &\leftarrow V_7
 \end{aligned}
 \quad \Rightarrow \quad
 \begin{aligned}
 V_1 &\leftarrow A(1 : 99 : 2) \\
 V_2 &\leftarrow B(1 : 20) \\
 V_1 &\leftarrow V_1 \times V_2 \\
 V_1 &\leftarrow 2 \times V_1 \\
 C(1 : 20) &\leftarrow V_1 \\
 &\text{(optimized code).}
 \end{aligned}$$

Please note here that the contents of A and B, A and C are same. So, we write one instruction :

$$V_1 \leftarrow 2 \times V_1$$

II. Constant folding at compile time

Constant folding means shifting computations from run-time to the compile-time.

For example : A Do-loop for generating the array, $A[I] = I$ for $I = 1, 2, \dots, 50$ can be avoided in the execution phase because the array, $A(I)$ can be generated by constant vector $(1, 2, 3, \dots, 50)$ initialized in the compile time.

III. Invariant expression movement

When the innermost loops are just scalar encodings of vector operation, then the innermost loop can be replaced by some vector operations if there is no data dependence relation. Most of the do-loops are the combination of loop variant and loop invariant statements. In such situations, the loop invariant expressions can be moved out of the loop. This process of moving the loop-invariant expressions out of the loop is called as **code motion**.

For example : Consider a nested do-loop given below :

```

Do 20 I = 1, N
...
Do 20 J = 1, M
...
B(I, J) = B(I, J) + A(J) * C(J)
20 CONTINUE

```

If no other computations affect the variables A, B, C, I and J then the above program can be vectorized as :

```

Do 20 I = 1, N
...
B(I, *) = B(I, *) + A(*) * C(*)
Do 20 J = 1, M
...
20 CONTINUE

```

IV. Reorder the execution sequence

When we consider a multifunction pipeline, reconfiguration of pipe for different function must be done which requires the flushing of the pipe, establishing new data paths etc. So, the instructions of the same type may be grouped together for pipeline execution. This process minimizes the number of required pipeline configurations.

For example : The instructions on the left of the following code block can be regrouped to yield the sequence on the right :

$$\begin{array}{ll}
 A_1 \leftarrow A_2 + A_3 & A_1 \leftarrow A_2 + A_3 \\
 A_4 \leftarrow A_5 \times A_6 & A_7 \leftarrow B_1 + B_2 \\
 A_7 \leftarrow B_1 + B_2 & \Rightarrow B_4 \leftarrow C_1 + C_2 \\
 B_3 \leftarrow A_8 \times A_9 & A_4 \leftarrow A_5 \times A_6 \\
 B_4 \leftarrow C_1 + C_2 & B_3 \leftarrow A_8 \times A_9 \\
 \end{array} \quad \begin{array}{l}
 (First\ sequence) \qquad \qquad \qquad (Second\ sequence)
 \end{array}$$

The first sequence above, requires three pipeline configurations while the second requires only one. An intelligent compiler should be able to reorder the execution sequence to minimize the number of required pipeline reconfigurations.

V. Temporary storage management

In case of long vector processing, the generation of too many intermediate vector quantities can lead to a serious problem. In this case, the vectorizer should have the capacity of allocating and deallocating the temporary storage dynamically. This means that the intermediate vectors must be stored in the memory. Temporary storage management is closely related to the policy of register allocation.

In any case, the run-time storage management is an expensive feature to be included in FORTRAN based language. So, to reduce the design cost, two techniques are used :

- (a) The compiler allocates a specific area to fixed length and variable length temporaries.
- (b) The number of fixed-length temporaries can be increased by **strip mining** with width equal to the length of vector registers in the target machine.

VI. Code avoidance

For vector optimization we avoid excessive copying arrays.

For example : Consider the following code sequence :

$A(1 : 50) = B(1 : 99 : 2)$

$C(1 : 50) = 2.0 * A(1 : 50)$

During its execution, the following code will be produced :

$V \leftarrow B(1 : 99 : 2)$

$A(1 : 50) \leftarrow V_1$

\vdots

$V_1 \leftarrow A(1 : 50)$

$V_1 \leftarrow 2.0 * V_1$

$C(1 : 50) \leftarrow V_1$

Here, a copy is avoided, if the compiler adjusts the storage mapping function for array, A to reference the storage for array, B. The optimized code will be :

$V_1 \leftarrow B(1 : 99 : 2)$

$V_1 \leftarrow 2.0 * V_1$

$C(1 : 50) \leftarrow V_1$

VII. Pipeline chaining and parallelization

This has already been discussed in section 8.4 and figure 8.5.

VIII. Vector register optimization

To perform chaining of vector operations efficiently, vector register allocation is an important aspect. The execution units must be fed a continuous stream of operands to achieve the maximum computing power. Retaining the vectors in the registers between the operations is one way to achieve this. However, the number of available vector registers in the system are limited. Vector register allocation emphasizes local allocation rather than global allocation.

For example : Consider the vector expression as : $X - Y * Z$. It can be executed in Cray-1 by using the three vector registers V_1 , V_2 and V_3 as follows :

```

 $V_1 \leftarrow X$ 
 $V_2 \leftarrow Y$ 
 $V_3 \leftarrow Z$ 
 $V_2 \leftarrow V_2 * V_3$ 
 $V_1 \leftarrow V_1 - V_2$ 

```

This sequence requires **31 clock periods**. If we do multiply before loading the vector-*A* into a register then,

```

 $V_1 \leftarrow Y$ 
 $V_2 \leftarrow Z$ 
 $V_1 \leftarrow V_1 * V_2$ 
 $V_2 \leftarrow X$ 
 $V_2 \leftarrow V_2 - V_1$ 

```

and this sequence requires **27 clock periods**. In general, the problem of register allocation is solved using “round-robin” allocation of registers. Another solution for this problem is to generate code for a group of arithmetic expressions using large number of virtual registers and map the virtual registers into a finite number of real registers. This process may require more than the number of available registers in which case spilling must be done.

IX. Tuning for interactive vectorization

Tuning is defined as an on line process of vectorization. Tuning tools are necessary to provide some user interaction in advance vectorization. The user can modify the source program with the help of an **interactive vectorizer package**. During compilation, various compiler codes will be displayed on the monitor by which the programmer can come to know whether vector length distribution is proper or not. The modified source program will be optimized towards full vectorization by the vectorizing compiler. The vectorizing compiler generates the optimal object code with vector instructions after the tuning process is completed. Both Cray-1 and VP-200 have some tuning facilities.

8.9 CASE STUDY : CRAY FAMILY & CRAY -1

If we look at the performance of four generations of Cray-systems then we find some interesting advancements. We compare different variants of Cray in a tabular form below:

System	CPUs	Clock (MHz)	FP results per Clock per CPU	Words moved perclock /CPU	MFLOP rate
Cray-1	1	80	2	1	80
X-MP	4	105	2	3	840
Y-MP	8	166	2	3	2667
C 90	16	240	4	6	15360

Out of these, we shall explain **Cray-1 system only**.

Cray-1 is a commercial supercomputer (1970) with pipelined vector and scalar processing units. It is based on a Register-to-Register architecture. Each vector register stores 64 elements. Each element or word has 64 bits. It has 12 functional pipelined units. All the 12 units can operate concurrently. The Cray-1 is the first vector processor to apply chaining technique. Please note that Cray-1 is not a standalone computer.

It requires a front-end host computer also, which will serve as a system manager. It may use Data General Eclipse computer as front-end.

There are four groups of functional units :

- (a) **Vector pipelines** : These pipelines perform integer or logical operations on vectors.
- (b) **Floating-point pipelines** : These pipelines execute floating-point operations using scalars or vectors.
- (c) **Scalar pipelines** : These pipelines carry out integer or logical operations on scalars.
- (d) **Address pipelines** : These pipelines perform address calculations.

All this is shown in Fig. 8.6

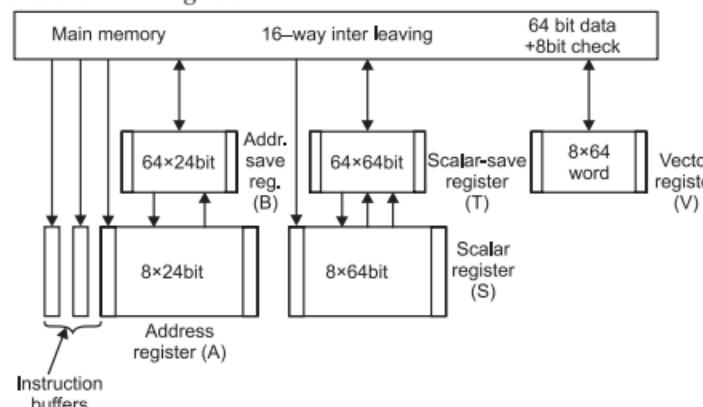


Fig. 8.6. Cray-1 (functional units and registers).

Note here that there is no floating-point divide unit. So, x/y is done as follows :

$$\frac{x}{y} = x \cdot \left(\frac{1}{y}\right)$$

So, a reciprocal approximation pipeline is used. Also shown in Fig. 8.6, there are five programmable registers : A, B, S, T and V.

The Cray-1 allows one memory read and write per clock cycle. Please note that it can read only one vector and write one vector result at the same time. When more than one read or write is required for the same operation then one of the operation is postponed.

The main memory is a 16-way interleaved memory. It is a semiconductor memory with error detection and correction logic. There are 12 I/O channels.

The Cray-1's FORTRAN compiler is an optimizing compiler. This compiler takes care of vectorizations and optimizations.

The 'A' registers are known as address registers. These are used as memory address registers, index registers, for shift counts and loop counts.

The 'B' registers are called as address-save registers. They are used as auxiliary storage/buffer for the 'A' registers.

The 'S' registers are called as scalar registers. They are used as source and destination registers for scalar arithmetic and logical instructions.

The 'T' registers are called as scalar-save registers. They are used as auxiliary storage for the 'S' registers.

The 'V' registers are called as vector registers. They are used as source and destination registers for the vector functional units (pipelines).

SUMMARY

In this chapter, vector processing is explained and compared with array processing. The basic architecture of a vector supercomputer has been explained. Vector processors have been classified. Role of vectorizers and optimizers has been explained. Finally, a case study on Cray-1 has been presented.

MULTIPLE CHOICE QUESTIONS (MCQS) WITH ANSWERS

10. The process of efficient allocation of the vectors is known as :

(a) Vectorization	(b) Optimization
(c) Code generation	(d) None of the above
11. Shifting computations from run-time to the compile time is called as :

(a) Constant folding	(b) Code motion
(c) Reordering	(d) None of the above
12. The process of moving the loop-invariant expressions out of the loop is called as :

(a) Loop unrolling	(b) Code motion
(c) Reordering	(d) None of the above
13. An online process of vectorization is called as :

(a) Tuning	(b) Vectorization
(c) Optimization	(d) None of the above
14. Which of the following systems can be used as a front-end computer with Cray-1 :

(a) CDC 6600	(b) IBM 360
(c) Data General Eclipse	(d) None of the above
15. 'A' -registers of Cray-1 can be used as :

(a) Index Registers	(b) Address -Save Registers
(c) Scalar-Save Registers	(d) None of the above

:: ANSWERS ::

- | | | | | |
|---------|---------|---------|---------|---------|
| 1. (c) | 2. (a) | 3. (c) | 4. (a) | 5. (a) |
| 6. (a) | 7. (a) | 8. (c) | 9. (b) | 10. (b) |
| 11. (a) | 12. (b) | 13. (a) | 14. (c) | 15. (a) |

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

1. Consider the following sequence of instructions :

$$I_1 : V_3 = V_1 \times V_2$$

$$I_2 : V_4 = V_3 + V_6$$

$$I_3 : V_0 = V_0 + V_7$$

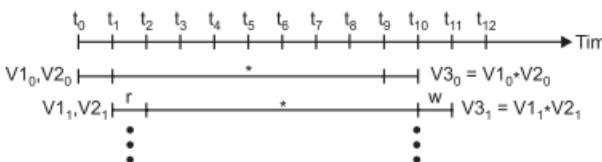
$$I_4 : S_2 = V_1 \times S_3$$

Explain why or why not, these instructions can be parallelized ?

- Ans.** Here, after I_1 has been issued, I_{12} will not be issued as then it results in RAW hazard. This is so, because of V_3 in both I_1 and I_2 . When I_2 is issued, I_3 will be held back till I_2 release the floating-point adder that it is using. Now, I_4 and its predecessors do not have dependencies (or conflicts) So I_4 can start the execution one clock period after the issue of I_3 .

Consider $I_1 : V_3 = V_1 \times V_2$ (for I elements, where $i \leftarrow 1, 2, 3 \dots$)

We can draw its timing diagram as follows :



Note here that each horizontal line indicates the complete processing of a pair of vector elements i.e.,

$V1_0$ and $V2_0$ are 'read' and

$V3_0 = V1_0 \times V2_0$ is computed. So, at time t_9 , the result of $V3_0$ is present which is written after time, t_9 . This writing continues till time, t_{10} . Please note that these read and write operations have been marked as 'r' and 'w' on the horizontal time. It reads from vector registers to the multiplier and write results back to the destination vector register. Here, 7-stage multiplier produces the first output at time, t_8 , starting at time, t_0 .

An additional clock period is required to write the product of the multiplier into an element of vector register. And the successive outputs can be written at one clock period intervals.

2. Consider the following instruction sequence :

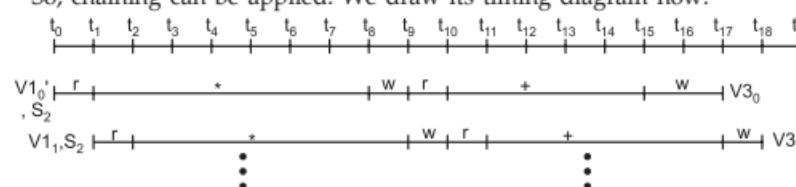
$$I_{31} : V2 := V1 \times S2$$

$$I_{32} : V3 := V2 + V0$$

Show how chaining can be done here. Find the respective chain slot times for each of the two instructions? How many clock cycles are required? Draw its timing diagram also.

Ans. From the two instructions, I_{31} and I_{32} we find that there are no dependencies except for the RAW data dependency involving $V2$ (in I_{32} instructions).

So, chaining can be applied. We draw its timing diagram now.



Each horizontal line indicates the production of vector, $V3$. The first element, $V2_0$ of $V2$ is available at t_9 and immediately the first element of $V3_0$ of $V3$ is available at t_{17} . We know that the time at which the successor instruction is issued as soon as the first output of the predecessor instructions is produced, is known as a chain slot time.

\therefore Chain slot time for I_{31} is t_9 and for I_{32} is t_{17} .

The first output of I_{32} is produced at time, t_{17} . Thus, from the timing diagram, we can say that the total time required to process both I_{31} and I_{32} is 80 clock periods.

3. Explain the following terms :

(a) Vector and Scalar balance point.

(b) Vectorization ratio is user code.

- (c) Vectorization compiler or vectorizer.
- (d) Vector reduction instructions.
- (e) Gather and scatter instructions.
- (f) Sparse matrix and masking instruction

[IGGSIPU, B. Tech. 7th sem; Dec. 2007]

- Ans.**
- (a) The percentage of vector code in a program required to achieve equal utilization of vector and scalar hardware is called as **vector and scalar balance point**.
 - (b) The percentage of code in a program which can be vectorized is called as **vectorization ratio** in user code.
 - (c) A compiler capable of vectorization is a **vectorizer**.
 - (d) The instructions corresponding to the following mappings :

$$\begin{aligned} f : V_i &\rightarrow S_i \\ g : V_i \times V_j &\rightarrow S_k \end{aligned}$$

- (e) A **gather instruction** fetches the non-zero elements of a sparse vector using indices.

$$f : M \rightarrow V_1 \times V_0$$

A **Scatter instruction** stores a vector in a sparse vector whose non-zero entries are indexed.

$$f : V_1 \times V_0 \rightarrow M$$

- (f) A **sparse matrix** is one in which most of the entries are zero. A **masking instruction** uses a mask vector to compress or expand a vector to a shorter or long index vector, respectively, corresponding to the following mapping :

$$f : V_0 \times V_m \rightarrow V_1.$$

- 4 Distinguish among the following vector processing machines in terms of architecture, performance and cost :
- (a) Full-scale supercomputers.
 - (b) High-end mainframes or near supercomputers.
 - (c) Mini supercomputers or supercomputing workstations ?

Ans.

Class	Architecture	Performance	Cost
Full-scale Computers	<ul style="list-style-type: none"> • Multiprocessor • Multi vector pipeline • Pipeline chaining 	>1 Gflops	\$2~25 million
High-end Mainframes or Near super-computers	Attached vector processor	> 200 Mflops	\$1~4 million
Mini-super Computers or Supercomputing workstations	Multi-computer	> 100 Mflops	\$0.1~1.5 million

5. Explain the following terms :
- (a) Compound vector functions.

- (b) Vector loops and pipeline chaining.
- (c) Systolic program graphs.
- (d) Pipeline network or pipe nets ?

- Ans.**
- (a) A composite function of vector operations converted from a looping structure of linked scalar operations is called as a **compound vector function**.
 - (b) The program construct for processing long vectors is called as a **vector loop**. When a vector has a length greater than that of the vector registers, segmentation of the long vector into fixed-length segments is necessary. One segment is processed at a time.
 - (c) Pipeline chaining links vector operations following a linear data flow pattern. Vector registers are used as interfaces between functional pipelines. Continuous data flow is maintained in successive pipelines.
 - (d) A **pipe net** is constructed from interconnecting multiple functional pipelines through two buffered crossbar networks, which are themselves pipelined.

6. Compare CM-2 and CM-5 machines ?

Ans.

Machine	Architecture mode	Operation performance	Potential	Improvement
CM-2	64k bit-slice Processors, Hypercube	SIMD	10 G Flops	—
CM-5	16k SPARCcs, 4-ary fat tree	SIMD, MSIMD, Synchronous MIMD	2T Flops	Mixture of parallel techniques

EXERCISE QUESTIONS

1. Write short notes on :
Vector-Access Memory schemes. [GGSIPU, B.Tech. 7th Sem; Dec. 2007]
2. Explain the structural and operation differences between register-to-register and memory-to-memory architectures in building multipiplined supercomputers for vector processing. Comment on the advantages and disadvantages in using SIMD computers as compared with the use of pipelined supercomputers for vector processing.

[UPTU, B. Tech. (CSE) 8th Sem.; 2008-09 & GGSIPU, M. Tech. 1st sem; Dec. 2003]

3. Write short notes on :
(a) Vector processors [GGSIPU, M. Tech. 1st sem; Dec. 2004]
4. What is a vector? Explain vectorization process with respect to the following example :

$$\begin{aligned}
 \text{Do } 50 I &= 1,100 \\
 A(I) &= B(I) \times C(I) \\
 D(I) &= \text{DULT}(A(I) \times X(I)) \\
 E(I) &= D(I)/B(I) + A(I)
 \end{aligned}$$

5. Write short notes on :
 - (a) Optimization of vector function.
 - (b) Vector processor.
 - (c) Types of vector instructions.

[Pune Univ., B.E. (CSE) ; 2nd Sem; May-2004]

6. What is vector chaining? How can it speedup the processing? Explain with suitable example? [Pune Univ., B.E. (CSE) ; 2nd Sem; May 2001, Dec. 2001, May 2006]

7. What are vector processors? Discuss two different architectural configurations of vector processor? [Pune Univ., B.E. (CSE); 2nd Sem; May 2002]

8. Explain the following terms related to vector processing :

1. Vectorization ratio in user code.
2. Vectorization compiler.
3. Vector reduction instruction.
4. Gather and scatter instruction.

[Pune Univ., B.E. (CSE); 2nd Sem, May 2006]

9. Explain various optimization techniques ?

10. Explain with a suitable diagram the working of Cray-1 computer ?

11. (a) Describe various vector-access memory schemes.

- (b) Write short notes on-vector instruction types.

[GGS IPU, B.Tech., (CSE)-7th Sem., Dec. 2011]



CHAPTER

9 SYNCHRONOUS PARALLEL PROCESSING : SIMD

9.0 INTRODUCTION

A synchronous array of parallel processors is called an **array processor**. It consists of multiple processing elements (PEs) under the supervision of one control unit (CU). An array processor can handle **single instruction and multiple data (SIMD)** streams. Thus, the **array processors** are also known as **SIMD computers**. SIMD machines are especially designed to perform vector computations over matrices or arrays of data.

SIMD computers appear in **two basic architectural organizations** :

1. **Array processors** using random-access memory.
2. **Associative processors** using content addressable (or associative) memory.

Please note here that an **associative processor** is a special type of array processor where PEs correspond to the words of an associative memory.

9.1 SIMD ARCHITECTURE

In general, an array processor/SIMD may assume two different configurations. We discuss these two configurations one by one now.

Configuration-I (Illiad-IV)

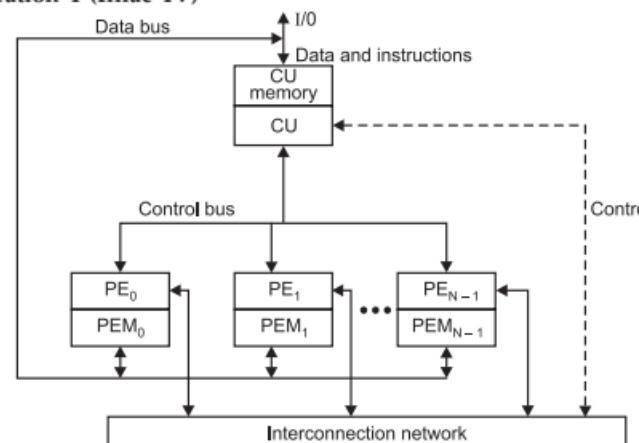


Fig. 9.1. Configuration -I (Illiad-IV).

Configuration -I has been implemented in the Illiac-IV computer. In this configuration-I, one control unit (CU) controls N synchronized PEs. Each PE_i is an

Arithmetic Logic Unit (ALU) with attached working registers and local memory, PEM_i , which stores the distributed data.

The CU also has its own main memory for storage of programs. The system and user programs are executed under the control of CU. The user programs are loaded into the CU memory from an external source. The function of CU is to decode all the instruction and determine where the decoded instructions should be executed.

Scalar or control type instructions are directly executed inside the CU. Vector instructions are broadcast to the PEs for distributed execution to achieve special parallelism through duplicate arithmetic units (PEs).

All PEs perform the same function synchronously in **lock-step fashion** under the command of CU.

Vector operands are distributed to the PEMs before parallel execution in the array of PEs.

Distributed data can be loaded into the PEMs from an external source via the system data bus or via the CU in the broadcast mode using control bus.

Masking schemes are used to control the status of each PE during the execution of vector instructions. Please note that each PE may be activated or deactivated during **instruction cycle**. A masking vector is used to control the status of each PE. Also note that only the enabled PEs perform computations. Data exchanges among the PEs are done via **inter-PE communication network** which performs all the necessary data routing and manipulation function. This interconnection network is under the control of the control unit.

An array processor is interfaced to the host computer through the control unit. The host computer is a general purpose machine which serves as "**operating manager**" of the entire system, consisting of host and processing array.

Functions of host computer :

- (i) Resource management.
- (ii) Peripheral and I/O Supervisions.

The CU of the processor array directly supervises the execution of programs and the host machine performs the executive and I/O functions with the outside world.

Configuration -II (BSP)

Another possible way of constructing an array processor is shown above in Fig. 9.2 Configuration -II

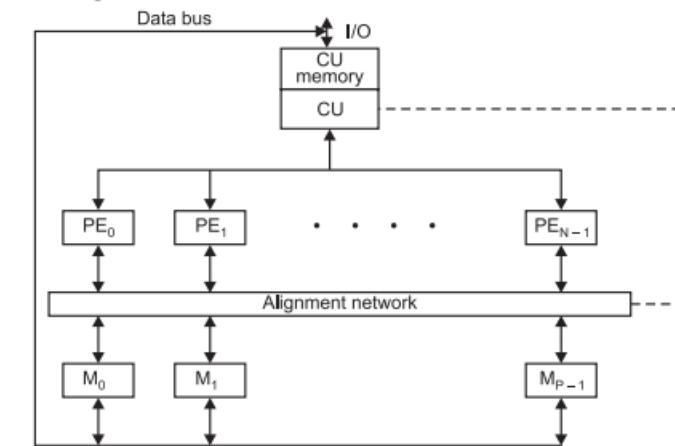


Fig. 9.2. Configuration-II (BSP).

has been implemented in Burroughs Scientific Processor (BSP). There are N PEs and P memory modules in this configuration. These two numbers are not necessarily equal. In fact, they have been chosen to be relatively prime. The alignment network is a path-switching network between the PEs and the parallel memories. Such an alignment network is desired to allow conflict-free accesses of the shared memories by as many PEs as possible.

This configuration-II differs from the configuration-I as -

- (a) The local memories attached to the PEs are now replaced by parallel memory modules shared by all the PEs through the alignment network.
- (b) The inter-PE communication network is replaced by inter-PE memory alignment network, controlled by CU.

9.2 MASKING AND DATA-ROUTING MECHANISM

A set of masking schemes is one of the parameters of SIMD model. Each mask partitions the set of PEs into enabled and disabled subsets. The masking scheme is built within each PE and is continuously monitored by the CU which can set and reset the status of each PE dynamically at run-time.

Static networks and dynamic networks have been implemented for inter-PE data routing in SIMD computers. In case of a multi-computer network, the data routing is achieved through message passing. Hardware routers are used to route message among multiple computer nodes. Data routing functions among the PEs include shifting, rotation, permutation (one-to-one), broadcast (one to all) multicast (many-to-many), shuffle, exchange etc. These routing functions can be implemented on ring, mesh, hypercube or multistage networks.

Now, we will consider the configuration-I of an SIMD computer. Each PE_i is a processor with its own memory, PEM_i . It consists of :

- (a) A set of working registers and flags (A_i, B_i, C_i, S_i)
- (b) An ALU
- (c) A local index register (I_i),
- (d) An address register (D_i),
- (e) A data-routing register (R_i).

The data routing register (R_i) of each PE_i is connected to the data-routing register (R_j) of other PEs via interconnection network. Some array processors use two routing registers, one for input and other for output.

During instruction cycle, each PE_i is either in the **active** or the **inactive mode**. If PE_i is active , it executes the instructions broadcast to it by the CU and if PE_i is inactive, it will not.

Here, we denote N PEs as PE_i for $i \leftarrow 0$ to $N - 1$ where ' i ' is the address of PE_i . Please note that the address register (D_i) is used to hold the address of PE_i .

The masking schemes are used to specify the status flag (S_i) of PE_i as

$$\begin{aligned} S_i = 1 & \text{ means } PE_i \text{ is active.} \\ \& \\ S_i = 0 & \text{ means } PE_i \text{ is inactive.} \end{aligned}$$

In CU, the global index register, I and masking register, M are present. M register has N-bits denoted as M_i where $i = 0, 1, 2 \dots N - 1$.

When masking is set, the bit patterns in registers M and S (i.e., status register) are exchangeable upon the control of the CU.

Masking can be considered through hardware viewpoint. The physical length of the vector can be determined by the number of PEs. One dimensional linear vector of n -elements can be stored in all PEMs if $n \leq N$. But if $n > N$ i.e., with respect to long vector, the CU performs the segmentation of long vector into vector loops by setting of a global base address and offset. This means that long vectors can be stored by distributing the n -elements cyclically among the N -PEMs. Please note that ideally, N elements of a vector are retrieved from different PEMs simultaneously.

In the worst case, all the vector elements are in single PEM.

9.3 INTER – PE COMMUNICATIONS

The fundamental decisions in determining the appropriate architecture of an interconnection network for SIMD machines are made between the operation modes, control strategies, switching methodologies and network topologies.

9.3.1 Operation Mode

The operation modes of the interconnection networks can be classified into three categories :

- (a) Synchronous
- (b) Asynchronous and
- (c) Combined

Synchronous communication is needed for establishing communication path synchronously for data manipulation function or for data instruction broadcast.

Asynchronous communication is needed for multiprocessing where the connection requests are done dynamically.

Combined communication means that a system may be designed to support both synchronous and asynchronous processing.

Please note that SIMD machines choose the synchronous operation mode, in which lock-step operations among all PEs are enforced.

9.3.2 Control strategy

Interconnection networks consists of a number of switching elements and inter connecting links. If the control setting functions are managed by a centralized controller then it is called as a centralized control strategy. On the other hand, if it is managed by the individual switching element then it is called as a distributed control.

Please note that most of the existing SIMD interconnection networks choose the centralized control on all switch elements by the control unit.

9.3.3 Switching methodology

Two major switching methodologies are :

- (a) Circuit switching
- (b) Packet switching

In circuit switching, the actual physical path is established between a source and a destination. It is suitable for bulk data transmission. It is used in SIMD networks.

In packet switching, data is put in a packet and routed through the inter-connection network. Physical connection is not established in this case. Packet switching is more suitable for short data message. It is basically used in MIMD networks.

In **Integrated switching**, we combine the capabilities, of circuit switching and packet switching (both).

Please note that most SIMD interconnection networks are hardwired to assume circuit switching operations. Also note that, packet switching has been suggested mainly for MIMD machines.

9.3.4 Network topology

Network can be represented by a graph, in which **nodes** represents switching points and edges represents the communication links. The topologies tend to be regular and can be grouped into two categories – Static and dynamic topology.

[We have already given the differences between static and dynamic networks in tabular form in chapter –3. We will now discuss it again in section 9.4 below.]

In static topology, the links between two processors are passive and the dedicated buses cannot be reconfigured for direct connection to other processor.

In dynamic topology, it can be configured by setting the network's active switching elements.

The space of the interconnection network can be represented as :

{Operation mode} × {control strategy} × {switching methodology} × {Network topology}

Not every combination of the design features is interesting. Please note that the choice of a particular interconnection network depends on the application demands, technology supports and cost effectiveness.

9.4 SIMD (ARRAY PROCESSORS) INTERCONNECTION NETWORKS

SIMD array processor is mainly characterized by data routing network used in interconnecting the PEs. These networks are specified by **data routing functions** (DRFs). Each **routing function** f is a **bijection** (one-to-one or onto mapping) from system to system. This routing function is executed via interconnection network. Then PE_i copies the contents of its R_i register ($i \leftarrow 0, 1, \dots, N-1$) into the $R_{f(i)}$ register of $PE_{F(i)}$. This data routing operation occurs in all active PEs simultaneously. An inactive PE may receive data from another PE if a routing function is executed but it cannot transmit data. To pass data between PEs that are not directly connected in the network, the data must be passed through intermediate PEs by executing a sequence of routing functions through the interconnection network.

9.4.1 Static Versus dynamic Network – A tabular form comparison

Static Network	Dynamic Network
<ul style="list-style-type: none"> 1. They are composed of point -to-point connections. 2. The links between the two processors are passive. They do not change during program execution. 3. They provide fixed connection between the nodes of a distributed system. 4. They are classified as 1D, 2D, or 3D or as a hypercube. <p><i>Examples :</i> linear Array, Ring, star, mesh etc.</p>	<ul style="list-style-type: none"> 1. They are composed of switched channels. 2. The links between the two processors are active or dynamic. Switches are dynamic. 3. They provide dynamic connection and are used in shared-memory multiprocessors. 4. They are classified as either single stage or as multistage dynamic networks. <p><i>Examples :</i> Digital Bus, switches, crossbar etc.</p>

9.4.2 Cube Interconnection Networks

This has already been discussed in chapter-3 in detail.

9.4.3 Hypercube Interconnection Networks

This has already been discussed in chapter-3 in detail.

9.4.4 Mesh connected Illiac Network

Mesh connected network is implemented in the Illiac-IV, array processor. As it is implemented in Illiac-IV, it is also known as **Mesh connected Illiac Network**. It is considered as single stage recirculating network. This network consists of 64 PEs i.e., $N=64$. Each PE_i is allowed to send data to any one of PE_{i+1} , PE_{i-1} , PE_{i+r} and PE_{i-r} where $r = \sqrt{N}$ (For Illiac-IV, $r = 8$) in one circulation step through the network. Formally, the Illiac network is characterized by the following four routing functions :

$$\begin{aligned} R_{+1}(i) &= (i + 1) \bmod N \\ R_{-1}(i) &= (i - 1) \bmod N \\ R_{+r}(i) &= (i + r) \bmod N \\ R_{-r}(i) &= (i - r) \bmod N \end{aligned}$$

Where $0 \leq i \leq N - 1$. In practice, N is commonly a perfect square, such as $N = 64$ and $r = 8$ in the Illiac-IV network.

A reduced Illiac network can be shown as :

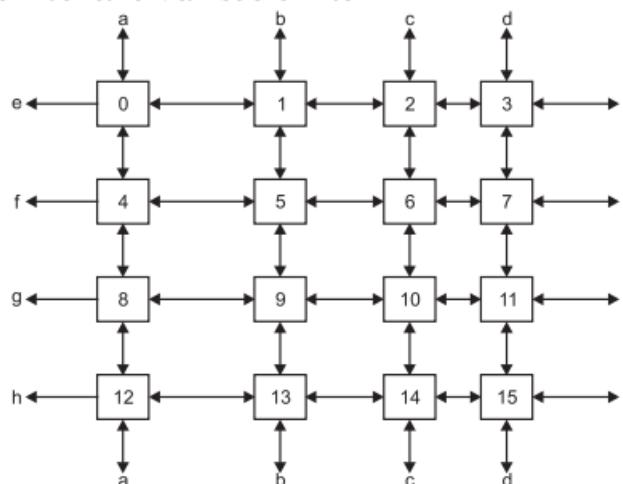


Fig. 9.3. Mesh connection.

In mesh network, each PE is connected to four adjacent PEs. In terms of permutation cycles, routing function can be expressed in **Horizontal** or **Vertical** manner. In **horizontal manner**, for example, the permutation cycles $(a\ b\ c)\ (d\ e)$ stand for permutation $a \rightarrow b$, $b \rightarrow c$, $c \rightarrow a$ and $d \rightarrow e$, $e \rightarrow d$ in circular fashion within each pair of parenthesis :

$$\begin{aligned} R_{+1} &= (0\ 1\ 2\ \dots\ N-1) \\ R_{-1} &= (N-1\ \dots\ 2\ 1\ 0) \end{aligned}$$

Vertically, the distance, r , shifting operations are characterized by the following two permutations with r cycles of order ' r ' each :

$$R_{+r} = \pi(ii + ri + 2r \dots i + N - r)$$

$$R_{-r} = \pi(i + N - r \dots i + 2ri + ri)$$

Say, $N = 16$, so $r = 4$. Here, we get two permutations, each with four cycles of order four each :

$$\begin{aligned} R_{+4} &= (0\ 4\ 8\ 12)\ (1\ 5\ 9\ 13)\ (2\ 6\ 10\ 14)\ (3\ 7\ 11\ 15) \\ &\& R_{-4} = (12\ 8\ 4\ 0)\ (13\ 9\ 5\ 1)\ (14\ 10\ 6\ 2)\ (15\ 11\ 7\ 3) \end{aligned}$$

The shifting operation in the cycle will be suspended if any PE required in the cycle is disabled. It means that cycle $(0\ 4\ 8\ 12)$ in above permutation R_4 will not be executed if one or more among PEO, PE4, PE8 and PE12 is disabled by masking.

9.5 SIMD PARALLEL ALGORITHMS

A parallel algorithm defines how a given problem can be solved on a given parallel computer i.e., how the problem is divided into sub problems, how the processors communicate and how the partial solutions are combined to produce the final result.

Next, we write a parallel sort algorithm using mesh—Cannon's and systolic approach.

9.5.1 Matrix Multiplication

(a) Sequential Code – Our classical matrix multiplication algorithm is :

```
for (i = 0 ; i < n ; i++)
    for (j = 0 ; j < n ; j++)
    {
        C[i][j] = 0;
        for (k = 0 ; k < n ; k++)
            C[i][j] = C[i][j] + a[i][k] * b[k][j];
    }
```

This algorithm requires n^3 multiplications and n^3 additions, leading to sequential time complexity of $O(n^3)$.

(b) Parallel code – Using Mesh implementation :

The most natural architecture for matrix operations is a 2D-mesh, where each node in mesh computes one element of the result array. The mesh connections will enable message to pass between adjacent nodes in the mesh simultaneously.

There are 2 methods of matrix multiplication for a mesh organization :

- (a) Cannon's algorithm.
- (b) Systolic approach.

I. Cannon's Algorithm—This algorithm uses a mesh of processors with wraparound connection, (a torus) to shift the A elements (or sub matrices) left & the B elements up. i.e., shown in Fig. S1.

All shifts are with wraparound. The steps of this algorithm are as follows :

S1 : Initially processor P_{ij} has elements a_{ij} and b_{ij} ($0 \leq i < n, 0 \leq j < n$).

S2 : Elements are moved from their initial position to an “aligned” operation. The complete i^{th} row of A is shifted i places

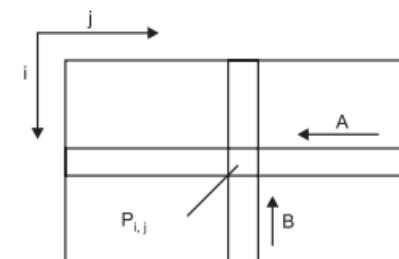


Fig. S1 Movement of A & B elements.

left & complete j^{th} column of B is shifted j places upward. This has the effect of placing the elements a_{ij+i} and the element $b_{i+j,j}$ in processor $P_{i,j}$. These elements are a pair of those required in accumulation of c_{ij} i.e.,

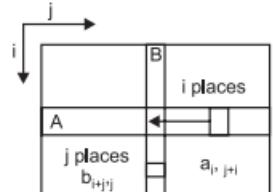


Fig. S2. Alignment of elements of A and B.

S3 : Each processor, p_{ij} multiplies its elements.

S4 : The i^{th} row of A is shifted one place left, & the j^{th} of B is shifted one place upward. This has the effect of bringing together the adjacent elements of A and B , which will also be required in the accumulation i.e.,

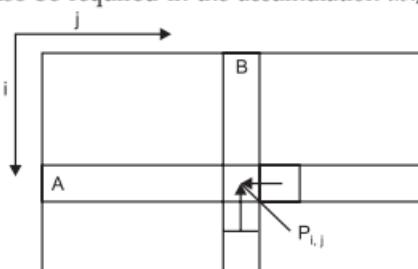


Fig. S4. One place shift of elements A and B.

S5 : Each processor, p_{ij} multiplies the elements brought to it & adds the results to the accumulating sum.

S6 : Steps (4) & (5) are repeated until final result is obtained. ($n-1$ shifts with n rows & n -columns of elements).

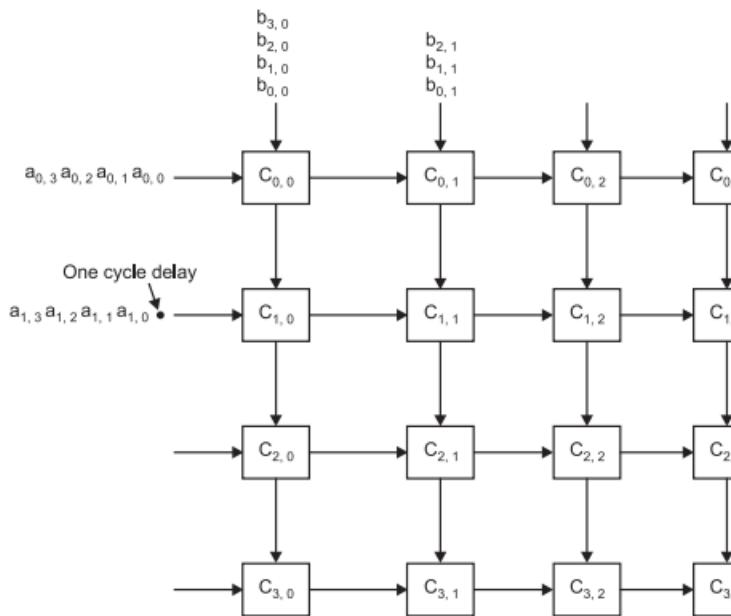
Additional storage for partial results is unnecessary with this algorithm.

Analysis—Each sub matrix multiplication requires m^3 multiplications and m^3 additions. Hence, with $S-1$ shifts,

$$t_{\text{comp}} = 2s m^3 = 2 m^2 n \\ \text{or time complexity is } O(m^2 n)$$

9.5.2 2D-Pipeline—Systolic Array

The word systolic has been borrowed from the medical field just as the heart pumps blood, information is pumped through, a systolic array in various directions, and at regular intervals. In 2D systolic array, information, is pumped from left to right and from top to bottom. The information meets at internal nodes where processing occurs. The same information passes onward (left to right or downward). Fig. below show a systolic array used to multiply two (4x4) matrices, A and B. The elements of A enter from left and elements of B enter from top.



The final product-terms of C will be held in processors. A suitable numbering of PEs is done, starting $(0, 0)$ in top left corner. Each processor $P_{i,j}$ repeatedly performs the same algorithm.

```

recv (& a, pi, j-1); /* receive from left */
recv (& b, pi-1, j); /* receive from right */
c = c + a * b; /* accumulate value for ci,j */
send (& a, pi, j+1); /* send to right */
send (& b, pi+1, j); /* send downwards */

```

Which accumulates required summations.

$$\begin{aligned} \text{So, } C_{0,0} &= a_{0,0} b_{0,0} + a_{0,1} b_{1,0} + a_{0,2} b_{2,0} + a_{0,3} b_{3,0} \\ C_{0,1} &= a_{0,0} b_{0,1} + a_{0,1} b_{1,1} + a_{0,2} b_{2,1} + a_{0,3} b_{3,1} \end{aligned}$$

A similar computation occurs with each of the other processors.

9.5.3 Sorting in Parallel

Various sorting algorithms have been presented here. Some of them are as follows :

1. Enumeration/Rank sort.
2. Compare and Exchange sort.
3. Odd–Even Transposition Sort.
4. Bitonic Merge Sort.

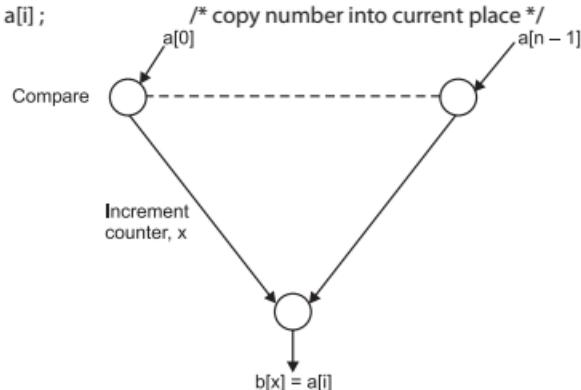
We shall discuss them one by one.

I. Enumeration Sort/Rank sort :

In rank sort, the number of elements that are smaller than each selected number, is counted. This count provides the position of the selected number in the list i.e., it's "rank" in the list.

Suppose there are n numbers stored in an array. $a[0] \dots a[n - 1]$. First $a[0]$ is read & compared with each of the other numbers $a[1] \dots a[n - 1]$, recording the number of numbers less than $a[0]$. Suppose this number is x . This is the index of the location in the final sorted list. The number $a[0]$ is copied into the final sorted list $b[0] \dots b[n - 1]$, at location $b[x]$. Then the next number $a[1]$, is read & compared with each of other numbers, $a[0], a[2] \dots a[n - 1]$ & so on. Comparing one number against $n - 1$ other numbers requires at least $n - 1$ steps, if performed sequentially. Doing this with all n numbers requires $n(n - 1)$ steps, an overall sequential sorting time complexity of $O(n^2)$. The actual sequential code might look like :

```
for (i = 0 ; i < n ; i++) { /* for each number */
    x = 0;
    for (j = 0 ; j < n ; j++)
        If (a[i] > a[j]) x++;
    } b[x] = a[i];
    /* copy number into current place */
```



Here, $(n - 1)$ processors are used to find the rank of one number and with n numbers, $(n - 1) * n$ processors or n^2 processors are needed. A single counter is needed for each number. Incrementing the counter is done sequentially and requires a maximum of n -steps, including one step to initialize the counter. Therefore, the total number of steps would be given by $1 + n$.

So, Rank sort can sort in $O(n)$ with n processors or in $O(\log n)$ using n^2 processors.

II. Compare-and-Exchange sorting

In this algorithm two numbers, say A and B , are compared. If $A > B$, A and B are exchanged, else the contents of the locations remain unchanged.

Sequential code :

```
if (A > B)
    {temp = A;
     A = B ;
     B = temp;
    }
```

with 2 for-loops each iterating for n -steps.

Using n processors : Suppose we have n processors. One processor would be allocated to one of the numbers & could find the final index of one number in $O(n)$ steps. With

all processor operating in parallel, the parallel time complexity would also be $O(n)$. In **for all loop** notation, the code would look like :

```
for all (i = 0 ; i < n ; i++) /* count each no, in parallel */
    x = 0
    for (j = 0 ; j < n ; j++)
        if (a[i] > a[j]) x++;
    b[x] = a[i]; /* copy number into correct place */
```

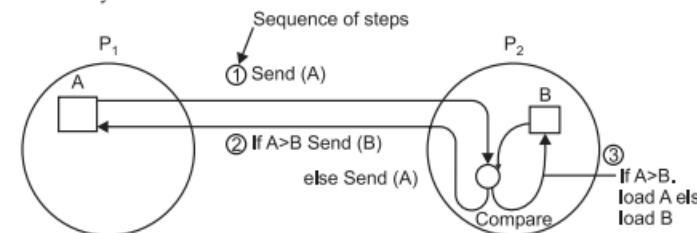
This parallel time complexity $O(n)$ is better, than any sequential algorithm. and we can even do better if we have more processors.

Using n^2 Processors : Comparing one selected number with each of the other number(s) in the list can be performed using multiple processors. F or e.g. a structure shown in Figure could be used.

Compare-and-exchange is well suited to a **message passing system** – Suppose the compare & exchange is between two number(s). A and B , where A is held in process P_1 and B is held in process P_2 . One simple way of implementing the compare and exchange is for P_1 to send A to P_2 , which then compares A and B and sends back B to P_1 if A is larger than B otherwise it sends back A to P_1 . It is not strictly necessary to send A back to P_1 since it already has A , but it does make it easier to have the same number of sends/receive in either case. The code could be :

Process 1	Process 2
send (&A, P ₂);	recv (& A, P ₁);
recv (&A, P ₂);	If (A > B){
	send (& B, P ₁);
	B = A;
	} else
	send (& A, P ₁);

Alternative way :



Here, P_1 sends A to P_2 and P_2 sends B to P_1 . Then both processes perform compare operations. P_1 keeps the smaller of A and B and P_2 keeps the larger of A and B . The code for this approach is :

Process P ₁	Process P ₂
send (& A, P ₂);	recv (& A, P ₁);
recv (& B, P ₂);	send (& B, P ₁);
if (A > B) A = B;	if (A > B) B = A;

i.e., process P_1 performs the send () first and process P_2 performs the recv () first to avoid deadlock.

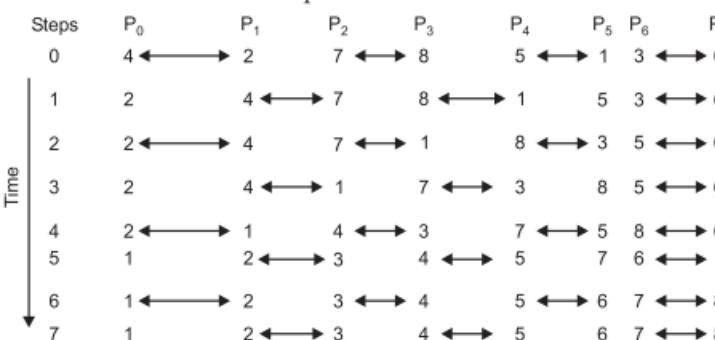
III. Odd–Even Transposition Sort

The odd–Even Transposition sort is designed for the processor array model in which PEs are organized into a 1D-mesh. It operates in 2 alternating phases – **an even phase & an odd phase**.

In even phase – even-numbered processes exchange numbers with their right neighbor.

In odd phase – odd – numbered processes exchange numbers with their right neighbor.

This sort normally would not be discussed for sequential programming as it has no particular advantage over normal bubble sort. However, the parallel implementation reduces time complexity to $O(n)$. Odd–even transposition sort applied to sequence of 8 numbers. one number stored in each process is shown below :



I. In even phase, we have the compare & exchanges: $P_0 \leftrightarrow P_1$, $P_2 \leftrightarrow P_3$ etc. Using the same form of compare & exchange as written earlier, the code could be :

```

 $P_i, \quad i = 0, 2, 4 \dots, n - 2$  (even)
recv (& A,  $P_{i+1}$ );
send (& B,  $P_{i+1}$ );
if ( $A < B$ )  $B = A$ ;
&  $P_i, \quad i = 1, 3, 5 \dots, n - 1$  (odd)
send (& A,  $P_{i-1}$ ); /* even phase */
recv (& B,  $P_{i-1}$ );
if ( $A < B$ )  $A = B$ ; /* Exchange */

```

Where the number stored in P_i (even) is B & the number stored in P_i (odd) is A .

II. In odd phase, we have compare & exchanges :

```

 $P_1 \leftrightarrow P_2, P_3 \leftrightarrow P_4$  etc. this could be codes as :
 $P_i, \quad i = 1, 3, 5 \dots n - 3$  (odd)       $P_i, \quad i = 2, 4, 6, \dots N - 2$  (even)
send (& A,  $P_{i+1}$ );                            recv (& A,  $P_{i-1}$ ); /* odd phase */
recv (& B,  $P_{i+1}$ );                            send (& B,  $P_{i-1}$ );
if ( $A > B$ )  $A = B$ ;                            if ( $A > B$ )  $B = A$ ; /* exchange */

```

In both cases, odd-numbered processes execute their send () routine first and even-numbered processes execute their recv () routine first.

Combining,

```

 $P_i, \quad i = 1, 3, 5, \dots n - 3$  (odd)       $P_i, \quad i = 2, 4, 6, \dots n - 2$  (even)

```

```

send (& A, Pi-1) ;
recv (& B, Pi-1) ;
if (A < B) A = B ;
if (i <= n - 3) {
    send (& A, Pi+1) ;
    recv (& B, Pi+1) ;
    if (A>B) A= B;
}
recv (& A, Pi+1) /* even phrase */
send (& B, Pi+1) ;
if (A < B) B = A ;
if (i >= 2) /* odd phase */
    recv (& A, Pi-1) ;
    send (& B, Pi-1) ;
if (A > B) B = A ;
}

```

IV. Bitonic mergesort

It was introduced by Batcher in 1968 as a parallel sorting algorithm.

What is a Bitonic Sequence? The basis of bitonic mergesort is a bitonic sequence, a list having specific properties that will be utilized in sorting algorithm. A monotonic increasing sequence is a sequence of increasing numbers. A bitonic sequence has two sequences, one increasing & one decreasing. Formally, a bitonic sequence is a sequence of numbers, $a_0, a_1, a_2, a_3 \dots a_{n-2}, a_{n-1}$, which monotonically increases in value, reaches a single maximum and then monotonically decreases in value. e.g., $a_0 < a_1 < a_2, a_3 \dots a_{i-1} < a_i > a_{i+1} \dots a_{n-2} > a_{n-1}$ for some value of i ($0 \leq i < n$).

A sequence is also bitonic if the preceding can be achieved by shifting the numbers cyclically (left or right). Bitonic sequences are :

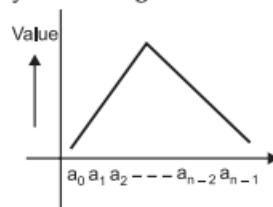


Fig. (a) Single maximum

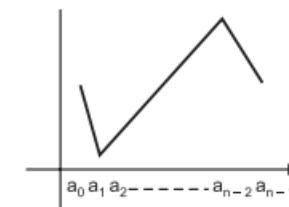
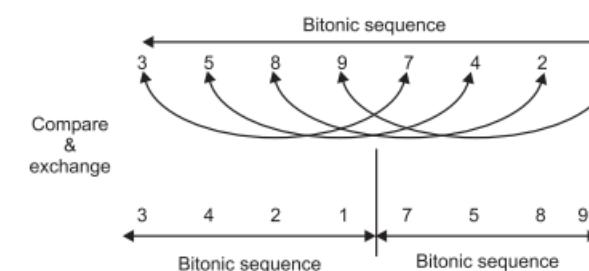


Fig. (b) Single maximum & single minimum

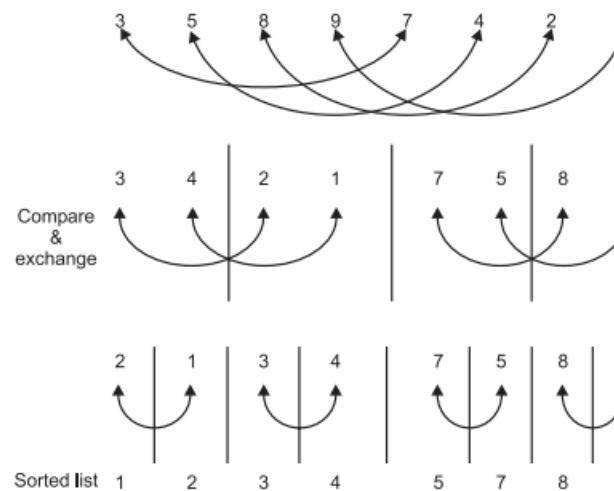
Note that a bitonic sequence can be formed by sorting two lists, one in increasing order & one in decreasing order & concatenating the sorted lists.

The 'special' characteristic of bitonic sequences is that if we perform a compare-and-exchange operation on a_i with $a_{i+n/2}$ for all i ($0 \leq i \leq n/2$) where there are n numbers in the sequence, we get two bitonic sequences, where the numbers in one sequence are all less than the numbers in other sequence. For e.g., : starting with bitonic sequence. 3, 5, 8, 9, 7, 4, 2, 1 & Performing a compare & exchange, a_i with $a_{i+n/2}$ we get the following sequences :

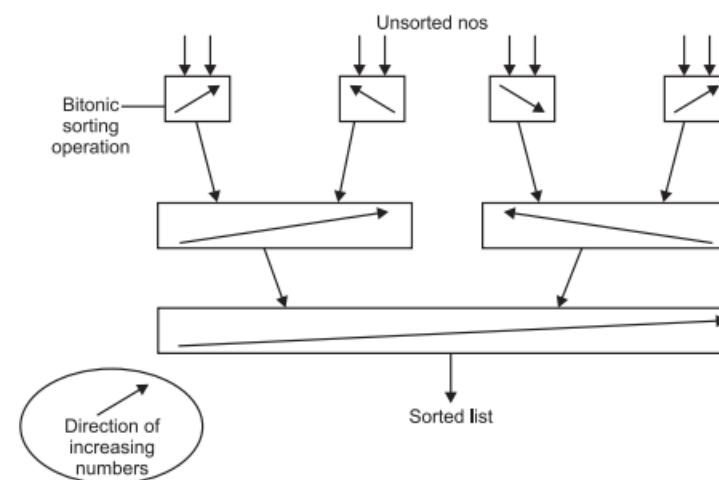
The compare-and-exchange operation moves the smaller numbers of each pair to the left sequence and the larger numbers of the pair to the right sequence both sequence being bitonic sequences. If this compare and exchange operation is performed recursively then we will get the fully sorted



list i.e.,

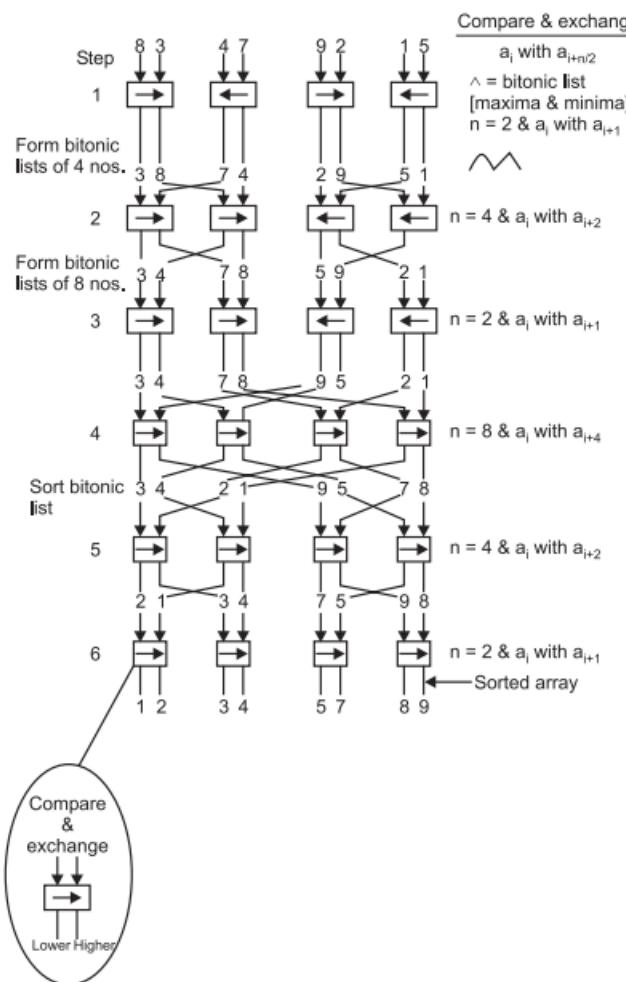


Sorting : To sort an unordered sequence, sequences are merged into larger bitonic sequences, starting with pairs of adjacent numbers. By a compare-and-exchange operation, pairs of adjacent numbers are formed into increasing sequences and decreasing sequences, pairs of which form a bitonic sequence of twice the size of each of the original sequences. By repeating this process, bitonic sequences of larger and larger lengths are obtained. In the final step, a single bitonic sequence is sorted into a single increasing sequence. i.e.,



We consider now the 8 numbers & sort them using bitonic Merge Sort :

Say, $a = [8 \ 3 \ 4 \ 7 \ 9 \ 2 \ 1 \ 5]$



These 6 steps are divided into 3-phases :

Phase 1 (Step 1) : Convert pairs of numbers into increasing/decreasing sequences & hence into 4-bit bitonic sequences.

Phase 2 (Step 2 & 3) : Split each 4-bit bitonic sequence into two 2-bit bitonic sequences, higher sequence at center sort each 4-bit bitonic sequence at center. Sort each 4-bit bitonic sequence increasing/decreasing sequences & merge into 8-bit bitonic sequence.

Phase 3 (Steps 4/5/6) : Sort 8 bit bitonic sequence with $n = 2^k$, there are k -phases, each of 1, 2, 3 ... k steps. Hence, total number of steps is given by

$$\text{Step} = \sum_{i=1}^k i = \frac{k(k+1)}{2} = \frac{\log n(\log n + 1)}{2}$$

$$= O(\log^2 n)$$

\therefore Time complexity is $O(\log^2 n)$ using n processors and with 1 processor for each number.

9.5.4 Fast Fourier Transform (FFT)

The Fourier transform sequence is given as -

$$A = [A_0, A_1, A_2, \dots, A_{N-1}]$$

where

$$A_1 = P_a [W_N]$$

and

$$P_a [X] = \sum_i a_i X_i, \text{ degree polynomial}$$

where, W_N = Principal N^{th} root of 1.

If this can be computed on a sequential machine, it will require N^2 time units. Using hypercube array structure and using divide and conquer strategy computation time is reduced. The total time required is $[N \log N]$.

SIMD algorithm for performing FFT are presented here. Let $S[k]$, where

$k = 0, 1, \dots, M - 1$ be M samples of time function.

The discrete fourier transform of $S[k]$ defined as

$$x(j) = \sum_{k=0}^{M-1} S(k) \cdot W_j^k, j = 0, 1, \dots, M-1$$

and

$$W = e^{2\pi i/M}, i = \sqrt{-1}$$

Consider SIMD machine perform an M point FFT of the discrete signal sequence $\{S(M), 0 \leq M \leq M-1\}$. SIMD machine requires $N/2$ PEs to perform the same.

This algorithm is a parallel implementation of decimation in frequency [D/F] technique. Consider $M = 16$ sampling points D/F algorithm divides the input sequence

$\{S(m)\}$ into $\{f(m)\}$, and $\{g(m)\}$, where $f(m) = S(m)$ and $g(m) = S\left(m + \frac{m}{2}\right)$ for $m = 0, 1, \dots, \frac{M}{2}-1$.

Diagrammatical representation of this is :

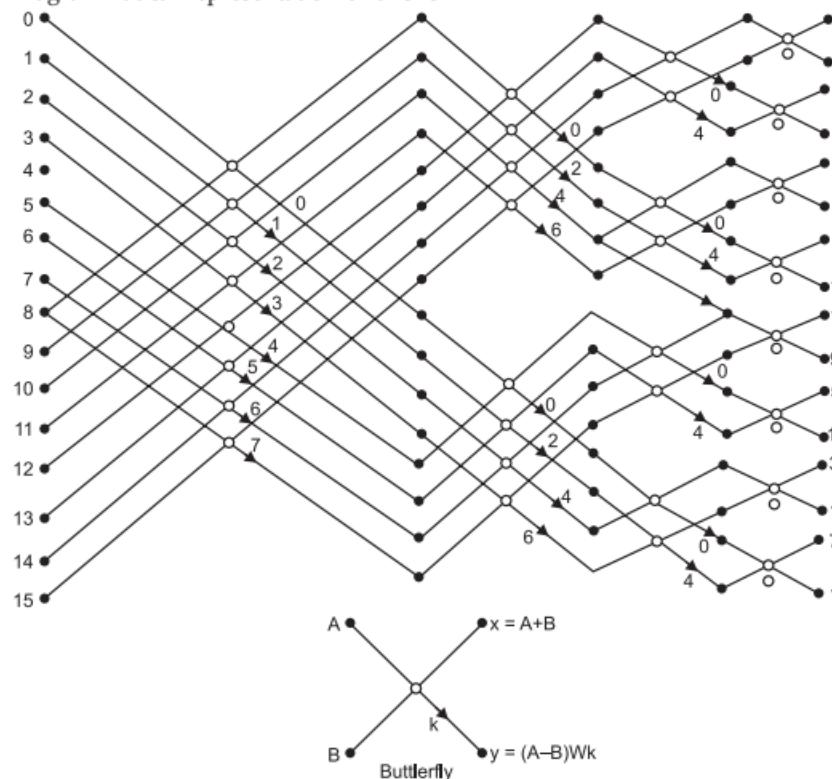


Fig. 9.4. Operations in a 16-point FFT Based on the Decimation – In Frequency Algorithm.

For parallel FFT algorithm, PE_i initially contains $S(k)$ and $S(K + M/2)$, where $0 \leq k \leq N$.

In serial method, $\log_2 M$ stages of computations are needed. At each stage $\frac{M}{2}$ butterfly operations are executed as shown in diagram.

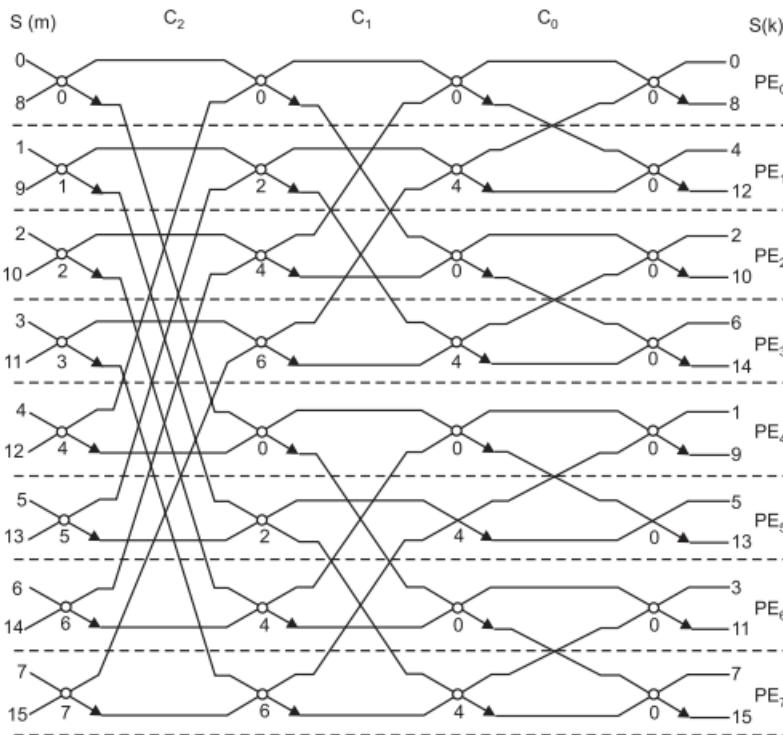


Fig. 9.5. Computation of 16-bit Point FFT in an SIMD machine with 8 PEs.

At each stage, the items being paired in butterfly at stage k , where $0 \leq k \leq \log_2 M$ are those whose indices differ in the $(\log_2 M - k - 1)^{\text{th}}$ bit position of binary representation. Because of this difference cube interconnection network used for specifying the interprocessors data transfers are required for the FFT algorithm. This network consists of n routing functions C_i for $0 \leq i \leq \log_2 N$.

SUMMARY

In this chapter we have studied about SIMD/Array Processors. Their architecture has been explained. Some SIMD parallel algorithms on matrix multiplication and sorting have been given and analysed too.

MULTIPLE CHOICE QUESTIONS (MCQS) WITH ANSWERS

1. SIMD stands for :
 - (a) Single instructions molded data.
 - (b) Single instruction multiple data.
 - (c) Simple instruction multiple data.
 - (d) None of the above.
2. An example of configuration-I of Array processors is :

(a) BSP	(b) Illiac IV
(c) IBM 360	(d) None of the above

3. B S P stands for :
 - (a) Burroughs Scientific Processor
 - (b) Basic Scientific Processor
 - (c) Beginner's Scientific Processor
 - (d) None of the above.
4. SIMD machines choose which sort of operation mode :

(a) Asynchronous	(b) Direct
(c) Synchronous	(d) None of the above
5. Centralized control is used by :

(a) SIMD machines	(b) SISD machines
(c) MISD machines	(d) MIMD machines
6. SIMD machines use :

(a) Circuit switching.	(b) Packet switching.
(c) Integrated switching	(d) None of the above
7. MIMD machines use :

(a) Circuit switching	(b) Packet switching
(c) Integrated switching	(d) None of the above
8. A mesh is an example of :

(a) Static network	(b) Dynamic network
(c) Switch	(d) None of the above
9. Rank sort can sort in $O(n)$ time with :

(a) 1 processor	(b) 2 processor
(c) n processor	(d) All of the above
10. Cannon's Algorithm requires a time of :

(a) $O(n)$	(b) $O(n^2)$
(c) $O(mn)$	(d) $O(m^2n)$

:: ANSWERS ::

- | | | | |
|--------|--------|---------|--------|
| 1. (b) | 2. (b) | 3. (a) | 4. (c) |
| 6. (a) | 7. (b) | 8. (a) | 9. (c) |
| | | 5. (a) | |
| | | 10. (d) | |

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

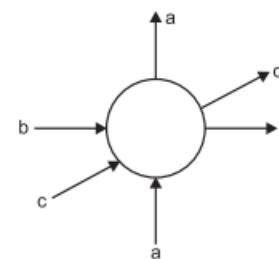
1. Following is the cell whose inputs and outputs are indicated :
Where $d = a * b + c$

Use this cell to multiply two (3×3) matrices. Also calculate how many cells are required to get the result

[KUD, B.E. (CSE) 8th Sem ; 1996]

- Ans.** The following is a cellular array for pipelined multiplication of two dense matrices:

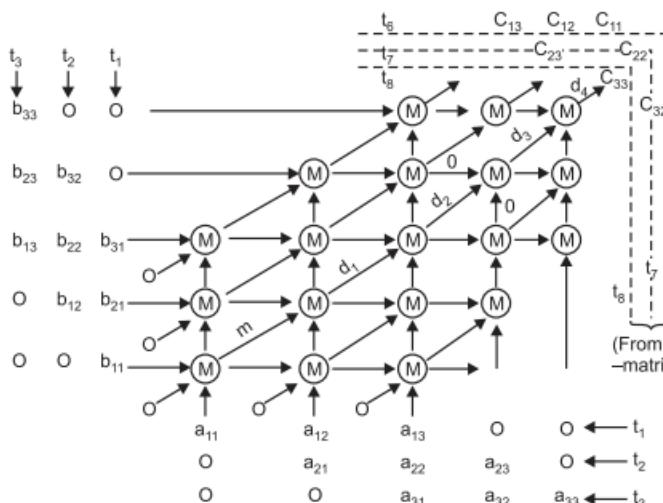
$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}_{3 \times 3}$$



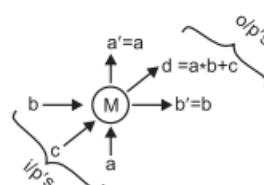
$$B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}_{3 \times 3}$$

$$\therefore C = A \times B = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}_{3 \times 3}$$

$$C = A^*B =$$



Where



M is a cell having 3 inputs a , b and c and 3 outputs—

$$a' = a$$

$$b' = b \text{ and } d = a * b + c$$

Working : The input matrices, A & B , to be multiplied are fed into the array in the horizontal & vertical directions. 3-clk periods are required for inputting the matrix entries :

One row at a time for A -matrix

and one column at a time for B -matrix

\therefore There are 3-pairs like this, so 3-clk periods are required.

Dummy Zero inputs are marked at unused input lines. Don't care conditions at the output lines are left blank.

i.e., During clk-period t_1 , R_1 & C_1 are fed to get C_{11} i.e.,

$$\begin{aligned} m &= (a_{11} * b_{11} + O) \\ a_{12} * b_{11} + O &= a \\ d_1 &= a_{12} * b_{21} + a_{11} b_{11} \\ d_2 &= a_{13} * b_{31} + a_{12} b_{21} + a_{11} b_{11} \\ d_3 &= a_{13} * b_{31} + a_{12} b_{21} + a_{11} b_{11} \\ d_4 &= a_{13} b_{31} + a_{12} b_{21} + a_{11} b_{11} = c_{11} \text{ (1st row of } c\text{-matrix)} \end{aligned}$$

This process is repeated 3-times, till whole of multiplication is over. Finally

we get c_{11} at t_6

We get c_{12} at t_6

We get c_{13} at t_6

We get c_{21} at t_6

We get c_{31} at t_6

&

c_{22}, c_{32}, c_{23} at t_7

&

c_{33} at t_8

In general, $(3n-1)$ CLK periods are required to complete the multiply process.

Construction of hardware : Fast latches (registers) are used at all input-output terminals & all interconnecting paths in the array pipeline. All latches are synchronously controlled by same CLK. M-cells are "basic-building blocks" in the array.

Limitations : When the matrix size becomes too large, this global-array approach will pose a serious problem for monolithic chip implementation because of density and I/O packaging constraints.

2. Repeat Q1 above to multiply two (2×2) matrices now.

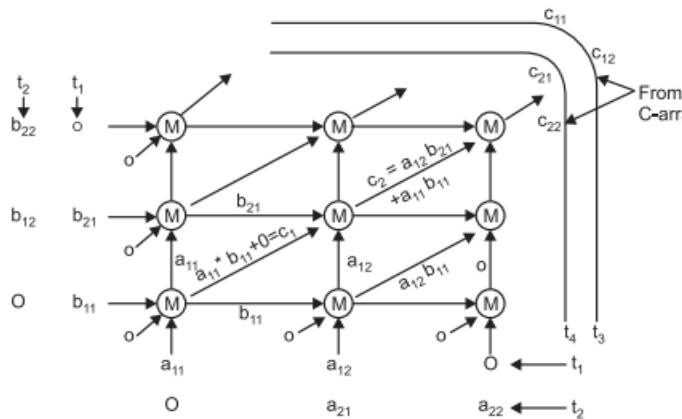
Ans. Now, A-matrix is :

$$A = \begin{bmatrix} \overrightarrow{a_{11}} & \overrightarrow{a_{12}} \\ \overrightarrow{a_{21}} & \overrightarrow{a_{22}} \end{bmatrix}_{2 \times 2}$$

$$B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}_{2 \times 2}$$

$$\therefore C = A * B = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

$$= \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$



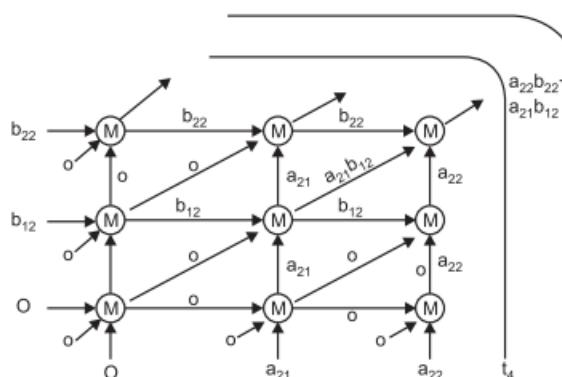
$$\begin{aligned} \therefore c_1 &= a_{11} b_{11} + \phi \text{ (see fig)} \\ \therefore c_2 &= a_{12} b_{21} + a_{11} b_{11} = c_{11} \text{ (from cell-rule)} \end{aligned}$$

This process is continued & 1 row & 1 column at a time are sent for evaluation.
i.e., output c_{11} is available after 1st clk period.

Now, when 2nd -clk period is given :

Then R_2 and C_2 are fed :

i.e.,



Similarly, we can show for C_{12} & C_{21} also.

3. It is desired to compute this expression :

$$\begin{aligned} S &= A_1 \times B_1 + A_2 \times B_2 + \dots + A_{32} \times B_{32} \\ &= \sum_{i=1}^{32} A_i B_i \end{aligned}$$

- (a) If a sequential/serial computer is used, find out the minimum compute time.
- (b) If an SIMD computer with 8 PEs ($PE_0, PE_1 \dots PE_7$) is used, each of which is connected by a bi-directional circular ring, inter-PE communications taking one time unit, then also find out the minimum compute time. The operands A_i and B_i are initially stored in $PE_{i \bmod 8}$ for $i = 1, 2, \dots, 32$.

Each PE can add or multiply at different times.

- Ans. (a) The algorithm to compute the expression in a serial computer is shown below :

```

 $S = A_1 \times B_1$ 
For  $i = 2$  to  $32$  Do
 $S = S + A_i * B_i$ 
End do

```

There are 32 multiply operations and 31 add operations.

∴ Number of time units needed

$$\begin{aligned}
 &= (32 \times 4) + (31 \times 2) \\
 &= 128 + 62 \\
 &= 190 \text{ cycles}
 \end{aligned}$$

- (b) The algorithm for SIMD machine is as follows :

```

Parfor  $j = 1$  to  $8$  Do
 $S(j) = A_{ij} \times B_{ij}$  // 1 multiply task
For  $i = 2$  to  $4$  Do
 $S(j) = S(j) + A_{ij} * B_{ij}$  // 1 multiply, 1 add.
End do
 $S(j) = S(j) + S(j + 1)$  // 1 Routing, 1 add
 $S(j) = S(j) + S(j + 2)$  // 2 Routing, 1 add
 $S(j) = S(j) + S(j + 4)$  // 4 Routing, 1 add
End do

```

In this code, there are 4 multiply operations, 6 add operations and 7 routing operations.

$$\begin{aligned}
 \text{Time needed} &= (4 \times 4) + (6 \times 2) + (7 \times 1) \\
 &= 16 + 12 + 7 \\
 &= 28 + 7 \\
 &= 35 \text{ cycles.}
 \end{aligned}$$

EXERCISE QUESTIONS

1. Write short notes on :
SIMD parallel algorithms.[GGSIPU, B. Tech. 7th sem; Dec. 2007 & Dec. 2004]
2. Explain with diagram a :
SIMD computer. [KUD, BE -8th sem; 1996]
3. (a) Draw a schematic of Illiac-IV, the SIMD computer and explain where data routing and manipulating functions are done.
(b) Draw a single-stage recalculating mesh connected network for $N = 16$ PEs.
Give the four permutations $R + 1, R - 1, R + 4, R - 4$ and the upper bound on steps to route data from PE i to PE j . [KUD, BE (CSE) ; 8th sem; 1996]
4. Explain sorting on SIMD model with suitable examples. What is it's time complexity. [KUD, BE (CSE); 8th sem; 1995 and 1997]
5. Write short notes on :

- Fast Fourier Transform. [KUD, BE(CSE); 8th sem; 1997]
6. Design an odd-even transposition sort on SIMD -CC model. Use odd-even transposition sort to sort these eight values :
(5, 8, 3, 2, 4, 6, 4, 1). [KUD, BE (CSE); 8th sem; 1996]
7. Prove that matrix multiplication on the 2D mesh SIMD model requires $\Omega(n)$ or for large values of n, approximately $s >= 0.35n$, data routing steps.
[DU, M.E (CSE); DCE, 2nd sem; 2002]
8. Explain basic architectural configurations of SIMD processors. What are the network design decisions for inter PE -communications with respect to interconnection network ? [Pune Univ., B.E; 2nd sem; May 2004]
9. Show that multiplication of ($n \times n$) matrix on a linear SIMD architecture is n times faster than that of SISD architecture. [Pune Univ, BE-2nd sem; May 2001]
10. Discuss any one of the parallel sorting algorithm with suitable architecture for doing that. [Pune Univ., BE-2nd sem; Dec 2001]



CHAPTER

10 PARALLEL ALGORITHMS AND PROGRAMMING

10.0 INTRODUCTION

A **parallel algorithm** is one which can be executed a piece at a time on many different processing devices and put back together again at the end to get the correct result. We describe in this chapter various algorithms that are mainly concerned with specification, design and analysis of MIMD (Multiple Instruction stream–Multiple Data stream) computers.

10.1 NEED OF PARALLEL PROGRAMMING

A Parallel program is a program in which there is partitioning and distribution of main task (function) and/or data. That is, there is a workload distribution and/or data load distribution. When the work load is distributed in the form of task, the parallelism is called as **functional parallelism**. When the data structure is partitioned and distributed among multiple processors for parallel executing then the parallelism is called as **data parallelism**. Thus, a parallel program may have either functional parallelism or data parallelism or a combination of both.

The parallel computing tremendously increases the throughput of the system. Today, a large number of applications such as scientific computations, image and signal processing, database and data mining etc. are implemented with the help of parallel computing. Parallel computing needs the supporting hardware architecture i.e. a parallel architecture. These parallel architectures allow concurrent execution of multiple processes or program units. Therefore, if this parallel architecture is to be utilized with its full potential then one must develop parallel program. Parallel program development is a special field in the computer science. There is a wide scope for research in this area.

10.2 CHARACTERISTICS OF PARALLEL ALGORITHMS

The following are the important characteristics of parallel algorithms :

1. **Deterministic v/s non-deterministic** : Only deterministic algorithms are implementable on real machines.
2. **Computational granularity** : Granularity decides the size of data items and program modules used in computation. In this sense, we can typically classify algorithms as fine grain, medium-grain or coarse-grain.
3. **Parallelism profile** : The distribution of the degree of parallelism (DOP) in an algorithm reveals the opportunity for parallel processing. Effectiveness of parallel algorithms is often affected.
4. **Communication patterns and synchronization requirements** : Communication

patterns address both memory access and interprocessor communications between processors or PEs. The patterns can be static or dynamic, depending on the algorithms. Please note that static algorithms are more suitable for SIMD or pipelined machines while dynamic algorithms are more suitable for MIMD machines. Also note that the synchronization frequency often affects the efficiency of an algorithm.

5. **Uniformity of the operations :** It refers to the types of fundamental operations to be performed. Please note that if the operations are uniform across the data-set then SIMD processing or pipelining is more desirable as the efficiency is improved. Randomly structured algorithms are more suitable for MIMD machines.
6. **Memory requirements and data structures :** When solving large scale problem, the data sets may require huge memory space. Data structures chosen and data movement patterns in the algorithms can affect the efficiency of a program. Both time and space complexities are key measures of granularity of parallel algorithms.

10.3 PARALLEL PROGRAMMING TECHNIQUES

There are two basic approaches in parallelizing the program and they are –

- (ii) Explicit Parallel programming
- (iii) Implicit Parallel programming

We discuss these one by one.

I. Explicit Parallel Programming

In this technique, the programmer develops a parallel program by explicitly specifying the parallelism. The programmer has to write down the parallel algorithm to solve given problem. There are three different techniques which can be used by the programmer to explicitly parallelize the program. The techniques are :

- (a) Use of specific programming languages which support parallel programming.
- (b) Use of compiler directives for library function calls.
- (c) Use of automatic parallelization tools that analyze the program and find parallelism.

II. Implicit Parallel Programming

In this technique, the programmer is relieved from parallelizing the program but it is the compiler which plays an important role. Here, the compiler is responsible for extracting the parallelism in the program. There is no burden on the programmer to know about the parallel programming languages and also the underlying parallel architecture.

We shall be considering only the explicit programming approach here.

10.4 PARALLEL PROGRAMMING MODELS

A parallel programming model is used by the application programmer to develop parallel program. Basically, a programming model specifies how the processes or parts of the parallel program communicate with each other and what synchronization techniques are available to coordinate their activities. Please note that a programming model provides a paradigm to the programmer for sharing, communication and synchronization among process of a parallel program.

There are **three parallel programming models for explicit parallel programming**.

The models are :

- (a) Message passing programming.
- (b) Shared memory programming.
- (c) Data parallel programming.

Which model to select ?

For parallel programming, one has to consider the underlying parallel architecture type and the model of interprocessor communication. In parallel architecture, the interprocessor communication takes place using either shared memory or distributed memory. **Please note here that the development of parallel program depends on whether the parallel architecture has shared or distributed memory and accordingly the programming model is selected.**

10.4.1 Message passing programming

This programming model is applicable for distributed memory parallel architecture in which each PE has local memory but there is no global shared memory. The application programmer uses this model for parallel program development for underlying parallel architecture with distributed memory. It is assumed that a program consists of multiple processes with local variables. These processes communicate with each other by **explicit coding**. The programmer explicitly puts 'send' and 'receive' commands in the source code. We can define the message passing programming model as :

" Multiple cooperating processes having only local variables "

Or

" Interprocess communication by use of ' send ' and ' receive ' messages.

Or

" Synchronization between the processes, that is, each ' send ' command must have corresponding matching ' receive ' command.

Message passing programming may be synchronous or asynchronous

I. Synchronous Message Passing

It requires synchronization between sending message and receiving message. In this scheme, the command '**Blocked Send**' is explicitly used for sending message to another processor and the sending processor waits till the receiving processor has not received the message. After proper reception, the sending processor can continue its operations. This scheme is used when the data or messages are to be send in a **proper order** and to ensure that it has reached processor.

To send data, the programmer uses '**Blocked Send**' command in the source code of the sending processor. There must be a corresponding, matching '**Blocked Receive**' command in the source code of the receiving or distinction processor. Such a 'send' is called as **synchronous send** and such a 'receive' is called as **synchronous receive**. Such a system is known as **synchronous message passing system**.

The general format of 'Block Send' command is :

Blocked Send	(destination 'E address,	Variable-name ,	Size or message)
--------------	---	----------------------------	-----------------	--------------------	---

Please note that here the address of the destination processor indicates the processor to which the message or data is to be send. Variable (data) name or message indicates the starting memory address of the data or message on the source or sending processor. The size (or message length) indicates the number of bytes to send.

Correspondingly, we have a general format of 'Blocked receive' also. It is given below:

Blocked Receive	(source , Variable-name , Size)
	'E address	or message	

Note here that the address of source processor (PE) tells the destination processor about the source of the message. The message or variable (data) name indicates the memory address on the destination where the received data variable will be stored. The size (or message length) indicates how many bytes will be received.

II. Asynchronous Message Passing

In this scheme, a simple 'Send' command is used explicitly by the user for sending data. The programmer puts 'Send' command in the source code of the sending processor. The sending processor sends data and continues its processing further. It does not wait for ensuring about data reception. Hence, this 'Send' is also called as a **non-blocking send**. There must be corresponding, matching 'Receive' command also, in the code of the receiving process.

Please note that the 'Send' and 'Receive' are used if ordering is not important and the system hardware reliably delivers messages. If a message passing system's hardware and O.S. guarantee that messages sent by 'Send' are delivered correctly in the right order to the corresponding 'Receive' then there will be **no need** for a Blocked send and Blocked Receive in that system. Such a system is called as an **asynchronous message passing system**.

The general format of 'Send' command is :

$$\text{Send} \left(\begin{array}{l} \text{address} \\ \text{of destination} \\ \text{PE} \end{array} , \begin{array}{l} \text{variable-name} \\ \text{or} \\ \text{message} \end{array} , \begin{array}{l} \text{Size} \\ , \\ , \end{array} \right)$$

The arguments of this commands are same as that of 'Blocked Send' command.

The general format of 'Receive' command is :

$$\text{Receive} \left(\begin{array}{l} \text{address} \\ \text{of destination} \\ \text{PE} \end{array} , \begin{array}{l} \text{variable-name} \\ \text{or} \\ \text{message} \end{array} , \begin{array}{l} \text{Size} \\ , \\ , \end{array} \right)$$

The arguments of this command are same as that of 'Blocked Receive' command.

We are in a position to solve a problem now.

Example. It is desired to add all the components of an array $A[i]$ for $i = 1, N$, where N is the size of the array. Write a program for message passing multicomputer assuming that only 2 PEs (or CEs) are available.

Ans. Since there are 2 PEs (or CEs) so we can assign them a task to each ; if N is even.

Task 1 : Add $A[1], \dots, A[N/2]$ to one PE

Task 2 : Add $A[N/2 + 1] \dots, A[N]$ to second PE.

The procedure for the two CEs is as follows :

PE – 0

```
Read A[i] for i = 1, ..., N
Send to PE1 A[i] for i = (N/2 + 1), ..., N
Find Sum 0 = Sum of A[i], for i = 1, ..., N/2
Receive from PE 1, Sum 1 computed by it
Sum = Sum 0 + Sum 1
Write Sum
```

Similarly, we write for PE – 1 :

```
Receive from PE 0, A[i] for i = (N/2 + 1), ..., N
Find Sum 1 = Sum of A[i] for i = (N/2 + 1), ..., N
Send Sum 1 to PE 0
```

Now, if we use the notation :

Send (1, $A[N/2 + 1]$, $N/2$) then it will be equivalent to :

"Send to PE 1, $N/2$ elements of the array, A starting from $(N/2 + 1)^{th}$ elements."

Similarly the notation—**receive** (0, $B[1]$, $N/2$) will be equivalent to saying :

"Receive from PE 0, $N/2$ values and store them starting at $B[1]$."

Using these notations, we can write programs for PE 0 and PE 1 (given above, in PASCAL as follows :

Program for PE 0 :

```
Array A[1 : N];
begin
  for i := 1 to N do
    Read A[i]
  end for;
  send (1, A[N/2 + 1], N/2);
  Sum 0 := 0
  for i := 1 to N/2 do
    Sum 0 := Sum 0 + A[i]
  end for;
  Receive (1, Sum 1, 1);
  Sum := Sum 0 + Sum 1;
  Write Sum
end
```

Program for PE 1 :

```
Array B[1 : N/2];
begin
  Receive (0, B[1], N/2);
  Sum 1 := 0;
  for i := 1 to N/2 do
    Sum 1 := Sum 1 + B[i]
  end for;
  Send (0, Sum 1, 1);
end
```

This is a complete program for two-processor addition of array elements.

Note : This example can be extended to develop a parallel program on the parallel architecture with multiple processors also. The array of integer elements is divided equally among multiple processors and each processor will add its own array set. So, multiple programs are running on multiple processors now. But, here, the programming mode can be modified by writing only one program instead of multiple programs. The same program is executed on multiple processors but each processor does computation only on its allotted data variable values. This type of programming approach is called as 'Single Program with Multiple Data' (SPMD) program.

Advantages of message passing programming model :

1. These programs are **portable**, from one machine to another.
2. These programs that are written using **libraries** such as MPI (Message Passing Interface) and PVM (Parallel Virtual Machine) that are platform independent. These programs can run on all parallel architectures such as multiprocessors, MPP (Massively Parallel Processors), SMP (Symmetric multiprocessors) etc.

10.4.2 Shared Memory Programming

This programming model is suitable for the shared memory parallel computer in which multiple processors share common global memory. The application programmer uses this model for writing parallel programs. He considers that the parallel program consists of processes which share variables in the global memory. In this model, there is no need to explicitly allot data for different processes because common data are stored in shared main memory. Thus, we can define the shared memory programming model as :

"Multiple cooperating processes which share variables i.e., processes have a single shared address space."

Or

"Processes execute asynchronously but need to be synchronized explicitly."

Or

"Interprocess communication is implicit and takes place through shared variables."

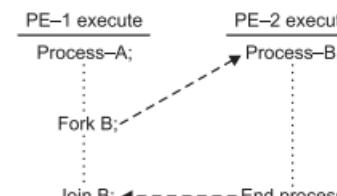
Please note that in this model, we need to create separate processes and assign task workload to each process. The main program uses 'fork' to create processes and then assigns processes to multiple processors.

Each PE independently carries out the work distributed to it. When all PEs execute their jobs then finally main program is executed which synchronizes all processes. Every process can combine to main program after finishing its work by the use of 'join' function.

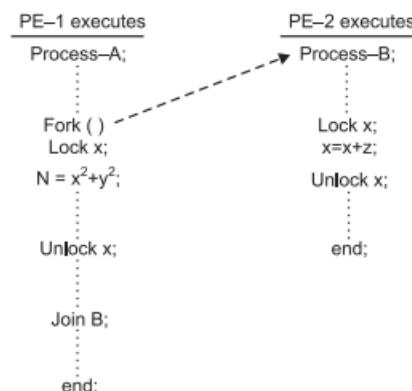
Let us take an example and understand it.

Herein, the process-A starts execution on processor (PE-1) 1. During its execution, when it comes to fork (), another child process, say, process B is created and it starts execution simultaneously on PE-2. Again when process-A encounters join B then it expects the result from process B. At this stage, if process B is yet not completed, then process-A has to wait otherwise it can continue.

But we have to take care when we implement interprocess synchronization while



accessing a shared memory variable. **Each process must not be able to access shared variable at the same time.** And this is achieved by using a 'lock mechanism'. Please note that when we do locking or unlocking of a shared variable, only one process is able to access and update the shared variable. So, during the time when the variable is locked by one process, other processes cannot access it. After the updation, the process must release the variable access by unlocking it. For example :



Here, the processor 1 (PE-1), starts the execution of process-A and when it encounters fork (), then process-B is created on PE-2. Now, concurrent execution of both the processes takes place. Both the processes use 'lock x' statement. Please note that the process which first executes this lock statement, gets the control of the shared variable x. It may use or update the shared variable. Only after unlocking it, by that particular process, other process may get access to it.

As explained earlier also, SPMD (Single Program Multiple Data) programming can be done using this parallel model also.

Let us now tabulate the points of differences between these two models.

Message Passing Programming model	Shared Memory Programming model
<ol style="list-style-type: none"> This model is suitable for parallel architecture with distributed memory. Multiple processes reside in separate address spaces. The parallel programs are written using standard libraries such as MPI and PVM that are portable. Users must explicitly allocate data to processes. These programs can run on any parallel architecture. Message passing programming standards are MPI and PVM. 	<ol style="list-style-type: none"> This model is suitable for parallel architecture with shared memory. Multiple processes share single address space. The parallel programs are written using platform specific languages and hence are not portable. There is no need of explicit data allocation. These programs run on specific parallel computers. Shared memory programming standards are open MP and Pthreads. The open MP standard includes many compiler directives and library routines. It provides shared-memory API for UNIX and WIN-NT.

10.4.3 Data Parallel Programming

There are many ways of defining data parallel model. Some of the definitions are :

"A model which exploits data parallelism. The elements of the data structure are assigned and distributed among multiple processors. A single statement of the program operates parallelly on different elements of the data structure."

Or

"A model where there is a single thread of control." (Programmer's View)

Or

"A model that does not demand for explicit synchronization. As each statement of the program operates on multiple and different data values parallelly, there is implicit synchronization after execution of every statement."

Or

"A model in which all data variables are placed in a single address space."

This model exploits data parallelism. It is suitable for SIMD (Single Instruction Multiple Data) or SPMD (Single Program Multiple Data) parallel machines. The important feature of this model is that the same operation is performed parallelly on different elements of the data structure such as arrays, matrices etc. The multiple processors of the parallel machine execute the same program on different data elements of data structure. **But this parallel program has a single thread of control. Please note that in this model there is data parallelism but no control parallelism/functional parallelism.** Control parallelism refers to the simultaneous execution of different instruction streams or threads.

Basically, this model is suitable for applications in which partitioning and distribution of data is possible. Writing such programs is easier here. **Some common examples of data parallel languages** are FORTRAN-90, High Performance Fortran (HPF) etc. These data parallel languages help the programmer to automatically perform repetitive operations on arrays, matrices like data structures. On the other hand, in serial languages, the programmer has to specify everything, right from the operation to perform, the partitioning of data structure, loops etc.

Therefore, the programmer using data parallel language can only concentrate on the logic required to solve a given problem.

10.5 PARALLEL ALGORITHMS FOR MULTIPROCESSORS

10.5.1 Classification of Parallel Algorithms

A parallel algorithm for the multiprocessor is a set of K concurrent processes which may operate simultaneously and co-operatively to solve given problem. When K = 1, it is called as a sequential algorithm.

In a task system, some points are there, where processes communicate with each other to ensure that a parallel algorithm works correctly and effectively. These points are called as **interaction points**. These points divide a process into stages. Thus, at the end of each stage a process may communicate with some other processes before the next stage of the computation is initiated.

Due to interactions between processes, some processes may be blocked at certain times.

The parallel algorithms in which some processes have to wait on other processes are called as **synchronized algorithms**.

Depending upon the input data and system interruption, the execution time of a process is different and as all the processes that have to synchronize at a given point has to wait for the slowest among them. This worst case computation speed may result in worse than expected speed up and processor utilization. This is the basic drawback of synchronized algorithms.

To overcome the problems encountered by synchronized parallel algorithms, **asynchronized parallel algorithms** exist for some set of problems in which processes are not generally required to wait for each other and communication is achieved by reading dynamically updated global variables stored in shared memory. Since concurrent memory access is performed so conflicts may occur which will introduce some small delay in processes accessing common variables.

An algorithm which requires execution on a multiprocessor system must be decomposed into a set of processes to exploit parallelism which can be done by **static** or **dynamic decomposition**.

In **static decomposition algorithm**, the set of processes and their precedence relations are known before execution. Static decomposition offers the possibility of very low process communication provided the number of processes that are small and their adaptability is limited.

In **dynamic decomposition algorithm**, a set of processes change during execution and can adapt effectively to variations in execution time of process graph but only at the expense of high process communication and other design overheads.

Note : Another approach of constructing parallel algorithms is macro pipelining which is applicable if computation can be divided into parts called as **stages**, so that the output of one or several parts is the input for another part which is shown in Fig. 10.1 below.

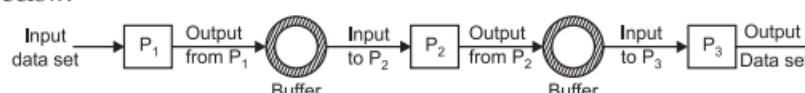


Fig. 10.1. Program flow in macropipelining.

Each computation part is realized as separate process, communication cost may be high unless communication is achieved by address transmission.

Please note that the processes that result from pipelining are heterogeneous while those resulting from partitioning are homogenous.

10.5.2 Synchronized Parallel Algorithms

A synchronized parallel algorithm is a parallel algorithm consisting of processes with the property that, there exist a process such that some stage of the process is not activated until another process has completed a certain stage of its program.

The synchronization can be performed using various synchronized primitives.

For example : To compute the matrix,

$$Z = A \cdot B + (C \times D) \cdot (I + G)$$

by maximum decompositon. New parallel algorithms may be constructed. The matrix is decomposed into three different processes as P_1 , P_2 and P_3 processes. It consists of different stages as shown below :

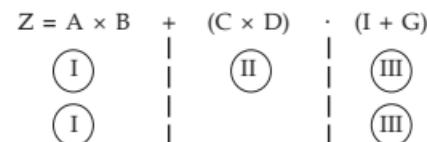


Fig. 10.2. Decomposition of matrix.

So, we observe that the processes-1 consists of two stages i.e., $(A \times B)$ and addition of $(A \times B)$ and $(C \times D)$ whereas process-2 and process-3 consists of one and two stage respectively, as $(C \times D)$ and $(I + G)$ and multiplication of $(C \times D)$ and $(I + G)$.

Algorithm :

```

Var w, y : Shared real ;
Var Sw, Sy : Semaphore ;
initial Sw = Sy = 0
Cobegin
    Process P1 : begin
        V ← A × B;           // stage 1 of P1
        P(Sy);
        Z ← V + Y;          // stage 2 of P2
    end
    Process P2 : begin
        W ← C × D;          // stage 1 of P2
        V(Sw);
    end
    Process P3 : begin
        X ← I + G;           // stage 1 of P3
        P(Sw);
        Y ← W + X;          // stage 2 of P3
        V(Sy);
    end
Coend

```

Please note here that the activation of second stage of process P_3 is subject to the condition that process P_2 is completed. Similarly, the second stage of P_1 cannot be initiated unless the second stage of P_3 is completed. Hence, the set of processes, P_1 , P_2 and P_3 is synchronized parallel algorithm.

10.5.3 Asynchronous Parallel Algorithms

In these algorithms, there is a set of global variables accessible to all processes. The communication between processes is achieved through global variables or shared data. When a stage of process is completed, the process reads some global variables. The process modifies a subset of the global variables and then activates the next stage or terminates itself based on the values of the variables read together with results obtained from the last stage. In asynchronous algorithms, **there is no explicit dependency** between the processes as in synchronous parallel algorithms which can be considered as one of the advantages of this algorithms. In this algorithm, processes never wait for inputs at

any time but continue execution or terminate according to the information contained in global variables. Herein, the processes may be blocked from entering critical sections, which can be considered as one of its disadvantages.

For example :

$$x_{i+1} = x_i - f'(x_i) f(x_i)$$

Here, an asynchronous iterative algorithm consisting of two processes, P_1 and P_2 and three global variables are defined. Note that, process P_1 updates variables V_1 and V_3 while process, P_2 updates V_2 .

Algorithm :

```

function f, f';
Var V1, V2, V3: shared real; // Global variables
Cobegin
Process P1 : begin
    While < termination criteria is not satisfied > do
        begin
            V1 <- (f(V3));
            V3 <- V3 - V2·V2;
        end
    end P1
Process P2 : begin
    While < termination criteria is not satisfied > do
        V2 <- f'(V3);
    end P2
Coend

```

In this program, as soon as the process completes updating a global variable, it proceeds to the next updating by using the current values of relevant variables without any delay. Here, process P_2 executes $f'(V_3)$ which is required by process P_1 . That is, if the initial values of variables are :

$$V_1 = f(x_0)$$

$$V_2 = f'(x_0)$$

$$V_3 = x_1$$

then the sequence and time period of step completion for each iteration within each process may be illustrated as :

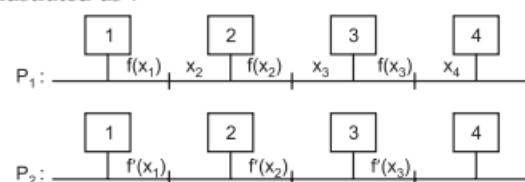


Fig. 10.3. Timing diagram for Asynchronous Parallel Algorithm.

The **main advantages** of this algorithm is its general applicability. These algorithms are mainly used when decomposition of the processes is difficult.

It's disadvantages are also there. First, note that critical sections are needed in the algorithms. Secondly, the speed-up of the algorithm is quite limited.

10.6 PERFORMANCE OF PARALLEL ALGORITHMS

The effectiveness of a multiprocessor for a synchronized iterative algorithm depends on the performance feature of the algorithm. Memory interference affects the performance in tightly coupled systems. In case of loosely coupled systems, the cost of the interprocess communication is the dominant factor. Synchronization and interprocess communication effects the performance of a synchronized iterative algorithm.

10.7 PARALLEL PROGRAMMING LANGUAGES

Parallel programming is the design, implementation and tuning of parallel computer programs which take the advantage of parallel computing system. It also refers to the application of the parallel programming models to existing serial programs.

For example : FORTRAN-90, OCCAM, C-Linda and CCC.

10.7.1 Fortran-90

Fortran 90 is a new programming language standard that permits efficient execution of code on conventional as well as on vector processors. But, Fortran-90 is less efficient on parallel computers.

Fortran 90 New features

1. **Free format on source code :** In Fortran 90, you can use either the Fortran 77 input format or free format.
2. **Dynamic allocation and pointers :** Dynamic memory allocation by means of the ALLOCATABLE attribute and the ALLOCATE and DEALLOCATE statements. POINTER attribute, pointer assignment, and NULLIFY statements to facilitate the creation and manipulation of dynamic data structures.
3. **User defined data types :** You can now define your own composite data types, similar to struct in C or record in Pascal.
4. **Built-in array operations :** Statements like $A = 0$ and $C = A + B$ are now valid when A and B are arrays.
5. **Operator overloading :** You can define your own meaning to operators like + and = for your own data types as you do in C++.
6. **Recursive procedures :** A vastly improved argument-passing mechanism, allowing interfaces to be checked at compile time.
7. CASE construct for multiway selection.

Once a Fortran 90 program is written it can be compiled using these steps :

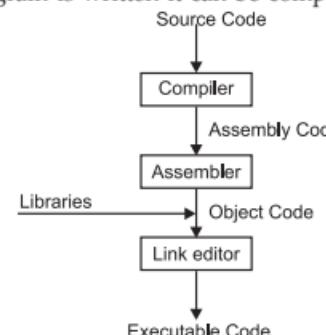


Fig. 10.4. Fortran-90 program compilation steps.

Note : A Fortran-90 source code file is stored with *f90* as a file extension.

FORTRAN-77 was quite non-portable, with no user-defined data types, with no dynamic storage, lack of explicit recursion and so on.

There are two standards for parallel computer :

- (a) High Performance Fortran (HPF).
- (b) Message Passing Interface (MPI)

HPF has many features that are not in FORTRAN-90 also. They are as follows :

- (a) Data Parallel programming.
- (b) Top performance on SIMD and MIMD computers with non-uniform memory access.
- (c) Code tuning for various architectures.
- (d) Minimal deviation from other standards.
- (e) Define open interfaces to other languages.
- (f) Encourage input from the high performance computing community.

FORTRAN 2000 is currently being worked upon. It will have a full support for OOP.

10.7.2 Occam

Introduction : Occam is a parallel programming language developed in Great Britain. Occam is a good language for exploring the ideas of the message passing style of parallel programming. Occam has been developed by **David May**.

Occam and the Transputer : From the Occam model, Inmos developed a hardware chip to support their concurrency model. This hardware is in the form of a very large scale integration (VLSI) integrated chip (IC) called the **Transputer**. The Transputer is a 32-bit microprocessor (20 MHz clock) that provides 10 MIPS (Million Instruction Per Second) and 2.0 MFLOPS (Million Floating Point Operations Per Second) processing power with 4 KB of fast static RAM and concurrent communication capability all on a single chip. Though Occam is a high-level language, it can be viewed as the assembly language for the Transputer.

The language and the hardware are so designed that an Occam program consisting of a collection of concurrent processes may execute on one Transputer or be spread over many Transputers with little or no change in the Occam code. Therefore, the designer can develop his or her Occam program on **one Transputer**. If higher performance is required then the designer can spread the Occam processes over a network of interconnected Transputers.

Different dialects (versions) of Occam : The first version of Occam was distributed in 1983, known as Occam 1. Then, Occam 2, an enhanced version of Occam 1 was released in 1986. Many new features were added to this language like floating-point arithmetic, multidimensional arrays, etc. In 1988, new protocols were added on the Occam channels. Presently, the work is going on Occam 3.

In Occam, communication between concurrent processes is achieved by passing messages along **point-to-point channels**. Please note that the point-to-point means that the channels source and destination must be at one point or reside in one concurrent process.

Features of Occam

1. All reserved words must be in capital letters.
2. Spaces are delimiters.
3. Occam is **line-oriented** which means that each statement starts on a new line.

4. The continuation to the next line is possible by breaking an expression at an operator, semicolon or comma. Comments are designated by ---- to the end of the line.
5. Each construct must be indented two spaces to show structure. This indenting has the potential to cause grief for programmers.

Program Structure :

The structure of a program is a process with declarations preceding it.

<declares>

<process>

For example :

INT j:

SEQ

j := 1

j := j + 1

Please note that the Occam's "process" can be considered as a generalization of "statement" in other languages e.g., PASCAL. An Occam process is different from the process that we study in O.S. (Operating Systems).

For example, Two Occam processes need not be "concurrent processes". Think of process as an "action", i.e., something that is done.

Primitive actions/processes

In Occam, there are five primitive actions/processes :

- (a) **assignment** : assigns value from an expression to a variable.
- (b) **receive** : receives value from a channel. User "?" to signify a query.
- (c) **send** : sends expression value on a channel. Uses "!" to signify an exclamation.
- (d) **SKIP** : do nothing and terminate the process i.e., no operation.
- (e) **STOP** : do nothing and never terminate the process i.e., never get to the next process.

The syntax and example for each primitive given below :

Primitive	Syntax	Example
Assignment	<variable> := <expression>	x := m + 10
Receive	<channel> ? <variable>	ch ? y
Send	<channel> ! <expression>	ch! a + 10
SKIP	SKIP	SKIP
STOP	STOP	STOP

Also note that in Occam expressions, parentheses are used to specify the order of operations as there is no operator precedence.

For example :

$x := 2 \times y + 1$ is illegal.

You must use parentheses as in the following expression :

$x := (2 \times y) + 1$ is valid.

Please note that since a major goal of occam was secure concurrent programs, so the language does not allow pointers. Also note that favourite language features like

dynamic memory allocation, recursions, dynamic process allocation are also missing from Occam. The security that occam brings to the programming concurrent program is especially important in a student environment where they are learning the concepts of concurrency while mastering the language.

10.7.3 C-Linda

Linda, developed by Professor David Gelernter (Yale University) is a coordination language *i.e.*, it combines both C-Linda and C. To write parallel programs, the programmers must be able to create and coordinate multiple execution threads. Linda is a model of process creation and coordination to the base language in which it is embedded.

The Linda model does not care how the multiple execution threads in a program compute. It deals with how these execution threads are created and how they are organized into a coherent program. The Linda model is a **memory model**. Linda memory (called as tuple space) consists of a collection of logical tuples. There are two kinds of tuples :

- (a) Process tuples that are under active evaluation.
- (b) Data tuples are passive.

The process tuples, which are all executing simultaneously, exchange data by generating, reading and consuming data tuples. A process tuple that is finished executing, turns into a data tuple, indistinguishable from other data tuples. Please note that Linda is a model, not a tool. A model (or paradigm) represents a particular way of thinking about a problem. It can be realized or implemented in many different ways. On the other hand, a software tool is a working system that can be used to solve problems. C-Linda is a tool—a piece of software that supports parallel programming. C-Linda is one realization of the Linda model.

The C-Linda Language : The C-Linda programming language is the combination of the C and the Linda languages.

The Linda shared memory space is called as a Tuplespace. A tuplespace is an implementation of the associative memory paradigm for parallel/distributed computing. It provides a repository of tuples that can be accessed concurrently. Tuplespace holds one type of data called tuples. Tuples are ordered typed lists of items.

For example : ("hello", 3, 2.8) is a tuple which contains three items, the first being a character string, the second an integer and the third a floating point number. Tuples can contain both actual and formal items. An actual item contains a specific value like 3 or 2.8.

Tuples are of two types. They can be :

- (a) Active tuples.
- (b) Passive tuples.

Active tuples contain atleast one value that has not yet been evaluated. Processes are created in Linda by sending active tuples into Tuplespace.

For example : ("factorial", 7, fact (7)) is an active tuple. When this tuple arrives in Tuplespace, Linda spawns a concurrent process to evaluate fact (7).

Passive tuple : When that process returns a value, the value returned replaces the function call, changing the value of the tuple to ("factorial", 7, 5040) and making the tuple, a passive tuple. Thus, a passive tuple is a tuple for which all values have been evaluated.

The model of parallel computation is shown in Fig. 10.5.

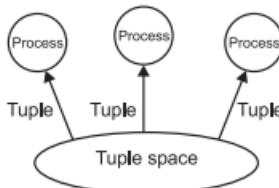


Fig. 10.5. Linda programmer's model.

Next we will discuss some of the functions used in C-Linda.

1. **Out ()** : It sends a passive tuple into Tuplespace.

For example : Out ("hello", 3, 2.8) will result in the tuple ("hello", 3, 2.8) appearing in the Tuplespace.

2. **eval ()** : It sends an active tuple into Tuplespace.

For example : eval ("factorial", 7, fact (7)) will result in the active tuple ("factorial", 7, fact (7)) appearing in the Tuplespace and a process being spawned to evaluate the function call fact (7) concurrently. When the function returns, the value will replace the function call in the tuple and the tuple will become passive.

Some other Linda functions are in (), inp (), rd () and rdp (). These functions are used to retrieve tuples from Tuplespace. The arguments to these functions are items in a tuple template or **anti-tuple**. A **tuple template** is a query that matches tuples in Tuplespace. Tuple templates contain both formal and actual values, just like regular tuples.

Matching follows some rules :

1. A tuple template will only match tuples with the same number of items. Matching is then tested on an item per item basis.
2. A formal template item will not match a formal tuple item.
3. A formal template item will match an actual tuple item if the types are the same.
4. An actual template item will match a formal tuple item if the types are the same.
5. An actual template item will match an actual tuple item if the types are the same and the actual values are equal.

For example : The tuple template ("data", ? x), where x is declared as an integer, will match the tuple ("data", 5). When a match occurs, data is returned to a formal template item when it matches an actual tuple item. So, here, x will get the value of 5 when the match occurs. Thus, the value is retrieved from Tuplespace.

The in () call will remove the matched tuple from Tuplespace, the rd() call will not.

The inp () and rdp () calls are predicate versions of in () and rd (). It means that it will return 1 if a match is found and 0 otherwise.

The non-predicate versions will block until a match is found.

C-Linda programs must call their main procedure—real-main () and must include

linda.h in each code module.

A sample C-Linda program is given below :

```
/* hello.clc : a sample C-Linda program */
#include "Linda.h"
int student (int num)
{
    int i;
    for (i = 0 ; i < num ; i++)
        out ("hello world");
    return 0;
}
int real_main (int argc, char * argv [])
{
    int result;
    eval ("student", student (5));
    in ("hello world");
    in ("hello world");
    in ("student", ? result);
    return 0;
}
```

10.7.4 CCC

CCC is a high level parallel programming language which provides a coherent integration of various parallel programming paradigms. CCC supports **both data and task parallelism**.

Data parallelism is specified in SIMD model and **task parallelism** is specified in MIMD model. The implementation is mainly based on a virtual shared memory machine interface that supports both shared and dynamic task creation. As shown in Fig. 10.6, CCC supports both task and data parallelism.

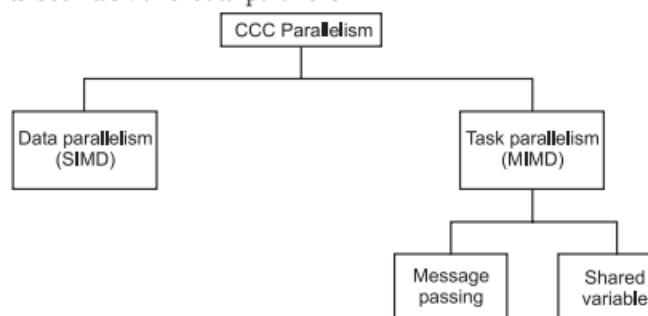


Fig. 10.6. Features of CCC.

A CCC program consist of a collection of coordinated concurrent data parallel or task parallel task. Please note that data parallelism (in SIMD) means that the data parallel tasks are executed synchronously and perform same operations on different data. For this shared memory abstraction is provided.

But **task parallelism** (in MIMD) means that the task parallel tasks are executed **asynchronously** and usually perform different operations on different data. For this message passing communication abstraction and shared variables synchronization abstraction are provided.

Now, we elaborate data and task parallelism in detail.

Data Parallelism : Data parallelism is specified by domain construct. Data parallel function is defined in domain construct. Each data-parallel function represents a collection of number of parallel tasks. Consider a domain construct as :

```
domain name [n]
{
    data-declarations ;
    data-parallel-functions ;
}
```

In this example, a number of parallel tasks is defined by '*n*'. A call of a data-parallel function will concurrently create '*n*' tasks. The **synchronization abstraction and communication abstractions** are implicitly specified. The **synchronization abstraction** is implicitly specified by the synchronous semantics of the SIMD model. On the other hand, **communication abstraction** provides remote read, remote write and reduction operations on the variables defined within the domain construction. The data distribution specifications can be given in parameter specifications in the data parallel function definitions.

Task parallelism : The concurrency concept for task parallelism is specified via the definition of task parallel functions and parallel section construct. The **Par** construct is used to concurrently invoke a group of task parallel functions :

```
par
{
    func-1 ;
    func-2 ;
    .....
    func-n ;
}
```

This will execute the *n* task parallel function fun-1, fun-2, fun-*n* as *n* task in parallel. Here parallel sections exists only when all created tasks exist.

The **Parfor** construct is used to concurrently invoke multiple instances of a group of task parallel functions.

```
Parfor (init-expr, exit-expr, step-expr)
{
    func-1 ;
    func-2 ;
    .....
    func-n ;
}
```

This will execute task parallel functions func-1, func-2,, func-*n* as task in parallel. The init-expr, exit-expr and step-expr in the parfor construct is the same as that in the FOR-loop of C.

Communication abstraction for task parallelism is specified via channels or asynchronous message queues. There are four types of channels : pipes, splitters, mergers and multiplexer's pipes for one-to-one communication. Other three types of channels can be implemented using pipes.

- Splitters are for one-to-many communications.
- Mergers are for many-to-one communications.
- Multiplexers are for many-to-many communication.
- Channels provides a simple abstraction for implementing communication in message passing programming model.

The synchronization abstraction for the task parallelism is specified via **active monitors**. Monitors are structured and efficient construct for implementing both the mutual exclusion and condition synchronization is shared variable programming model. The functions defined are mutually exclusive by default. A read keyword can be put before the definition function, if function only reads the variables in the monitors.

The runtime library : CCC routine library contains a collection of functions that implements the most important abstraction of CCC on top of the virtual shared memory machine interface.

For efficiency, the SIMD model of domain construct will transform and be executed in the SPMD model.

The CCC Compiler : CCC compiler translates the CCC programs into C program that call functions provided by the virtual shared memory machine interface and the runtime library.

In data parallelism, the CCC compiler first needs to map the virtual tasks into the physical task based on the number of processors available. Usually, multiple virtual tasks will be mapped into one physical task. Thus, the mapping would be done by merging multiple data mapped to the same physical task.

In **task parallelism**, the static analysis of communication structures checks whether the channels are used as they are declared. This analysis is based on a **compact control flow graph** for each function. There are five different types of nodes in this compact control flow graph. They are as follows :

1. **Call node** : Call to sequential function.
2. **Spawn node** : Call to parallel function.
3. **Compound node** : Represent sequence of nodes.
4. **Alternative** : Represent list of nodes that are mutually exclusive.
5. **Repetition node** : Represent a node executed multiple times.

10.8 PROBLEMS BASED ON PARALLEL ALGORITHMS

10.8.1 Problem of Prime numbers

Sequential Code : A sequential program for this problem usually employs an array with elements initialized to 1 (True) and set to 0 (False) when the index of the element is not a prime number. Let the last number be n and the square root of n be sqrt__n , we might have a sequential code like this :

```
for (i = 2 ; i < n ; i++)  
    prime [i] = 1;  
for (i = 2 ; i <= sqrt_n ; i++)  
    if (prime [i] == 1)
```

```
for (j = i + i; j < n; j = j + i)
    prime[j] = 0;
```

Herein, there are $\left\lfloor \frac{n}{2} - 1 \right\rfloor$ multiples of 2, $\left\lfloor \frac{n}{3} - 1 \right\rfloor$ multiples of 3 and so on. Hence,

the total sequential time is given by :

$$t_s = \left\lfloor \frac{n}{2} - 1 \right\rfloor + \left\lfloor \frac{n}{3} - 1 \right\rfloor + \left\lfloor \frac{n}{5} - 1 \right\rfloor + \dots + \left\lfloor \frac{n}{\sqrt{n}} - 1 \right\rfloor$$

assuming the computation in each iteration equates to one computational step. The sequential time complexity is $O(n^2)$.

Parallel Code : A pipelined implementation is quite useful here. First, a series of consecutive numbers is generated that feeds into the first pipeline stage. This stage extracts all multiples of 2 and passes the remaining numbers onto the second stage. The second stage extracts all multiples of 3 and passes the remaining numbers onto the next stage and so on.

The code for a (process P_i) could be based upon :

```
recv (&x, P_{i-1});
// repeat following for each number
recv (&number, P_{i-1});
if (number % x != 0)
    send (&number P_{i+1})
```

Note that a simple for-loop is insufficient for repeating the actions because each process will not receive the same amount of numbers and the amount is not known before-hand. A general technique for dealing with this situation in pipelines is to use a "terminator" message, which is sent at the end of the sequence. Then each process could be :

```
recv (&x, P_{i-1});
for (i = 0; i < n; i++) {
    recv (&number, P_{i-1});
    if (number == terminator) break;
    if (number % x != 0)
        send (&number P_{i+1});
}
```

Analysis : Analysis of the algorithm is similar to the sorting algorithm except that each process in the pipeline will complete fewer steps than the previous process because it will not receive all the numbers that the previous process receives. The parallel implementation has $n + n - 1 = (2n - 1)$ pipeline cycles where there are n pipeline processes and n numbers to sort.

Each pipeline cycle requires atleast :

$$\begin{aligned} t_{\text{comp}} &= 1 \\ t_{\text{comm}} &= 2(t_{\text{startup}} + t_{\text{data}}) \end{aligned}$$

\therefore Total execution time, t_{total} is given by :

$$\begin{aligned} t_{\text{total}} &= (t_{\text{comp}} + t_{\text{comm}})(2n - 1) \\
&= (1 + 2(t_{\text{startup}} + t_{\text{data}}))(2n - 1) \end{aligned}$$

10.8.2 Searching

Sequential Search : In sequential computing, the problem is solved by scanning the sequence S and comparing x with every element in S , until either an integer equal to x is found or the sequence is exhausted without success. This algorithm takes $O(n)$ time which is optimal since every element of S must be examined.

Parallel Search : Searching on a tree—We use a binary tree network with n leaf nodes (processors). The total number of nodes and hence, the number of processors is thus $2n - 1$. Let L_1, L_2, \dots, L_n denote the processors at the leaf nodes. The sequence $S = \{S_1, S_2, \dots, S_n\}$ which has to be searched is stored in the leaf nodes.

The processor L_i stores the element S_i . The algorithm has 3-stages :

1. The root-node reads x and passes it to its two child nodes. In turn, these send x to their child nodes. The process continues until a copy of x reaches each leaf-node.
2. Simultaneously, every leaf node L_i compares the element x with S_i . If they are equal, the node L_i sends the value 1 to its parent, else it sends a 0.
3. The results (1 or 0) of the search operation at the leaf nodes are combined by going upward in the tree. Each intermediate node performs a logical OR of its two inputs and passes the result to its parents. This process continues until the root node receives its two inputs, performs their logical OR and produces either a 1 (for yes) or 0 (for no). e.g.,

Say

$S = \{26, 13, 36, 28, 15, 17, 29, 17\}$ and

$x = 17$

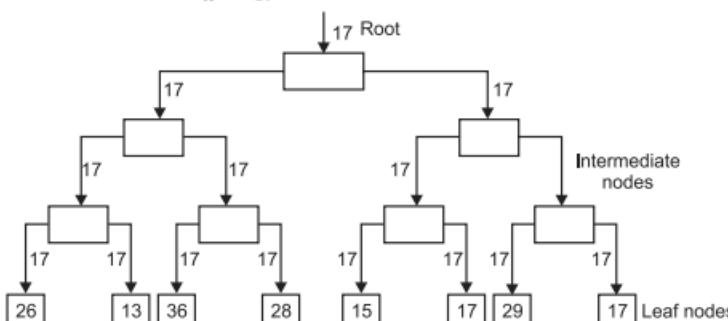


Fig. 10.7. (a) Search Tree.

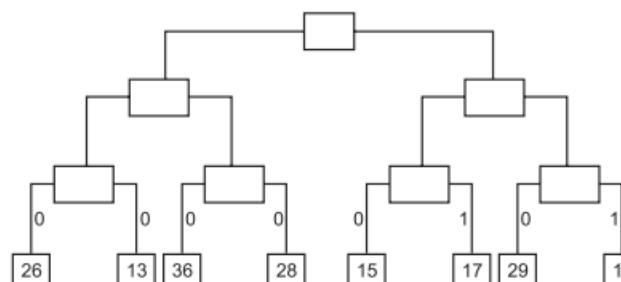


Fig. 10.7. (b) Search Tree.

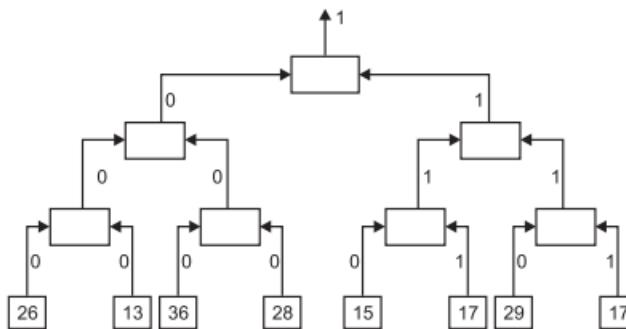


Fig. 10.7. (c) Searching on a tree.

It takes $O(\log n)$ time to go down the tree, constant time to perform the comparison and again $(\log n)$ time to go back up the tree. Thus, the time taken by algorithm to search for an element is $O(\log n)$.

10.8.3 Graph Algorithms

Moore's shortest path algorithm

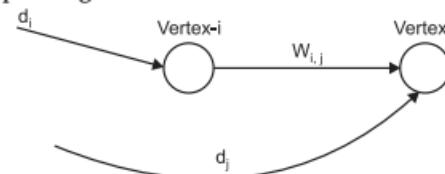


Fig. 10.8. Moore's shortest path algorithm.

Find the distance to vertex- j through vertex- i and compare with the current minimum distance to vertex- j . Change the minimum distance if distance through vertex- i is shorter. Mathematically, if d_i is the current minimum distance from the source vertex to vertex- i & w_{ij} is the weight of the edge from vertex- i to vertex- j , we have

$$d_j = \min(d_j, d_i + w_{ij})$$

The problem could be solved by applying this formula repeatedly.

The formula here is implemented using a directed search. A FIFO vertex-queue is created which holds a list of vertices to examine. Vertices are considered only when they are in the vertex queue. Initially, only the source vertex is in the queue.

Suppose there are n vertices and vertex 0 is the source vertex. The current shortest distance from the source vertex to vertex- i will be stored in the array dist [i], ($i \leq 1 < n$).

At first, none of these distances will be known and the array elements are initialized to infinity. Suppose $w[i][j]$ holds the weight of the edge from vertex- i and vertex- j & infinity if no edge. The code could be of the form.

```

new dist[j] = dist[i] + w[i][j];
if (new dist[j] < dist[j])
    dist[j] = new dist[j];

```

When a shorter distance is found to vertex- j , vertex- j is added to the queue, which will cause vertex- j to be examined again.

Sequential Code : Let next-vertex () return the next vertex from the vertex-queue or no vertex if none we will assume that an adjacency matrix is used, named $w [] []$, which is accessed sequentially to find the next edge. The sequential. Code could be then of the form :

```
While ((i = next - vertex ()) != no - vertex)      /* while a vertex */
    for (j = 0 ; j < n ; j++)                      /* get next edge */
        if (w [i] [j] != infinity)                  /* if an edge */
        {
            new dist[j] = dist [i] + w [i] [j];
            if (new dist[j] < dist [j])
            {
                dist [j] = new dist[j];
                append-queue (j);                     /* vertex to queue if not there */
            }
        }
    }
```

Parallel implementation : Centralized work pool. This parallel implementation uses a centralized work pool holding the vertex queue, vertex-queue [] as tasks. Each slave takes vertices from the vertex-queue of this central pool. For the slaves to identify edges and compute distances, they need access to both the structure holding the graph weights (adjacency matrix) and the array holding the current minimum distance, dist []. If this information is held by the master process, message will need to be sent to the master to access the information. This could lead to a very significant communication overhead. Since the structure holding the graph weights is fixed, this structure could be copied into each slave. Let us assume that the distance array, dist [] is held centrally and simply copied with the vertex in the entirety. The code could be of the form :

Master :

```
While (vertex-queue () != empty) {
    recv (PANY, Source = Pi);           /* Request task from slave */
    V = get-vertex-queue ();
    send (&V, Pi);                      /* send next vertex */
    send (&dist, &x, Pi);              /* current dist array */
    recv (&j, &dist [j], PANY, source = Pi); /* new dist */
    append-queue (j, dist [j]);          /* append vertex to queue */
};

recv (PANY, source = Pi);           /* request task from slave */
send (Pi, termination-tag);        /* termination message */
```

Slave (process i) :

```
send (Pmaster);                      /* send request for task */
recv (&V, Pmaster, tag);             /
/* get vertex no. */

if (tag != termination-tag) {
    recv (&dist, &n, Pmaster);       /* dist array */
    for (j = 0 ; j < n ; j++)
        if (w [V] [j] != infinity) { /* if an edge */

```

```
    new dist[j] = dist[v] + w[v][j];
    if (new dist[j] < dist[j]) {
        dist[j] = new dist[j];           /* add vertex to queue */
        send (&j, &dist[j], Pmaster); /* send updated distance */
    }
}
```

The master waits for requests from any slave but must respond to the specific slave that makes a request. The notation source = P_i is used to indicate the source of the message.

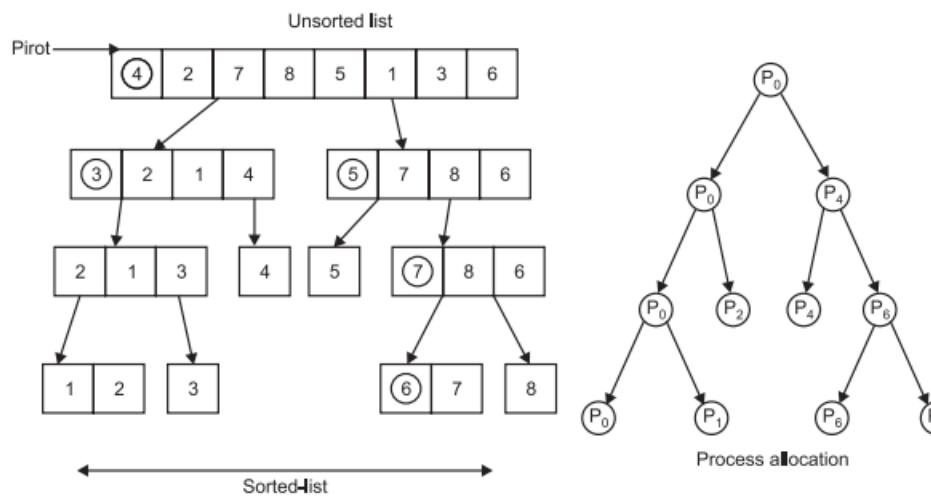
10.8.4 Quick Sort

Sequential code : Quick Sort is usually described by a recursive algorithm. Suppose an array list () holds the list of number & **pivot** is the index in the array of final position of the pivot. We could have the code of the form :

```
quick sort (list, start, end)
{
    if (start < end) {
        partition (list, start, end, pivot);
        quick sort (list, start, pivot-1);      /* recursively call on sublists */
        quick sort (list, pivot + 1, end);
    }
}
```

Partition () moves numbers in the list between start to end so that those less than the pivot are before the pivot and those equal or greater than the pivot are after the pivot. The pivot is in its final position of the sorted list.

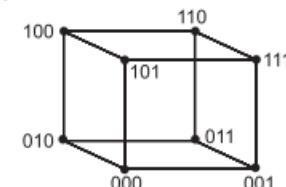
Parallelizing Quick sort : One obvious way to parallelize quick sort is to start with one processor and pass on one of the recursive calls to another processor while the keeping other recursive call to perform. This will create now a familiar tree structure as shown below :



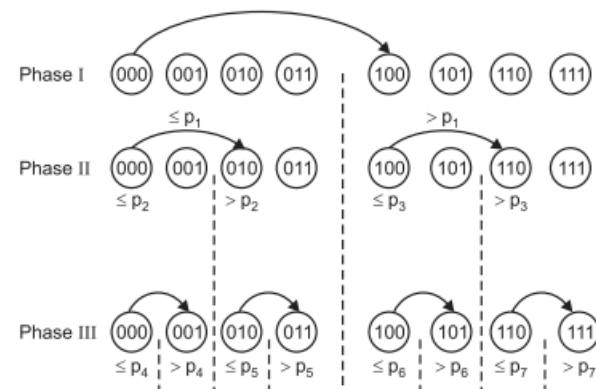
Here, the pivot is carried with the left list until final sorting action. **Computation.** First one processor operates upon n numbers. Then two processors each operate upon $\frac{n}{2}$ numbers. Then four processors each operate upon $\frac{n}{4}$ numbers & so on. So,

$$t_{\text{comp}} = n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots \approx 2n$$

Quick sort on Hypercube



Suppose a list on n numbers is initially placed on one node of a d -dimensional hypercube. The list can be divided into two parts according to the quick sort algorithm by using a pivot determined by processor, with one part sent to adjacent node in the highest dimension. Then the two nodes can repeat the process, dividing their lists into two parts using locally selected pivots. One part is sent to a node in next highest dimension. This process is continued for d -steps in total so that every node has a part of the list. For a 3d hypercube with numbers original in node 000, we have the splits as follows :



where $p_1, p_2, p_3, p_4, p_5, p_6$ and p_7 are pivot elements.

Finally, the parts can be sorted using a sequential algorithm, all in parallel. If required, the sorted parts can be returned to one processor in a sequence that allows the processor to concatenate the sorted lists to create the final sorted list.

Computation : With a sorted list, pivot selection can be done in one step, $O(1)$, if there always were $\frac{n}{p}$ numbers & we simply choose the $\frac{n}{2p}^{th}$ number in this list (i.e., the median) In more general case, the time complexity will be higher.

Storage requirements : Since we have considered the case in which all numbers fall into one processor, which means that sufficient storage must be present to hold the whole list of numbers at each processor.

10.8.5 Dictionary Operations (Searches)

Search algorithms operate on elements, called **keys**, stored in a table of finite size. The goal is to organize the table and implement the algorithms so that functions such as inserting keys and their associated data into the table, deleting keys and data from the table and searching for keys in the table, execute as quickly as possible.

Different Search Algorithms are discussed below :

1. **Weak Search :** Returns a result that is not guaranteed to be up-to-date. Weak search should be used whenever possible because it does not require locking any nodes. Thus, the parallel algorithm has no overhead and a process performing a weak search interferes with no other processes.
2. **Strong Search :** Looks for a given key and returns the node, v , associated with that key. To make sure that v will remain the node associated with the key as long as it is needed, the process performs a weak search, then locks v . When strong search returns a node, that node is guaranteed to be the one associated with the key and is locked. The node is unlocked only after the operation update, insert or delete is completed.

10.8.6 PRAM Model (s) Parallel Random Access Machine

The PRAM model allows parallel algorithm designers to treat processing power as an unlimited resource, much as programmers of computers with virtual memory are allowed

to treat memory as an unlimited resource. The PRAM model is simple and it ignores the complexity of interprocessor communication. Because communication complexity is not an issue, the designer of PRAM algorithm can focus on parallelism inherent in a particular computation.

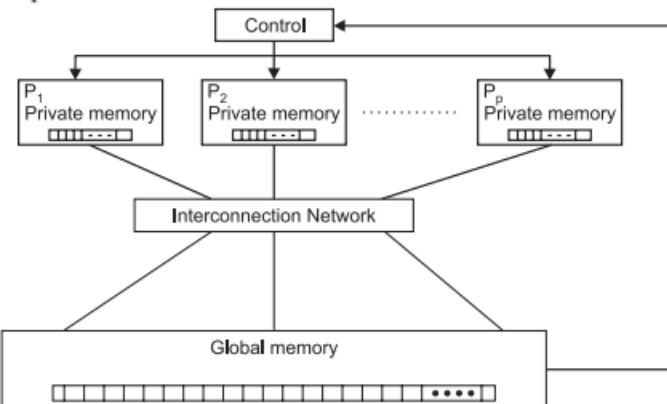


Fig. 10.9. PRAM Model.

PRAM consists of a control unit, global memory and a set of processors, with its own private memory. A PRAM computaton begins with the input stored in global memory and a single active PE. During each step of computation an active, enabled processor may read a value from a single private or global memory location, perform a single RAM operation and write into one local or global memory location alternatively, during a computation step, a processor may activate another processor. All active, enabled processor must execute the same instruction, on diffeent memory locations. The computation terminates when the last processor halts. There are 4 subclasses of the PRAM model, provided to define how simultaneously reading and writing should be handled :

I. Exclusive read, exclusive write (EREW PRAM)

In EREW model, a memory location can only be accessed by one processor at a time, whether for reading the contents or writing new values. This is most restrictive model.

II. Concurrent read, exclusive write (CREW PRAM)

In CREW model, a memory location can be accessed by more than one processor simultaneously but only for reading the contents of the location. All accesses to write new values can only be done one at a time.

III. Exclusive read, concurrent write (ERCW PRAM)

In ERCW model, a memory location can be accessed simultaneously for writing but not for simultaneous reading. This model is usually not considered and is subsumed in next model.

VI. Concurrent read, concurrent write (CRCW PRAM)

In CRCW model, a memory location can be accessed simultaneously by different processors, for either reading or writing. This is the most powerful model.

In those cases where simultaneous writing is permitted (*i.e.*, ERCW and CRCW), additional specification is necessary to explain how conflicts are resolved and consequently what will be the final result stored in the location. There are 4 quoted possibilities :

- (a) **Common** : Simultaneous writing is only allowed if all the new values to be written are same. Then the result is defined as this value.
- (b) **Arbitrary** : One processor is selected to perform the write operation and all other processors are inhibited.
- (c) **Priority** : It is similar to arbitrary (above) but here a predefined priority order is established to select the processor that will succeed, the rest of processors failing. The predefined order is by process identification number (*id*).
- (d) **Sum** : Here, the final result is the arithmetic sum of values being written by the individual processors.

PRAM Algorithms

Because a PRAM algorithm begins with only a single processor active, PRAM algorithms have 2 phases :

In first phase, a sufficient number of processors are activated.
& in 2nd phase, these activated processors perform computation in parallel.

Given a single active processor, it is easy to see that $\lceil \log p \rceil$ activation steps are both necessary & sufficient for p processors to become active, since the number of active processor can double by executing a single instruction (see fig. 10.10) All logarithms are to base 2. In our presentation of PRAM algorithms we use meta-instruction :

Spawn (<processor names>)

to denote this logarithmic time generation of processors from a single active processor.

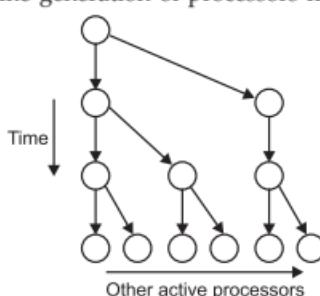


Fig. 10.10. Shows how 1 active processor activates p other processors.

To make, 2nd phase of PRAM algorithms computations easier to read, we allow references to global registers to be array references. We assume there is a mapping from these array references to appropriate global registers. The construct

```
for all <processor list>
do
<statement list>
end for
```

denotes a code segment to be executed in parallel by all specified processors. The symbol \leftarrow denotes assignment.

e.g., A PRAM algorithm to sum n elements using $n/2$ processors, is :

SUM (EREW PRAM)

Initial condition : List of $n \geq 1$ elements stored in $A[0] \dots n-1$.

Final condition : Sum of elements stored in $A[0]$

Global variables : $n, A[0], \dots (n-1), j$

begin

spawn ($P_0, P_1, P_2, \dots, P_{n/2-1}$)

for all P_i where $0 \leq i \leq (n/2-1)$ do.

 for $j = 0$ to $(\log n-1)$ do

 if i modulo $2^j = 0$ and $2i + 2^j < n$ then

$A[2i] \leftarrow A[2i] + A[2i + 2^j]$

 end if

 end for

end for

10.8.7 Message passing libraries for Parallel Programming

Parallel processing can be achieved using three public domain passing systems :

- (a) MPI – Message Passing Interface.
- (b) PVM – Parallel Virtual Machine.
- (c) pthreads.

Although not formally a standard, PVM is very stable and also widely used.

Parallel programming models are divided into implicit and explicit programming mode. **Message passing is an explicit parallel programming model.** Explicit parallelism means that parallelism is explicitly specified in the source code by the programmer using special language constructs, compiler directives or library function calls.

It consists of processes with multiple address space and control flow. Users must explicitly allocate data and workload to processes in message-passing programs. These programs are multithreaded and asynchronous requiring explicit synchronization to maintain correct execution order. These processes have their own separate address space.

MPI is a standard specification for a library of message passing functions, which is based on consortium of parallel computer vendors, library writers and specialists. MPI activates portability by providing a public-domain, platform independent standard of message passing library.

Parallelism issues in MPI

If static processes are assumed, all processes are created when a parallel program is loaded and they stay alive until the entire program terminates. All these processes are grouped together and identified by MPI-COMM-WORLD.

Let us discuss another standard as PVM.

PVM or Parallel Virtual Machine is a self contained, public domain software system that was originally designated to enable a network of heterogeneous unix computers to be used as a large-scale, message passing parallel computer. In PVM, the programmer decomposes the problem into separate programs. Each program is written in C (or Fortran) and compiled to run on specific types of computers in the network. **With PVM, user can construct a virtual machine.** The set of computers used on a problem must be defined first, prior to running the programs. **This forms a virtual parallel machine.** The user can then dynamically create and manage a number of processes to run on this virtual machine. PVM provides library routines to support interprocess communication and other functions.

Please note that although we discuss the computing platform for both PVM and MPI as a workstation cluster, the software tools can be used on a wide range of multicomputers and multiprocessor platforms.

SUMMARY

In this chapter, we have seen the characteristics of parallel algorithms, their need and various programming models. We have also discussed about different types of parallel algorithms for multiprocessors—synchronous and asynchronous algorithms and their performance. We have discussed PRAM model and PRAM algorithms also. Various searches, sorting and graph based parallel algorithms have also been discussed.

MULTIPLE CHOICE QUESTIONS (MCQS) WITH ANSWERS

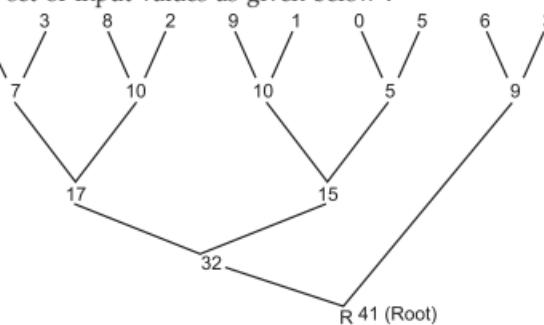
ANSWERS

1. (b) 2. (c) 3. (a) 4. (b) 5. (c)
6. (a) 7. (d) 8. (a) 9. (c) 10. (a)
11. (c) 12. (a)

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

1. The most important paradigm for parallel computing is a binary tree. Why ?

Ans. Consider a set of input values as given below :



Note here that the data flow is from leaves to root *i.e.*, Bottom-up. This is called as **Fan-in or reduction operation**. We have simply mapped an array to a tree which is straight forward. Hence, it is true.

2. Write a sequential-code for processing of the student grades (on a uniprocessor). How will you parallelize your code for a message-passing system ?

[KUD, BE (CSE), 8th sem ; 1995, 1996]

Ans. Firstly, we need to write a sequential code for student's grade processing :

```
begin {Main}
    no_of_students ← 0;
    sum_marks ← 0;
    sum_sq_marks ← 0;
    While not eof do
        Read (Roll_No, Marks[i] for i = 1 to 4);
        Total_marks ← 0;
        no_of_students ← no_of_students+1;
        for i = 1 to 4 do
            Total_marks ← Total_marks+Marks[i]
        endfor;
        Find_class (input:Total_marks, output: class);
        Write Roll_No, (Marks[i] for i = 1 to 4), Total_marks, class;
        Find_statistics (input:Total_marks,
                         output:sum_marks,sum_sq_marks);
        Find_histogram(input:Total_marks,output:Marks_distn[1:10]);
    endwhile;
    class_avge ← sum_marks/no_of_students;
    std_dev ← sqrt(sum_sq_marks/no_of_students)
        – (class_avge ** 2));
    Write class_avge,std_dev, (Marks_distn[j] for j = 1 to 10)
end {Main}.

Find_class(input:Total_marks, output:class);
begin {Find_class}
    if total_marks > = 60 then class = 1
    else
        if total_marks > = 50 then class = 2
        else class = 0
        endif
    endif
    endif
end ;{Find_class}

Find_statistics (input:Total_marks, output: sum_marks,
                 sum_sq_marks);
begin {Find_statistics}
    sum_marks ← sum_marks + total_marks;
```

```

        sum_sq_marks ← sum_sq_marks + Total_marks ** 2
    end ; {Find_statistics}
    Find_histogram (input:Total_marks, output: Marks_distn[1:10]);
    begin {Find_histogram}
        slot_found ← false; limit ← 10; i ← 1 ;
        repeat
            if Total_marks < = limit then
                begin
                    Marks_distn[i] ← Marks_distn[i] + 1 ;
                    slot_found ← true
                end
            else
                begin
                    i ← i + 1 ;
                    limit ← limit + 10
                end
            endif
        until i > 10 or slot_found ;
    end; {Find_histogram}

```

Next we write it's message passing program. We assume that there are 4 CEs in the system. The input student file is read by CE0 and records are distributed equally among CE1, CE2 and CE3. Each CE receives the record sent by CE0. CE0 is assumed to be the “**master CE**” which receives results computed by the other CEs for the records received by them, consolidates and writes the results. Hence, we write this program :

Procedure for CE0

```

begin {CE0 procedure}
    no_of_students ← 0 ;
    sum_marks ← 0
    sum_sq_marks ← 0
    while not eof do
        { We assume that the number of student records is divisible by 4 }
        for i = 1 to 3 do
            Read (Roll_No, Marks[j] for j = 1 to 4) ;
            Send (i, Roll_No, 1) ;
            Send (i, Mark[1], 4)
        endfor ;
        Read(Roll_No, Marks[i] for i = 1 to 4) ;
        Total_marks ← 0; no_of_students ← no_of_students + 1
        for i = 1 to 4 do
            Total_marks ← Total_marks + Marks[i]
        endfor ;

```

```
Find_class(input:Total_marks, output:class) ;
Write Roll_No, (Marks[i] for i = 1 to 4),
      Total_marks, class ;
Find_statistics (input:Total_marks, output:sum_marks,
                sum_sq_marks) ;
Find_histogram(input:Total_marks, output:
               Marks_distn [1:10]) ;
endwhile ;
{Remarks: The following statements are added for CEO in the Parallel Program.}
k ← 3 ;
for i = 1 to k do
  Receive (i, part_sum,1) ;
  sum_marks ← sum_marks + part_sum ;
  Receive(i,part_sum_sq, 1) ;
  sum_sq_marks ← sum_sq_marks + part_sum_sq ;
  Receive(i,Marks_hist[1], 10) ;
  for j = 1 to 10 do
    Marks_distn[j] ← Marks_distn[j] + Marks_hist[j]
  endfor ;
  Receive(i,part_stud,1) ;
  no_of_students ← no_of_students + part_stud ;
  endfor ;
{ Remarks : The following statements are as in sequential Procedure }
class_avge ← sum_marks/no_of_students ;
std_dev ← sqrt((sum_sq_marks/no_of_students
                 - class_avge ** 2)) ;
Write class_avge, std_dev, (Marks_distn[j] for j = 1 to 10)
end {Procedure CEO}
{ Remarks : Procedures Find_statistics and Find_histogram are same as in Procedure.
}

Procedure for CE1, CE2 and CE3
begin
  no_of_students ← 0 ; sum_marks ← 0 ;
  sum_sq_marks ← 0 ;
  while not eof do
    Receive (0, Roll_No, 1) ;
    Receive (0, Marks[1],4) ;
    Total_marks ← 0
    no_of_students ← no_of_students + 1 ;
    for i = 1 to 4 do
      Total_marks ← Total_marks + Marks[i]
    endfor ;
    Find_class (input:Total_marks, output:class) ;
```

```
Write Roll_No, (Marks[i] for i = 1 to 4),
      Total_marks, class ;
Find_statistics (input: Total_marks, output: sum_marks,
      sum_sq_marks) ;
Find_histogram (input: Total_marks, output: Marks_distrn[1:10]) ;
endwhile ;
{ Remarks : The following statements are added for CE1, CE2 and CE3. }
      Send (0, sum_marks, 1) ;
      Send (0, sum_sq_marks, 1) ;
      Send (0, Marks_distrn[1], 10) ;
      Send (0, no_of_students, 1) ;
{ Remarks : clss_avge, std_dev are found by CE0 using the data sent above. }
end. { CE1, CE2, CE3 procedures }
```

Please note that CE1, CE2 and CE3 send the results computed by them to CE0 which receives them and adds them to the results calculated by itself and writes the results. The task granularity is coarse and the time to communicate results is very small. In this method we have done a static assignment of task to CEs.

Observe that the procedure can be easily extended to any number of CEs by assigning appropriate value to k . If k is very large then the system will work well only if the input file has a number of records $\gg k$.

3. Develop an algorithm to add all the elements of the array using ' p ' CEs.

Ans. The procedure for adding elements of an array, $A[1:n]$ is given below :

Program for CE0

```
Array A[1 : N], Sum[0:p - 1];
begin
  for i = 1 to N do
    Read A[i]
  endfor;
  increment ← N div p ;
  last_inc = increment + N mod p;
  for j = 1 to (p - 2) do
    Send (j, A[j * increment + 1], increment)
  endfor;
  Send (p - 1, A[p - 1] * increment + 1, last_inc);
  Sum[0] ← 0;
  for i = 1 to increment do
    Sum[0] ← Sum[0] + A[i]
  endfor;
  for j = 1 to (p - 1) do
    Receive (j, Sum[j], 1)
  endfor;
  grand_sum ← 0;
  for j = 0 to (p - 1) do
```

```

        grand_sum ← grand_sum + Sum[j]
    endfor;
    Write grand_sum
end.

Procedure for CEk, k = 1, ..., (p - 2)
    Array B[1:increment];
begin
    Receive (0, B[1], increment);
    Sum[k] ← 0; {Sum[k] is the partial sum of CE k}.
    for i = 1 to increment do
        Sum[k] ← Sum[k] + B[i]
    endfor;
    Send (0, Sum[k], 1)
end.

Procedure for CE (p - 1)
    Array B[1:last_inc];
begin
    Receive (0, B[1], last_inc);
    Sum[p - 1] ← 0;
    for i = 1 to last_inc do
        Sum[p - 1] ← Sum[p - 1] + B[i]
    endfor;
    Send (0, Sum[p - 1], 1)
end.

```

Please note here that when the array size, N is not evenly divisible by N then the integer quotient ($N \text{ div } p$) is allocated to CE0 to CE($p - 2$) and ($N \text{ div } p + N \text{ mod } p$) is allocated to CE ($p - 1$)

Worst case : In worst case, CE ($p - 1$) is allocated ($N \text{ div } p + p - 1$) array elements.

If $N \gg p$ then $N \text{ div } p \gg p$ and the extra load on CE ($p - 1$) compared to the load on other CEs is not significant.

4. Develop an algorithm/procedure for shared memory multiprocessor to process student grades. [KUD, B.E (CSE) 8th sem ; 1996]

Ans. A shared memory multiprocessor procedure for processing students grades is as follows :

```

begin
    global Total_marks, no_of_students;
    while not eof do
        Read (Roll No, Marks[i] for i = 1 to 4);
        Total_marks ← 0;
        no_of_students ← no_of_students + 1;
        for i = 1 to 4 do
            Total_marks ← Total_marks + Marks[i]

```

```

        endfor ;
        fork (1) Find_clas (input:Total_marks, output: Class) ;
        fork (2) Find_statistics (input:Total_marks, output: sum_marks, sum_sq_marks) ;
        fork (3) Find_histogram (input:Total_marks, output: marks_distrn [1 : 10])]
join 1 ;
    join 2 ;
    join 3 ;
    Write Roll No, (Marks[i] for i = 1 to 4), Total_marks, class ;
endwhile ;
    class_avge ← sum_marks/no_of_students ;
    std-dev ← sqrt(sum_sq_marks/no_of_students—(class_avge ** 2)) ;
    Write class_avge, std_dev,
        (Marks_distrn[j] for j = 1 to 10) ;
end.

```

5. What is the difference between micropipelining and macropipelining ?

Ans. The finest level of pipelining is called **micropipelining** with a subdivision of pipeline stages at the logic gate level.

On the other hand, the **coarse level** for pipeline stages can be conducted at the **processor level** called **macropipelining**.

EXERCISE QUESTIONS

1. (a) What is an ideal situation for user in programming parallel computer ?

Example :

- (b) Give a brief account of the salient features of FORTRAN 90.

[KUD, B.E. (CSE); 8th sem; 1997]

2. Write short notes on :

(a) Programming of message-passing systems.

(b) OCCAM for transputers.

(c) Parallel processing activity in India context.

[KUD, B.E. (CSE); 8th sem ;1997]

3. (a) What do you mean by tuple space model of parallel programming ? Write a Linda program for any task graph you assume.

(b) Discuss formulation of parallel algorithms.

[KUD, B.E. (CSE); 8th sem ;1995]

4. Write short notes on :

(a) Transputer series of processors and its programming languages.

(b) Parallel prefix-sum algorithm for hypercube multicomputers.

[DU ; M.E. (CS)-DCE, 2nd sem ; 2002]

5. Explain “all pairs shortest path algorithm” for parallel processors. What is the complexity of your algorithm ? [DU ; M.E. (CS)-DCE, 2nd sem ; 2003]

6. Explain important features of parallel programming language considering OCCAM. [DU ; M.E. (CS)-DCE, 2nd sem ; 2003]

7. What type of FORTRAN 90 programming model is ? Illustrate any four important

features of FORTRAN 90 that are not found in Fortran 77.

[DU ; M.E. (CS)-DCE, 2nd sem ; 2003]

8. Give a parallel algorithm for finding the sum :

$$S = \sum x_i y_i$$

where $X = (x_1, x_2, \dots, x_n)$ and

$$Y = (y_1, y_2, \dots, y_n)$$

Assume that you have p -processors and that p divides n . Explain your algorithm in the following steps :

- (i) Basic idea.
- (ii) Time complexity ?

9. Explain the following terms :

- (a) Synchronous and asynchronous message passing scheme.
- (b) Blocking and non-blocking communication.

[Pune Univ., BE (CSE) ; 2nd sem ; 2004]

10. Show parallel addition in a hypercube and 2D mesh.

[UPTU, B. Tech. (CSE) 8th Sem.; 2005-06]

11. What do you understand by PRAM algorithms. Discuss and explain with suitable example about PRAM algorithms for merging two sorted lists.

[UPTU, B. Tech. (CSE) 8th Sem.; 2008-09]



11 MULTITHREADED ARCHITECTURE

11.0 INTRODUCTION

Multithreading is a technique that allows multiple threads within a single process to execute independently and share the resources of the process. When the concept of multithreading is applied to a single processor then it causes the single processor to switch between the contexts efficiently but at a time executes only one thread.

But when the concept of multithreading is applied to a parallel computing system consisting of multiprocessors then each processor may execute independently, single thread of one process at the same time. Thus, the multiple processors may execute all threads of process concurrently and hence the over all processing will speed up.

The basic principle behind multithreading is to hide the memory latency in scalable parallel systems by switching between threads. When the current thread requires a remote memory reference then instead of waiting for that memory access, the PE switches to execute another thread. So, memory latency is avoided. Figure 11.1 below shows the model of multithreading in multiprocessor system :

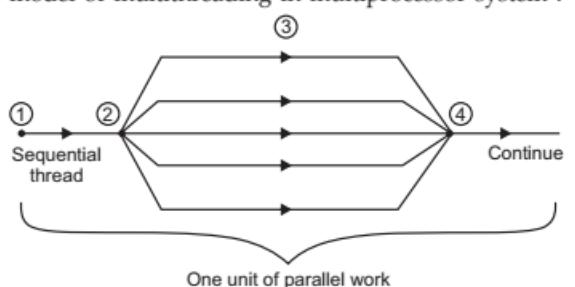


Fig. 11.1. Multithreading in Multiprocessor.

Here, we consider one unit of parallel work. A single process contains many threads. The processing starts with the sequential thread at (1). as shown in Fig. 11.1. Then the scheduler plays an important role and schedules multiple threads at (2). The processors begin execution of threads in parallel, as represented in (3). During the concurrent execution of threads, they may communicate by passing the messages. Finally, as shown, at (4), the multiple threads are synchronized and one unit of parallel work is completed.

11.1 LATENCY HIDING TECHNIQUES

Basically there are many scalable parallel systems which use distributed memory architecture. As the memory is distributed in these systems, so any PE in multiprocessor system may access the remote memory. As the memory is remote (*i.e.*, not local to PE), so some time is required to fetch that memory data. This time delay is called as **memory latency**. This delay is due to the delays of interconnection hardware, communication delay and memory response time. **Please note that even today, the processor is much faster than memory.** That is, the memory response is slow. All these factors cause a long latency in accessing the remote memory ('remote' means 'far').

In summary, we list some important factors that contribute to increase in memory latency :

1. Memory is not physically close to the processor.
2. The speed of memory is less compared to that of the processor.
3. Communication delay.
4. The speed of interconnection network between processor and memory is less.

Solution to the latency problem

To improve the performance of large scale computer systems, it is necessary to enhance their scalability and programmability. As far as the programmability is concerned, the effective performance of the program depends upon many factors and memory latency is one of them. There are many ways to tackle this problem of latency. Out of this, the three important mechanisms are as follows :

- (a) Latency avoidance mechanism.
- (b) Latency reduction mechanism.
- (c) Latency hiding mechanism.

We shall explain all these mechanisms one by one now.

I. Latency Avoidance Mechanism

As we have already studied that the program has some properties like temporal locality and spatial locality. **Latency Avoidance mechanism emphasizes on improving and enhancing the temporal and spatial locality of the program.** This will reduce the cache miss operations and hence latency is avoided.

There are many techniques to avoid latency. One of the simplest technique is to properly organize the user application. So as to improve the localities. This technique requires that the parallel computing system should provide the supporting architecture such as cache coherency techniques, fast messaging technique.

In yet another technique, user can explore the locality in the program by use of specific languages. The programmer can represent certain data structures in a way to improve the locality properties. The other techniques involve the compiler optimization to improve the locality.

II. Latency Reduction Mechanism

As discussed earlier also, the latency avoiding mechanism can be effective upto certain extent. **At some critical point, it becomes difficult to enhance the data locality.** At this stage, one must give attention on reducing the latency. The latency can be reduced by using appropriate communication hardware and software. The scalable parallel system should use efficient communication hardware such as network interfaces and interconnection network.

III. Latency Hiding Mechanism

This technique emphasizes on hiding the latency by the use of overlapping operations during latency. This means that the operation of remote memory access is overlapped with another useful operation by the processor. Hence, the processor need not wait till the remote data is accessed.

There are four (4) main techniques to hide latency. They are as follows :

- (a) Data prefetching techniques.
- (b) Use of coherent caches.
- (c) Use of relaxed memory consistency models.
- (d) Use of multithreading to do context switching to hide the latency.

We shall describe these techniques one by one now.

(a) Data prefetching techniques : Herein, the required data for the program is to be prefetched and brought closer to the processor in advance. To do this, first it is necessary to gather information about such remote references in advance and then start fetching that remote data. Then, move this prefetched data, close to the processor before that data is actually required. In other words, **this technique finds out the expected data misses in the program, then accordingly prefetches that data and places it close to the processor before that data is really needed.**

There are two types of prefetch techniques, namely :

- Binding prefetch.
- Non-binding prefetch.

Binding prefetch : There is a binding on the prefetch operation. When the required data is prefetched in advance then it must be loaded directly into the registers *i.e.*, there should not be a time interval between prefetch and actual reference. Otherwise any other processor may modify the prefetched value before it is referenced by the processor which actually needs it. This indirectly decides the moment at which the binding prefetch can be issued.

Non-binding prefetch : In this technique, the prefetched data is placed in the cache in advance before it is referred and it is maintained consistent by cache coherence mechanism. As the prefetched data maintains the consistency, there will not be any problem even if the processor reads the referenced value later.

The prefetch technique may be controlled by hardware or by software.

Thus, the prefetching techniques definitely hides the latencies due to remote access.

(b) Use of coherent caches : In large-scale computers, the caches are distributed and they are made coherent by implementing cache coherence protocols. **Please note that the small-scale multiprocessor systems use snoopy protocol whereas large scale systems use directory based protocol. Coherent caches help to hide latency by increasing the cache hit ratio for read operations.** When the cache miss occurs, the number of cycles are wasted. The benefit of using distributed coherent caches is to reduce the cache miss.

(c) Use of relaxed memory consistency models : There are different memory models developed for shared-memory multiprocessor systems. These memory models specify the ordering of global memory reads and memory writes generated by the same processor or different processors. The ordering of these reads/writes may cause the consistency problem. A memory consistency model indicates how the memory reads or writes generated by one processor should be observed by other processor in the multiprocessor system.

There are sequential consistency and relaxed consistency memory models. In **sequential consistency model**, the ordering of shared memory reads, writes and exchanges by all processors in system must be in a manner to maintain the program order in the individual processor. In **relaxed memory model**, the sequential constraint of ordering is relaxed. The different relaxed memory models are **weak consistency model**, **processor consistency model** and **release consistency model**.

The **weak consistency model** has only restricted the sequential ordering to synchronizing variable which are recognized by hardware.

But between the synchronization points, this model relaxes the ordering of read/write operations i.e., read/write operations may not follow program order.

In **processor consistency relaxed model**, there is need of a sequential consistency for writes issued within each processor but writes from different processors may be out of program order. This model relaxes the write orders from different processors. It enables the pipelined memory accesses which hides the latency.

The **release consistency model**, also allows buffering and pipelined memory accesses between two synchronization accesses. It is useful in hiding the write latency.

Conclusions : Thus, the relax memory consistency models relax the ordering of memory events such as reads, writes to some extent. This relaxation allows the reordering of memory reads/writes and facilitates the buffering and pipelined memory accesses. This reduces the long latencies of remote memory accesses.

(d) **Use of multithreading to do context switching to hide the latency :** The **multithreaded processor**, also called as the **multiple context processor** is designed in a way to support multithreading. The processor has multiple sets context storage. Therefore, when the processor executes a single thread, which requires a data from remote memory, then this processor does not have to wait. It simply switches to another thread and executes it. Thus, the memory access waiting period due to one thread is overlapped by the execution of another thread.

If this new thread also generates cache miss i.e., if the required data is not in the cache then processor again switches to new thread and starts executing it. By that time, memory data required by initial thread may be available. Thus, the processor always remain busy in doing important work during memory latency.

11.2 PRINCIPLES OF MULTITHREADING

We have already seen a multithreading computation model in previous section. Now, let us examine a multithreaded system, architecture environment. It is shown in Fig. 11.2 below.

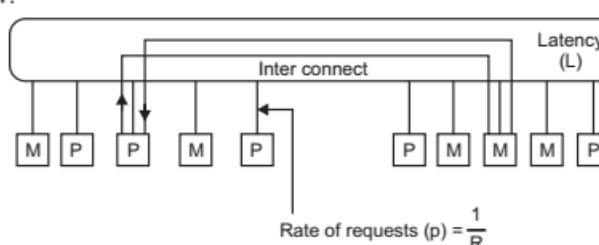


Fig. 11.2. The architecture environment.

The distributed memories form a global address space. Four machine parameters are defined below to analyze the performance of this network :

1. **Latency (L)** : This is the communication latency on a remote memory access. The value of L includes the network delays, cache-miss penalty and delays caused by contentions in split transactions.
2. **The number of threads (N)** : This is the number of threads that can be interleaved in each processor. A thread is represented by a context, consisting of a program counter (PC), a register set and the required context status words.
3. **The context switching overhead (C)** : This refers to the cycles lost in performing context switching in a processor. This time depends on the switch mechanisms and the amount of processor states devoted to maintaining active threads.
4. **The interval between switches (R)** : This refers to the cycles between switches triggered by remote reference. The inverse, $P = \frac{1}{R}$ is called the rate of requests for remote accesses. This reflects a combination of program behavior and memory systems design.
Please note that in order to increase efficiency, one approach is to reduce the rate of requests by using distributed coherent caches. Another approach is to eliminate processor waiting through multithreading.

11.3 MULTITHREADING ISSUES AND SOLUTIONS

Now we have understood the basics of multithreading and the architecture of processor required for multithreading. There are some issues in multithreading that need to be discussed here. These issues are directly related to the overall performance achieved due to multithreading. These issues are as follows :

- | | |
|--------------------------------|--------------------------------------|
| 1. Latency. | 2. Number of threads and efficiency. |
| 3. Context switching overhead. | 4. Context switching policy. |

We shall discuss each one by one now.

1. Latency : The memory latency slows down the efficiency of the processor. Hence, multithreading is one of the ways to hide the memory latency. The operations are overlapped with memory latency and ideally the processor always remains busy. Thus, the processor's efficiency is boosted.

2. Number of threads and efficiency : As already discussed, the multithreaded processor is always kept busy by switching among the multiple contexts. In general, the efficiency of the processor is given by the formula :

$$\text{Efficiency} = \frac{\text{Processor busy time (or useful time)}}{\text{Processor busy time} + \text{Context Switching time} + \text{Processor idle time}} \quad \dots(1)$$

Idle time is not always negligible because of many reasons. In the worst case, it may happen that all threads/context may be blocked and hence processor becomes idle.

The efficiency is very much dependent upon the total number of threads in a process. If the number of threads is less, then the processor may remain idle for some duration after completing switching among all available threads. But, if the number of threads is very large then the processor continues switching among multiple contexts for long duration. In this case, the multithreaded processor continues context switching even after the memory latency is over. Thus, the processor goes into the saturation state of thread switching. Please note that in saturation state, the efficiency of the processor remains constant.

Therefore, there is a threshold value for the total number of contexts. **Below the threshold value**, the efficiency of the processor is calculated as per equation : (1) given above. **Above the threshold value**, the saturation of the number of switches takes place and the efficiency is given by the formula :

$$\text{Efficiency} = \frac{\text{Processor busy time (or useful time)}}{\text{Processor busy time (or useful time)} + \text{Context Switching time}} \quad \dots(1)$$

3. Context switching overhead : When the processor does context switching there are some cycles lost in saving the previous context and loading the new context. This context switching overhead affects the efficiency of multithreaded processor. Large context switching overhead drastically slows down the efficiency. **Practically, it is observed that context switching overhead should not be more than 2 cycles for achieving good efficiency.**

4. Context switching policy : There are many policies for context switching but one has to consider the effect of policy or the efficiency of the processor.

The context switching policy may affect the spatial locality in cache and pipeline dependencies. It may degrade the efficiency due to pipeline flushing.

Solutions to achieve effective multithreading

1. Selection of appropriate context switching policy.
 2. The total number of threads should be at the threshold value.
 3. Minimum context switching overhead.

11.4 MULTITHREADED ARCHITECTURES

From the point of view of multithreaded architectures, three computational models are relevant:

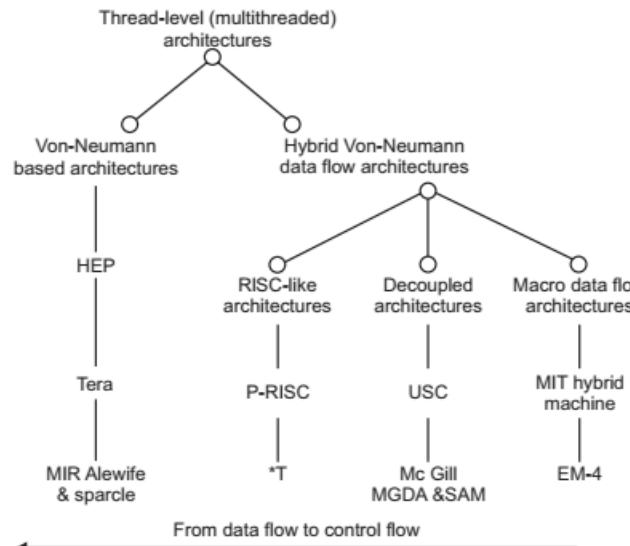


Fig. 11.3. Classification of multithreaded architectures

- (a) Von-Neumann model.
 - (b) Data flow model.

(c) Hybrid Von-Neumann/data flow model.

The classification of multithreaded architectures is shown in Fig. 11.3.

We shall discuss these models one by one now.

I. Sequential flow control/Von-Neumann/conventional model

In Von-Neumann computational model, the flow of control and data are separated. Program instructions should be executed sequentially, from top to bottom. Any deviation from this order should be explicitly defined by special control instructions like GOTO, JUMP, CALL and so on. The hardware support for this sequential execution scheme consists of the program counter (PC) which is automatically incremented after executing ordinary instructions.

II. Data flow model

In this model, the control is tied to the flow of data. The order of instructions in the program text has no effect on the execution order. An instruction is executed if all data needed for execution is available. Since data can be available for several instructions at the same time, these instructions can be executed in parallel.

III. Hybrid model

They combine features of both the data flow and control flow models. Hybrid multithreaded machines can be classified according to their position in the scale of computational models. **Macro data flow architectures** are the closest to the original data flow machines.

Decoupled architectures are based on the control token concept and present the next step towards control flow machines.

RISC like hybrid architectures are the closest relatives of the original control flow machines. They work on a parallel control flow model based on parallel control operators.

11.5 CLUSTER COMPUTING (CC)

When two or more computers are used together to solve a problem, it is considered as a computer cluster. It is a collection of interconnected stand alone computers working together as a single, integrated computing resource.

11.5.1 Characteristics of CC

In general, the following are the characteristics of cluster computing :

1. It consists of many of the same or similar type of machines (Heterogeneous clusters are a subtype, still mostly experimental).
2. They are tightly-coupled using dedicated network connections.
3. All machines share resources such as a common home directory.
4. It must have software such as an MPI implementation installed to allow programs to be run across all nodes.

Cluster consists of :

- Nodes
- Cluster middleware
- Network
- OS

Standard Components

- Avoiding expensive proprietary components.

11.5.2 Why use a cluster ?

There are two reasons :

1. One wants to achieve **High Availability (HA)** meaning getting higher reliability or one wants **High Performance Computing (HPC)**, which is used to get greater computing power than a single machine can provide.
2. A cluster of the same size and power as a mainframe is many times cheaper than a mainframe. This is also a big reason for the same.

11.5.3 Types of Clusters

1. **High Performance Clusters (HPC)**
 - (a) Parallel, tightly coupled applications
 - (b) Example : AKA Beowulf.
2. **High Throughput Clusters (HTC)**
 - Large number of independent tasks.
3. **High Availability Clusters (HA)**
 - (a) Generally small, less than 8 nodes.
 - (b) Multiple communication paths.
 - (c) Used for mission critical applications.
4. **Load Balancing Clusters**
Web Servers, Mail Servers.
5. **Hybrid Clusters**
Example : HPC + HA

11.5.4 How to do it ?

The minimum size of a cluster is just two computers, a setup like that is called as a **two-node cluster**. If you add more computers or nodes then you get a **multi-node cluster**. A two-node cluster is usually used for **HA**. The computers involved can be set up in two different ways :

1. **Active or passive** : Where one of the computers is doing all work and the other one is just a backup in case of a failure.
2. Another way to set them up is the **active/active way**, where both computers do some work. In case of a failure, the other computers must take over the failed computers work as well. **This process of taking over a failed node's work and/or resources is called as a failover.**

Almost all cluster projects were started with the goal to find something cheaper but equally fast as a supercomputer. Both **Beowulf** that started out as an idea in late 1993 and **ARCNET** started this way.

Example of clusters : The first commercially available cluster protocol was **ARCNET (Attached Resource Computer Network)**. It was developed by **Data point corporation** in 1977. The original goal with ARCNET was to provide an alternative to the larger and more expensive systems available at the time. The protocol was run on a single machine with several terminals attached to it. When the number of users became too large, additional computers could be attached as well as more disk resource computers. When microcomputers took over, **ARCNET** was reused as a cheap LAN.

11.5.5 Pros and Cons of CC

Pros of Cluster Computing :

1. Clusters are phenomenal price/performance computational engines.
2. Mainstream tools for a variety of scientific fields.
3. Expanded performance from HPC to :
 - (a) High availability
 - (b) Visualization.
4. Benefits come due to :
 - (a) Using inexpensive commodity servers.
 - (b) Open source software.
 - (c) Large and expanding community of developers.

Cons of Cluster Computing

1. They can be hard to manage without experience.
2. High-performance I/O is still unsolved.
3. Programming environment could be vastly improved.
4. Technology is changing very rapidly. Scaling up is becoming a common place.
5. The largest problem in clusters is **software skew**. That is, when the software configuration on some nodes is different than on others, even small differences can cripple a parallel program.

11.6 NEURAL COMPUTING

Neural computers are trained (not programmed) so that given a certain starting state (data input), they either classify the input data into one of the number of classes or cause the original data to evolve in such a way that a certain desirable property is optimized.

A Neural computing paradigm is shown below. in Fig. 11.4.

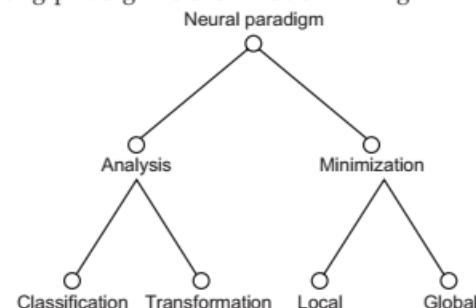


Fig. 11.4. Neural Computing Paradigm

Classification—Involves many inputs and few outputs trained.

Transformation—Involves many inputs and many outputs trained.

Local—Involves many inputs and few outputs rule-based.

Global—Involves many inputs and one output rule-based.

Neural computers implement **data parallelism** and **functional parallelism**.

We discuss the paradigms of Neural Computing in detail now.

I. Classification : First, data about a series of objects (the training set) is fed to the computer. This data includes both the parameters used to describe the object and the

specified classification. When the training phase is complete, the computer will decide on categories for new data which include only the parametric description.

II. Transformation : The second paradigm, transformation has many similarities to the ideas of classification.

Classification involves, a reduction in the quantity of information. Instead of thousands of instances of data, classification results in just a few categories. In computer terms, the amount of data can be reduced from megabytes to bytes – an enormous reduction. The purpose of transformation is quite different. It is concerned with changing one representation of data into another. In this process, any analysis of the characteristics of the data is incidental to the real purpose, although it may serve as a useful tool for improving efficiency.

III. Minimization : This third technique, minimization, is used in a rather different way from the other two and demand a different starting point in order to arrive at an understanding of its operation.

By allowing the system to evolve under different conditions from some arbitrary starting configuration, solution very close to the minimum cost can often be found and in some cases the minimum itself can be achieved.

Neural Network Versus Conventional Computers

Conventional computers use algorithmic approach *i.e.*, the computer follows a set of instructions in order to solve a problem. Neural networks process information in a similar way the human brain does. The network is composed of a large number of highly interconnected PEs (**neurons**) working in parallel to solve a specific problem. Neural networks learn by example. They cannot be programmed to perform specific task. **Please note that neural networks and conventional algorithmic computers are not in competition but complement each other.**

We discuss some Neural Network Software packages now.

11.7 NEURAL NETWORK SOFTWARE PACKAGES

The discussion includes major projects aimed at developing neural net software and also Neural Network Software packages already available.

1. Pygmalion

Pygmalion project is one of the major European projects in the field of neural network.

The main objective of Pygmalion is to demonstrate to European Industry, the potential of neural networks in various applications and to develop a European standard Neural Network programming environment.

1.1 Pygmalion Environment.

The Pygmalion environment is designed to achieve two things.

The first one is an open programming environment that can easily be extended and interfaced with other tools.

The second one is to provide portable neural network applications.

Figure 11.5. illustrates the pygmalion neural programming environment.

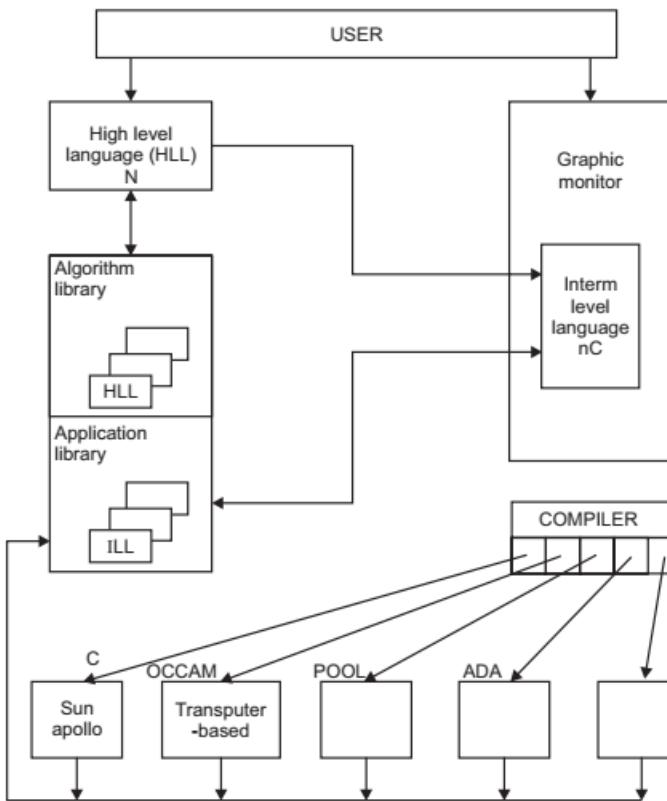


Fig. 11.5. Pygmalion Neural Programming environment.

It has got five major parts.

1. Graphic Monitor
2. Algorithm Library
3. High level Language N
4. Intermediate level language 'C'.
5. Compilers

1.1.1 Graphic Monitor

The graphic monitor is a global software environment used for controlling the execution and monitoring of Neural Network application in simulation.

It also includes a simulation command language for setting up a simulation, monitoring its execution, interactively changing values, and having a trained Network.

Figure 11.6. shows the graphic monitor and the nC specifications. The graphic monitor displays a hierarchy of windows and nC contains a hierarchical data structure including the system, layer, cluster, neuron and synapse.

Graphics monitor displays the contents of data structure, with a window corresponding to each level in the data structure. There are two types of windows :

Top window and Level Window.

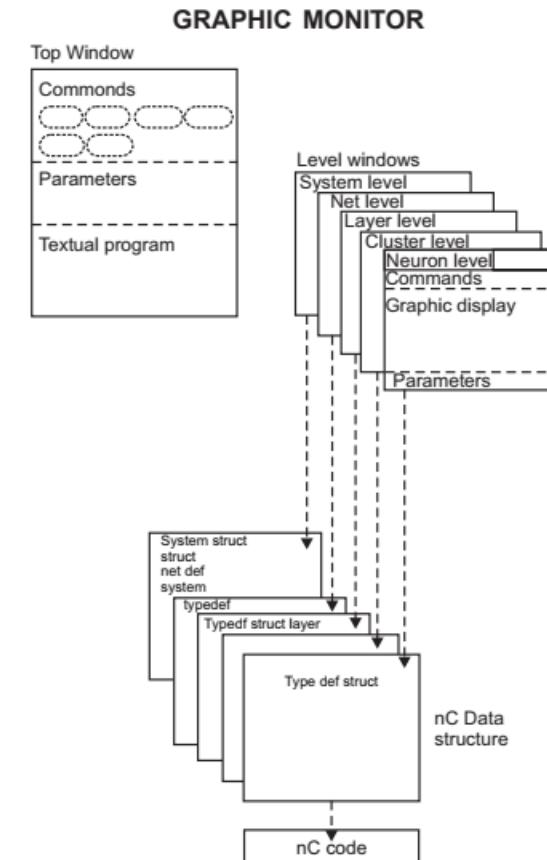


Fig. 11.6. Pygmalion Graphic monitor and nC specifications.

Algorithm Library : The algorithm library is a parameterized library written in a high level language.

It provides the user a number of validated modules for constructing applications.

Galatea is the algorithm C library in the Pygmalion project.

It provides efficient versions of algorithms most commonly used for applications developed within the project.

Galatea Organization : The organization of Galatea is shown in Figure 11.7. There are five components in the Galatea.

1. Algorithm independent path (AIP)
2. Tools Library
3. Algorithm Modules
4. Algorithm evaluation programs
5. Environment Library.

High Level Language N : N : N is a high level neural network programming language (shown in figure 11.7) for both expert and inexperienced users.

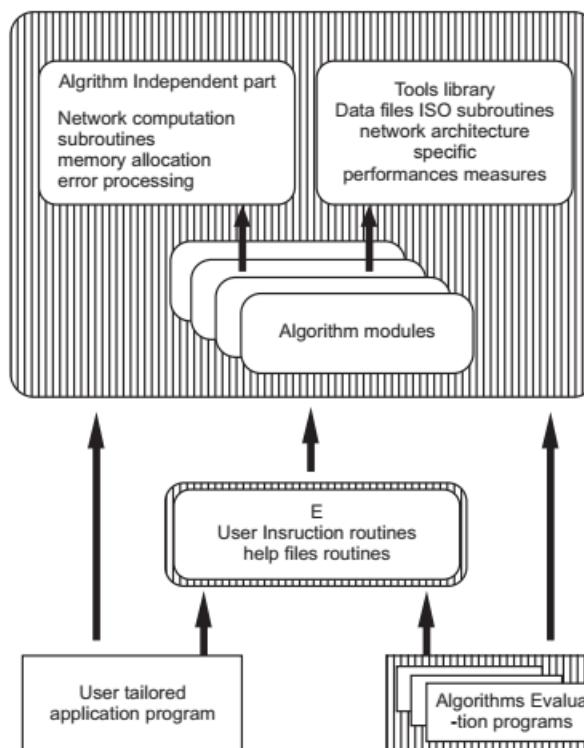


Fig. 11.7. Organization of Galatea.

The object oriented programming language is used for defining and describing the network topology and its dynamics.

It is used with the algorithm library and a neural network algorithm.

It allows development of neural algorithms and use of operation algorithms in applications. Its syntax is a subset of C++ with additional neural oriented features.

A typical N program consists of a list of type definitions. One type may be defined from previously defined ones.

The advantages of this is that, it is easy to convert N program into equivalent nC structure.

Intermediate Level Language 'C' : The intermediate level language nC is the low level machine independent network specification language for representing the partially trained to trained applications. (The nC is based on a small subset of 'C' language).

Compilers : Different compilers are provided to the target Unix-based workstations and parallel transputer based machines.

1.2 Using Pygmalion Environment

Pygmalion environment provides four elementary facilities for programming a neural network application.

Creation : Graphic monitor allows the user to create a neural network system from scratch, using a specific high level language.

Modification : Graphic monitor provides the user with the ability to alter his neural network application during the simulation.

Execution : The graphic monitor enables the user to examine with a way to control the execution of a neural network system.

Monitoring : The status of an executing can be network examined in two ways.

1. Static –by defining the current status.
2. Dynamic –recording the history of the status.

Applications

The way of programming a neural network application using the Pygmalion environment is shown in figure 11.8. A high level language is used to construct a neural network around a parameterized algorithm supplied in the algorithm library. This application part defines the configuration of the algorithm and application specific codes and initial data to the network.

The 'nC' specification is compiled into a binary simulation for specific target machine.

11.8 ANNIE PROJECT (APPLICATIONS OF NEURAL NETWORK FOR INDUSTRY IN EUROPE) PROJECT

The ANNIE project was a combined attempt of nine industrial enterprises and research institutions.

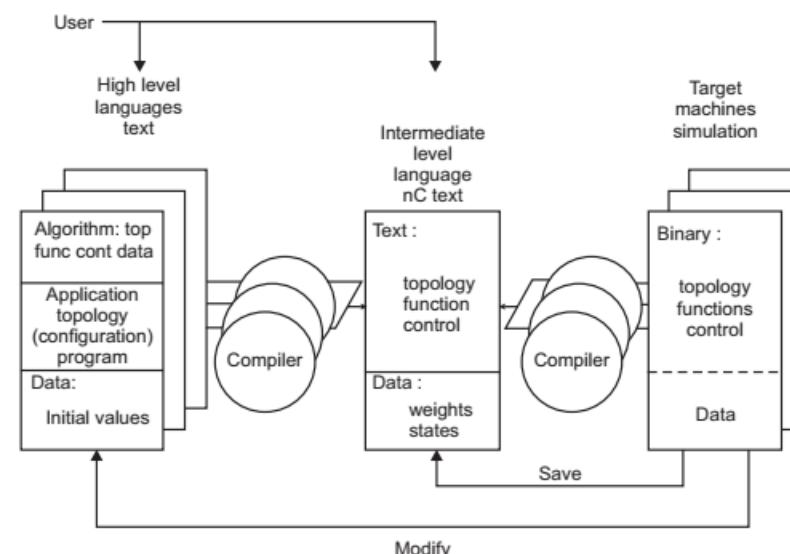


Fig. 11.8. Programming a Neural Application.

The main objective of ANNIE project is to define areas of industrial problems where neural networks perform better compared to the conventional methods.

Figure 11.9. shows the interrelationships between the ANNIE partners and their interests of applications. Their interests were mainly on three fields – Pattern recognition, control and optimization.

Pattern Recognition/Image Processing

Features extraction and Feature processing are the two phases of pattern recognition.

ANNIE project has proved that the artificial Neural Network has not got an advantage over conventional methods by reducing the necessity of feature space analysis.

Control

The possible areas of applications and the desired performance is surveyed in control field. Based on this knowledge and on the technical problems to be found in control, a single prototype system was selected. This aims to tackle a problem which contains maximum possible aspects of control.

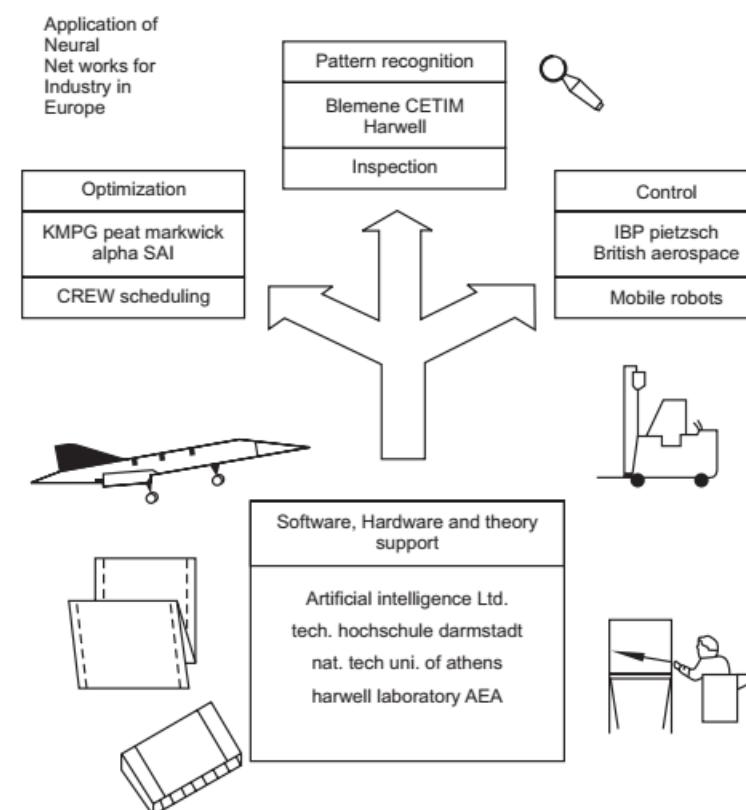


Fig. 11.9. Interests and interactions of ANNIE parents.

Optimization

The main field of application of neural network in **optimization** is the category of hard problems which are not efficiently solved by conventional optimization algorithms.

It was suggested that ANNIE and Pygmalion have to cooperate for the further improvement of testing environment developed in Pygmalion.

3. BRAINTRACER

The braintracer software package is dedicated to the development of neural network with complex structure.

It was inspired by the anatomy of the real brain and aims to simulate their behaviour on computer.

Braintracer allows to create, modify and evaluate the performance of different Network structures rather than activation of different learning algorithms.

3.1 Structure of the Model

The user begins by defining the classes of neurons. A class consists of neurons which are topologically around in bidirectional layer and which have similar projective and receptive fields as well as similar parameters controlling the activation and learning rules.

Connections between neurons are specified by defining mapping between classes *i.e.*, by declaring a set of (source_class, target_class) pairs.

The definition of each mapping automatically establishes the synapses between all neurons belonging to the specific classes.

This is done according to some rules.

3.2 Environment

The network can interact with any simulated environment the user has developed simply by defining a suitable set of receptive and motor class and by preparing a file containing a 'C' source of simple functions.

3.3 Program

The program is written in 'C' and can be run on different systems. The package consists of two executable modules.

1. Interpreter Module.
2. Simulator Module.

Interpreter Module

The function of the interpreter module is to create the files which contain the description of the network. The interpreter interacts with the simulator by providing many commands for the creation, modification and analysis of a network as well as help and syntax checkig facilities.

Simulator Module

The simulator module simulates the evolution of the network.

4. THE CALM SYSTEM (CATEGORIZING AND LEARNING MODULES)

The CALM system was built mainly for the simulation of CALM networks.

The CALM networks are learning neural networks that consist of CALM modules.

A CALM module can categorize and learn an arbitrary activation pattern.

Network Architecture of CALM

A CALM module consists of :

- 2 rows of n nodes.
- R representation nodes.
- V veto nodes.

and an Arousal node (A) and External node (E) and n size of CALM module.

All the nodes are connected by non-modifiable connection according to a certain wiring scheme. So the learning and categorization in CALM are sensitive to the strangeness of the input activation pattern.

Interconnecting two modules F and T means connecting all R-nodes in F to all R-nodes in T.

The main objective of CALM is to ensure users to generate Networks on the basis of high level specifications. For this purpose, an interactive component was developed with a text based user interface.

Limitations of CALM System

1. It lacked graphical facilities.
 2. The control structure of the non-interactive facility was limited.
 3. It could only run CALM network and it did not allow experimentation with other networks.
 4. System itself was very large and only a relatively small network could be run on a PC.
 5. To control processing in various kinds of hardware was not possible.
- To overcome their drawbacks, the MetaNet system was evolved.

5. METANET

MetaNet programming environment is used for developing modular neural networks.

Figure 11.10. shows the Metanet programming environment. It consists of a graphical editor, a network compiler, a graphical de-compiler, a network specification language MetaNet and hardware drive.

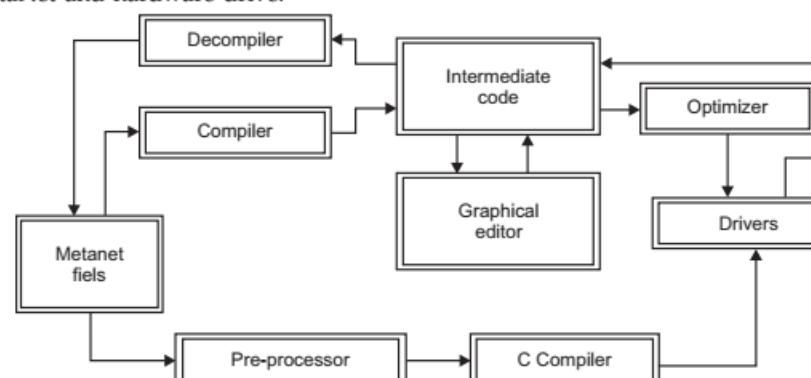


Fig. 11.10. Meta Net Network Environment.

5.1 Graphical Editor

The functions in graphical editor is same as that found in most other programming environments.

The editor enables the user to graphically edit the network.

Nodes, modules and other objects are placed on the screen and the connections can be easily formed using a mouse.

The activation, learning and transfer functions can be chosen from a menu.

"Charged-particle Algorithm" is used for the objects to mimic the motions of charged particles.

5.2 Compiler/De-compiler

Inside the MetaNet environment networks, the control flows are represented in "intermediate code". This code is processed by an optimizer, and sent to hardware drivers which actually run the networks.

Graphical compiler converts the MetaNet files into intermediate code. In otherwords, it converts logical structures into object code.

The graphical de-compiler enables the user to graphically edit network and convert them to MetaNet files.

Intermediate code can be sent to the graphical editor for further changes or it can be sent to the optimizer directly and from there to the drivers.

5.3 MetaNet Language

MetaNet language is a subset of 'C' and it is a simple one. The networks can be defined by using MetaNet objects such as Node, Group, Module in a hierarchical fashion.

The MetaNet's syntax concentrate more on the structure than the process. This enables to perform calculations and control by using only a small subset of 'C'.

5.4 Network Drivers

Network drivers increase the machine independence. This saves a lot of memory since only the relevant drivers need to be loaded.

6. APPLICATION ORIENTED PROGRAMMING ENVIRONMENT

Application oriented programming environment is very user friendly and are completely menu driven. The user does not have to worry about the functioning of complicated neural network part.

Behav Heuristics Airline Marketing Tactician (AMT) and Nestor's Decision Learning System (DLS) come under this group.

6.1 Airline Marketing Tactician (AMT)

The AMT is an Adaptive Decision Making system for airline transportation control. It accesses data on advance booking and reduces the overtime in the allocation of seats between discount and standard fare classes.

The AMT also provides extra capabilities such as the possibility of modifying security, information display and also data inputs to reflect the style of airline management.

6.2 Decision Learning System (DLS)

The DLS is mainly suitable for problems that involve many interdependent variables which are difficult to process using conventional statistical approaches.

The DLS provides a predictive modelling system that learns from experience and can be used objectively to resolve differences between the expert opinions.

The DLS is menu driven. Apart from the programming environment, this package includes a tutorial and example applications.

7. ALGORITHM ORIENTED PROGRAMMING ENVIRONMENT

Algorithm oriented programming environments are designed to support specific neural network algorithm. They are usually supplied in a form that allows easy integration with user applications.

Algorithm-specific environments and the algorithm libraries are the two subclasses of this.

7.1 Algorithm Specific Environment

This consists of only a single neural network model that can be applied to a wide range of applications. However, they are frequently expanded to include other models.

Neuroshell and **BrainMaker** are the examples for algorithm specific environment.

7.1.1 NeuroShell

- NeuroShell is a menu driven software.
- It provides the facility to compare and display the operation of the network with the desired operation.
- It provides an advanced options menu to do extra things.
- NeuroShell provides a utility program that will graph and manipulate the NeuroShell files.
- It can be trained in batch mode.
- It can read Base files directly using the database option.
- To improve the speed NeuroShell can be implemented on an acceleratorboard called **Neuroboard** which works hundred times faster than 80386 machine.

7.1.2 BrainMaker

It has a hypersonic algorithm which is about a hundred to thousand times faster than the backpropagation algorithm, but with 80% accuracy.

7.2 Algorithm Libraries

The algorithm libraries offer a collection of parameterized neural network models presented in a standard language such as 'C'.

Algorithm libraries provide probability of neural network model implementations.

They have the capability of holding common network algorithms in a parameterized form, and can easily be incorporated to user application programs.

7.2.1 Owl (Olmstead and Watkins)

Owl is an algorithm library, written in 'C'. It supports nineteen different neural network models. All models are seen by the programmer in the same way as data structures.

The data structures specify the properties of the network and service calls invoke entry points within the Network objects.

To use the network, the user specifies the object and library modules, and an include file corresponding to the required model and environment.

8. NEURAL WORKS PROFESSIONAL II

To run this software we require an IBM PC AT or compatible machine, with 512 KB main memory. MS-DOS version 3.0 or higher or at least 3 MB RAM.

Neural Works Professional II simulates a large number of neural networks and their learning algorithms. This software provides different menus and submenus.

9. NEURAL WORKS PROFESSIONAL II PLUS

This is an advanced version of neural network works professional II.

It supports IBM RS 6000, i 860 coprocessor and transputer boards.

It simulates several new networks that are not included in Neural Works Professional II.

9. AXON

The axon program describing the neural networks is divided into five major blocks.

- Parameter definition blocks.
- Processing element and layer definition block.
- Creation of the network and connection and definition block.
- Scheduling block.
- Function block.

The syntax of axon is very similar to the 'C' language syntax. Different types of neural network can be constructed easily using this software.

10. NET TALK

Net talk is a speech recognition neural network developed by Torrence Sejnowski and Charles Rosenberg in 1985.

Figure 11.11 shows the Net talk speech system. It consists of Net talk neural network, speech synthesizer, sound generator and a loud speaker.

The text to be spoken is given as the **input** to the synthesizer.

The system receives seven characters only and then instructs the sound system to generate the phoneme sound.

After recording the speech on a tape, it is given as input port to the neural network and is trained for a large set of patterns.

It was observed that the performance of the neural network and the speech synthesizer was almost the same.

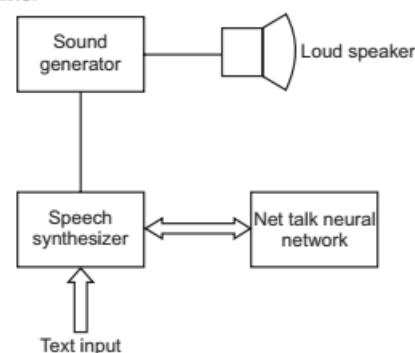


Fig.11.11. NET talk Speech System.

1. INTRODUCTION TO COUNTER-PROPAGATION NETWORKS

The counterpropagation network developed by Robert Hech Nielsen goes beyond the representational limits of single-layer networks. As compared to backpropagation, it can reduce training time by one hundredfold. Counterpropagation is not as general

as backpropagation, but it provides a solution for those applications that cannot tolerate long training sessions. We point out that in addition to overcoming the limitations of other networks, counter-propagation has some interesting and useful features of its own.

Counterpropagation is a combination of two well-known algorithms : the self-organizing map of Kohonen and the Grossberg outstar. Together they possess properties not available in either one alone.

Method such as counterpropagation that combine network paradigms in building-block fashion may produce networks closer to the brain's architecture than any homogeneous structure. It does indeed seem that the brain cascades various specialized modules to produce the desired computation.

The counterpropagation network functions as a **look-up table** capable of generalization. The training process associates input vectors with corresponding output vectors. These vectors may be binary, consisting of ones and zeros, or continuous. Once the network is trained, application of an input vector produces the desired output vector. The generalization capability of the network allows it to produce a correct output even when it is given an input vector that is partially incomplete or partially incorrect. This makes the network useful for pattern-recognition, pattern-completion, and signal-enhancement applications.

2. NETWORK STRUCTURE

Figure 11.12. shows the simplified feed forward version of the counterpropagation network; it illustrates the functional characteristics of this paradigm. The full bidirectional network uses the same principles.

The neurons in layer 0 (shown as circles) serve only as a **fan-out points** and perform no computation. Each layer 0 neuron connects to every neuron in layer 1 (called the **Kohonen layer**) through a separate weight w_{mn} ; these will be collectively referred to **weight matrix W**. Similarly, each neuron in the Kohonen layer (layer 1) connects to every neuron in the Grossberg layer (layer 2) by weight v_{np} ; these comprise the weight matrix V. This looks much like other networks, the difference lies in the processing done by the Kohonen and Grossberg neurons.

As in many other networks, counterpropagation functions in two modes : the normal mode, in which it accepts an input vector X and produces an output vector Y, and the training mode, in which an input vector is applied and the weight are adjusted to yield the desired output vector.

3. NORMAL OPERATION

3.1 Kohonen Layer

In its simplest form, the Kohonen layer functions in a "winner-take-all fashion"; that is, for a given input vector, one and only one Kohonen neuron outputs a logical one; all others output a zero. One can think of the Kohonen neurons as a series of light bulbs, only one of which comes ON for a given input vector.

Associated with each Kohonen neuron is a set of weights connecting it to each input. For example, in Figure 11.12, Kohonen neuron K_1 has weights $w_{11}, w_{21}, \dots, w_{m1}$,

comprising a weight vector w_1 . These connect by way of the input layer to input signals x_1, x_2, \dots, x_m , comprising the input vector x . As with neurons in most networks, the NET output of each Kohonen neuron is simply the summation of the weighted inputs. This may be expressed as follows :

$$\text{NET}_j = w_{1j} x_1 + w_{2j} x_2 + \dots + w_{mj} x_m \quad \dots(1)$$

where NET_j is the NET output of Kohonen neuron j .

$$\text{NET}_j = E_i x_i w_{ij} \quad \dots(2)$$

or in vector notation

$$N = XW \quad \dots(3)$$

Where N is the vector of Kohonen layer NET outputs.

The Kohonen neuron with the largest NET value is the "winner". Its output is set to one; all others are set to zero.

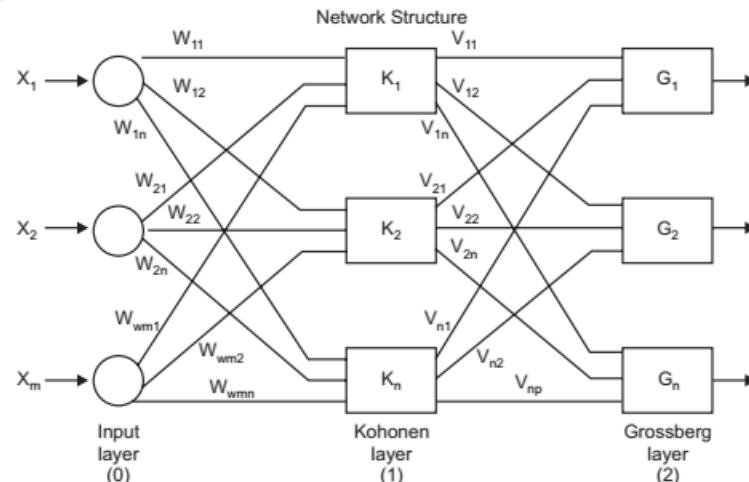


Fig. 11.12. Counterpropagation Network.

3.2 Grossberg Layer

The Grossberg layer functions in a familiar manner. Its NET output is weighted sum of the Kohonen layer outputs k_1, k_2, \dots, k_n , forming the vector K . The connecting weight vector designated V consists of the weights $v_{11}, v_{21}, \dots, v_{np}$. The NET output of each Grossberg neuron is then

$$\text{NET}_j = E_i k_i w_{ij} \quad \dots(4)$$

where NET_j is the output of Grossberg neuron j , or in vector form

$$Y = KV \quad \dots(5)$$

where

Y = the Grossberg layer output vector.

K = the Kohonen layer output vector.

V = the Grossberg layer weight matrix.

If the Kohonen layer is operated such that only one neuron's NET is at one and all other are at zero, only one element of the K vector is nonzero, and the calculation is

simple. In fact, the only action of each neuron in the Grossberg layer is to output the value of the weight that connects it to the single nonzero Kohonen neuron.

4. TRAINING THE KOHONEN LAYER

The Kohonen layer classifies the input vectors into groups that are similar. This is accomplished by adjusting the Kohonen layer weights so that similar input vectors activate the same Kohonen neuron. It is then the responsibility of the Grossberg layer to produce the desired outputs.

Kohonen training is a self-organizing algorithm that operates in the unsupervised mode. For this reason, it is difficult (and unnecessary) to predict which specific Kohonen neuron will be activated for a given input vector. It is only necessary to ensure that training separates dissimilar input vectors.

4.1 Preprocessing The Input Vectors

It is highly desirable to normalize all input vectors before applying them to the network. This is done by dividing each component of an input vector by that vector's length. This length is found by taking the square root of the sum of the squares of all of the vector's components. In symbols,

$$x_i = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2} \quad \dots(6)$$

This converts an input vector into a unit vector pointing in the same direction; that is, a vector of length n -dimensional space.

Equation 6 generalizes the familiar two-dimensional case in which the length of a vector equals the hypotenuse of the right triangle formed by its x and y components, an application of the familiar Pythagorean theorem. In Figure 11.13, such a two-dimensional vector V is drawn on x - y coordinates, where V has x and y components of four and three, respectively. The square root of the sum of the squares of these components is five. Dividing each component of V by five yields a vector V' with components $4/5$ and $3/5$ where V' points in the same direction as V , but is of unit length. This may be verified by calculating the square root of the sum of the squares of the components V' , which equals one.

Figure 11.13 shows some two-dimensional unit vectors. These terminate at points on a unit circle (a circle with a radius of one) which is the situation if there are only two inputs to the network. With three inputs, vectors would be represented as arrows terminating on the surface of a unit sphere. This idea can be extended to networks having an arbitrary number of inputs, where each input vector is an arrow terminating on the surface of a higher dimensional-unit hypersphere (a useful abstraction even if it cannot be visualized).

To train the Kohonen layer, an input vector is applied and its dot product is calculated with the weight vector associated with each Kohonen neuron. The neuron with the highest dot product is declared as the "winner" and its weights are adjusted. Because the dot product operation used to calculate the NET values is a measure of similarity between the input and weight vectors, the training process actually consists of selecting the Kohonen neuron whose weight vector is most similar to the input vector, and making it still more similar. **Note again that this is unsupervised training ; there is no teacher.** The network self-organizes so that a given Kohonen neuron has maximum output for a given input vector. The training equation that follows is used :

$$w_{\text{new}} = w_{\text{old}} + a(x - w_{\text{old}}) \quad \dots(7)$$

where

w_{new} = the new value of a weight connecting an input component x to the winning neuron.

w_{old} = the previous values of this weight

'a' = training rate coefficient that may vary during the training process.

Each weight associated with the winning Kohonen neuron is changed by an amount proportional to the difference between its value and the value of the input to which it connects. The direction of the change minimizes the difference between the weight and its input.

Figure 11.13 shows this process geometrically in two-dimensional form. First, the vector $x - W_{\text{old}}$ is found by constructing a vector from the end of W to the end of x . Next, this vector is shortened by multiplying it by the scalar 'a', a number less than one, thereby producing the change vector s . Finally, the new weight vector W_{new} is a line from the origin to the end of s . From this it may be seen that the effect of training is to rotate the weight vector toward the input vector without materially changing its length.

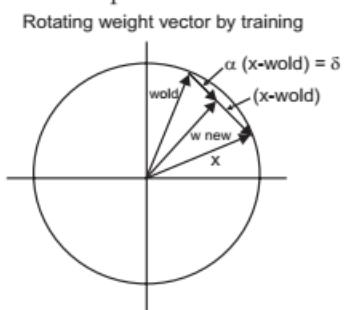


Fig. 11.13. Rotating weight vector.

The variable a is a training-rate coefficient that usually starts out at about 0.7 and may be gradually reduced during training. This allows large initial steps for rapid, coarse training and smaller steps as the final value is approached.

If only one input vector were to be associated with each Kohonen neuron, the Kohonen layer could be trained with a single calculation per weight. The weights of a winning neuron would be made equal to the components of the training vector ($a = 1$). Usually the training set includes many input vectors that are similar and the network should be trained to activate the same Kohonen neuron for each of them. In this case, the weights of that neuron should be the average of the input vectors that will activate it. Setting a to a low value will reduce the effect of each training step making the final value an average of the input vectors to which it was trained. In this way, the weights associated with a neuro will assume a value near the "center" of the input vectors for which that neuron is the "winner".

4.2 Initializing The Weight Vectors

All of the network weights must be set to initial values before training starts. It is common practice with neural networks to randomize the weights to small numbers. For Kohonen training, randomized weight vectors should be normalized. After training, the weight vectors must end up equal to normalized input vectors. Therefore, prenormalization

to unit vectors will start weight vectors closer to their final state and thereby shorten the training process.

Randomizing the Kohonen layer weights can cause serious training problems, as it will uniformly distribute the weight vectors around the hypersphere. Because the input vectors are usually not evenly distributed and tend to be grouped on a relatively small portion of the hypersphere surface, most of the weight vectors will be so far away from any input vector that they will never be the best match. These Kohonen neurons will always have an output of zero and will be wasted. Furthermore, the remaining weights that do become the best match may be too few in number to allow separation of input vector categories that are close together on the surface of the hypersphere.

Suppose there are several sets of input vectors all of which are similar, yet must be separated into different categories. The network should be trained to activate a different Kohonen neuron for each category. If the initial density of weight vectors is two low in the vicinity of the training vectors, it may be impossible to separate similar categories; there may not be enough weight vectors in the vicinity to assign one to each input vector category.

Conversely, if several input vectors are slight variations of the same pattern and should be lumped together, they should fire a single Kohonen neuron. If the density of weight vectors is very high near a group of slightly different input vectors, each input vector may activate a different Kohonen neuron. This is not catastrophic, as the Grossberg layer can map different Kohonen neurons into the same output, but it is wasteful of Kohonen neurons.

The most desirable solution is to distribute the weight vectors according to the density of input vectors that must be separated, thereby placing more weight vectors in the vicinity of large-number input vectors. This is impractical to implement directly, but several techniques approximate its effects.

One solution, called the **convex combination method**, sets all the weights to the same value $1/\forall(n)$, where n is the number of inputs and hence, the number of components in each weight vector. This makes the weight vectors of unit length and all coincident. Also, each component x_i of the input is given the value $ax_i + \{[1/\forall(n)](1-a)\}$, where, n is the number of inputs. Initially a is given a very small value, causing all input vectors to have a length near $1/\forall(n)$ and coincident with the weight vectors. As the network trains, a is gradually increased to a limit of 1. This allows the input vectors to separate and eventually assume their true values. The weight vectors follow one or a small group of input vectors and end the training process by producing the desired pattern of outputs. The convex combination method operates well but slows the training process, as the weight vectors are adjusting to a moving target. Another method adds noise to the input vectors. This causes them to move randomly, eventually capturing a weight vector. This method also works, but it is even slower than convex combination.

A third method starts with randomized weights but in the initial stages of the training process adjusts all of the weights, not just those associated with the winning Kohonen neuron. This moves the weight vectors around to the region of the input vectors. As training progresses, weight adjustments are restricted to those Kohonen neurons that are nearest to the winner. This radius of adjustment is gradually decreased until only those weights are adjusted that are associated with the winning Kohonen neuron.

Still another method gives each Kohonen neuron a "conscience". If this has been winning more than its fair share of the time (roughly $1/k$, where k is the number of Kohonen neurons), it temporarily raises a threshold that reduces its chances of winning, thereby allowing the other neurons an opportunity to be trained.

4.3 Interpolative Mode

Up to this point we have been discussing a training algorithm in which only one Kohonen neuron is activated for each input vector, this is called the "accretive mode". The accuracy of this method is limited in that the output is wholly a function of a single Kohonen neuron.

In the interpolative mode, a group of the Kohonen neurons having the highest outputs is allowed to present its output to the Grossberg layer. The number of neurons in this group must be regarding an optimum size. Once the group is determined, its set of NET outputs is treated as a vector and normalized to unit length by dividing each NET value by the square root of the sum of the squares of the NET values in the group. All neurons not in the group have their outputs set to zero.

The interpolative mode is capable of representing more complex mappings and can produce more accurate results. Again, no conclusive evidence is available to evaluate the interpolative versus the accretive modes.

4.4 Statistical Properties of the Trained Network

Kohonen training has the useful and interesting ability to extract the statistical properties of the input data set. Kohonen has shown that, in a fully trained network, the probability of a randomly selected input vector (selected according to the probability density function of the input set) being closest to any given weight vector is $1/k$, where k is the number of Kohonen neurons. This is the optimal distribution of weights on the hypersphere. (This assumes that all of the weight vectors are in use, a situation that will be realized only if one of the methods discussed is used to distribute the weight vectors.)

5. TRAINING THE GROSSBERG LAYER

The Grossberg layer is relatively simple to train. An input vector is applied, the Kohonen output (s) are established, and the Grossberg outputs are calculated as in normal operation. Next, each weight is adjusted only if it connects to a Kohonen neuron having a nonzero output. The amount of the weight adjustment is proportional to the difference between the weight and the desired output of the Grossberg neuron to which it connects. In symbols,

$$v_{ij} \text{ new } = v_{ij} \text{ old } + B(y_j - v_{ij} \text{ old}) k_i \quad \dots(8)$$

where

k_i = the output of the Kohonen neuron i (only one Kohonen neuron)

y_j = component j of the vector of desired outputs.

Initially B is set to approximately 0.1 and is gradually reduced as training progresses.

From this it may be seen that the weights of the Grossberg layer will converge to the average values of the desired outputs, whereas the weights of the Kohonen layer are trained to the average values of the inputs. Grossberg training is supervised; the algorithm has a desired output to which it trains. The unsupervised, self-organizing operation of the Kohonen layer produces outputs at indeterminate positions; these are mapped to the desired outputs by the Grossberg layer.

6. THE FULL COUNTERPROPAGATION NETWORK

Figure 11.14 shows the full counterpropagation network. In normal operation, input vectors X and Y are applied and the trained network produces output vectors X' and Y' , which are approximations of X and Y , respectively. In this case, X and Y are assumed to be normalized unit vectors; hence, they will tend to produce normalized vectors on the output.

During training, vectors X and Y are applied both as inputs to the network and as desired outputs. X is used to train the X' outputs, while Y is used to train the Y' outputs of the Grossberg layer. The full counterpropagation network is trained using the same method described for the feedforward network. The Kohonen neurons receive inputs from both the X and Y vectors, but these are undistinguishable from a single layer vector composed of the X and Y vectors; thus, this arrangement does not affect the training algorithm.

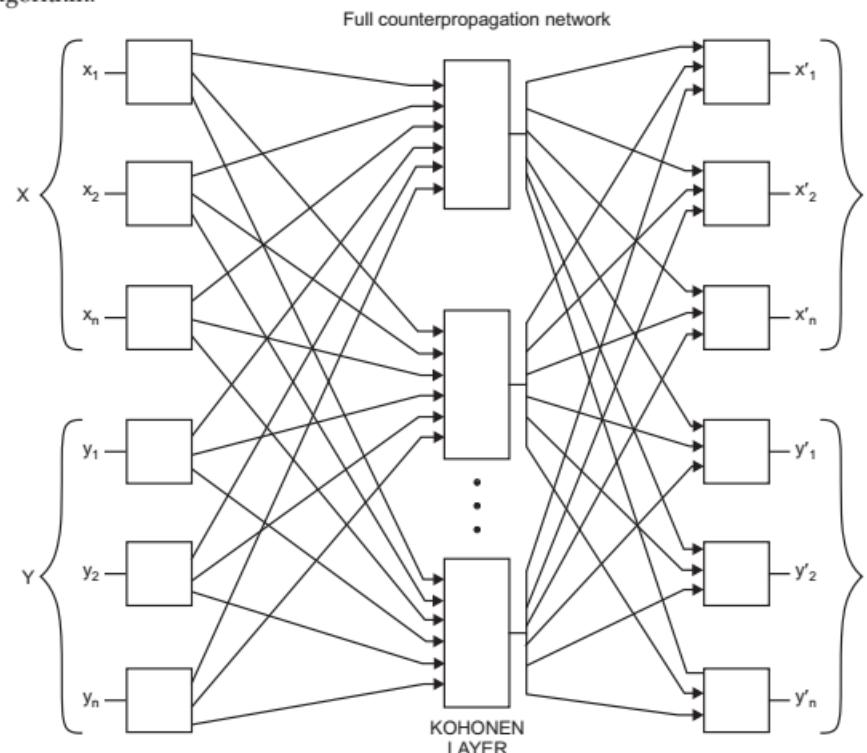


Fig. 11.14. Full Counterpropagation Network.

The result is an identity mapping in which the application of a pair of input vectors produces their replicas on the output. This does not seem very interesting until one realizes that applying only the X vector (with the Y vector set to 0) produces both the X' and the Y' outputs. If F is a function mapping X to Y' , then the network approximates it. Also, if the inverse of F exists, applying only the Y vector (setting X to 0) produces X' . This unique ability to generate a function and its inverse makes the counter-propagation network useful in a number of applications.

Figure 11.14, unlike Hecht-Nielsen's original configuration, does not make apparent the counterflow nature of the network for which it is named. This form was chosen because it also illustrates the feedforward network and facilitates the extension of concepts developed in earlier chapters.

7. AN APPLICATION : DATA COMPRESSION

In addition to usual vector-mapping functions, counterpropagation is useful in certain less obvious applications. One of the more interesting examples is data compression.

A counterpropagation network can be used to compress data prior to transmission, thereby reducing the number of bits that must be sent. Suppose that an image is to be transmitted. It can be divided into subimages S as shown in Figure 11.15. Each subimage is further divided into pixels (picture elements). Each subimage is then a vector, the elements of which are the pixels of which the subimage is composed. For simplicity, assume that each pixel is either one (light) or zero (dark). If there are n pixels in the subimage, substantially fewer bits are actually required to transmit typical images, thereby allowing an image to be transmitted more rapidly. This is possible because of the statistical distribution of subimage vectors; some occur frequently, while others occur so seldom that they can be approximated roughly. The method of vector quantization finds these shorter bit strings that best represent the subimages.

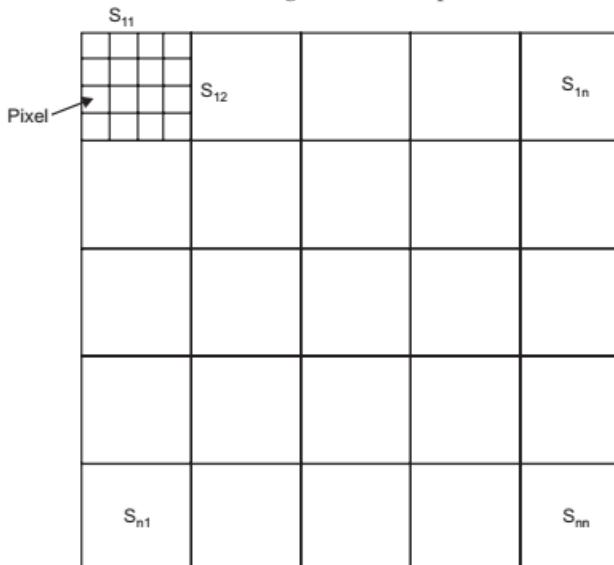


Fig. 11.15. Subimages for images, S .

A counterpropagation network layer can be used to perform vector quantization. The set of subimage vectors is used as input to train the Kohonen layer in the accretive mode in which only a single neuron is allowed to be 1. The Grossberg layer weights are trained to produce the binary code of the index of the Kohonen neuron that is 1. For example, if Kohonen neuron 7 is 1 (and the others are all 0), the Grossberg layer will be trained to output 00 ... 000111 the binary code for 7). It is this shorter bit string that is transmitted.

At the receiving end, an identically trained counterpropagation network accepts the binary code and produces the inverse function, an approximate of the original subimage.

This method has been applied both to speech and images, yielding data compression ratios of 10:1 to 100:1. The quality has been acceptable, however, some distortion of the data at the receiving end is inevitable.

11.9 GRID COMPUTING

Grid computing or grid clusters are a technology closely related to cluster computing. The key differences between the grids and traditional clusters are that grids connect collection of computers which do not fully trust each other and hence operate more like a computing utility than like a single computer. In addition to this, grids typically support **more heterogeneous collections** than are commonly supported in clusters.

Grids use the resources of many separate computers connected by a network (usually the Internet) to solve large-scale computation problems. Grids provide us the ability to perform computations on large data sets, by breaking them down into many smaller ones or provide the ability to perform many more computations at once than would be possible on a single computer, by modeling a **parallel division of labour between processes**. Today, resource allocation in a grid is done in accordance with **SLAs (Service Level Agreements)**.

Grids serve to manage the allocation of jobs to computers which will perform the work independently of the rest of the grid cluster. **Resources such as storage may be shared by all the nodes but intermediate results of one job do not affect other jobs in progress on other nodes of the grid.**

The **main features** of Grid computing are as follows :

1. It offers a model for solving massive computational problems by making use of the unused resources (CPU cycles, disk storage etc) of large numbers of disparate computers, often desktop computers, treated as a virtual cluster embedded in the distributed telecommunications infrastructure.
2. Grids offer a way to solve challenging problems like **financial modeling**, earth -quakes simulation and climate/weather modeling.
3. Grids offer a way of using the IT resources optimally inside an organization.
4. It has the design goal of solving problems too big for any single super computer while retaining the flexibility to work on multiple smaller problems. Thus, grid computing provides a **multi-user environment**.
5. Grid computing involves sharing **heterogeneous resources** (based on different platforms, hardware/software architectures and computer languages), located in different places belonging to different administrative domains over a network using open standards. **Please note here that it involves virtualizing computing resources.**

Also note that **grid computing and cluster computing are different**. A **cluster** is a single set of nodes sitting in one location while a **Grid** is composed of many clusters and other kinds of resources like networks and storage facilities.

Functional Classification of Grids

1. **Computational Grids** : It includes CPU scavenging grids which focuses primarily on computationally intensive operations.

2. **Data Grids** : It does controlled sharing and management of large amounts of distributed data.
3. **Service/Equipment Grids** : They have a primary piece of equipment e.g., a telescope and where the surrounding grid is used to control the equipment remotely and to analyse the data produced.

11.10 COMPARISON OF CLUSTER COMPUTING AND GRID COMPUTING—TABULAR FORM

We tabulate the differences between a cluster and a Grid which is very important.

Cluster	Grids
<ol style="list-style-type: none">1. They are closely coupled, homogeneous—operates as a single computing resource.2. Cluster computers all have the same hardware and OS.3. The machines in a cluster are dedicated to work as a single unit.4. They often have high performance networking interconnects.5. They use specialized OS, designed to appear as a single computing resource.	<ol style="list-style-type: none">1. They are heterogeneous—operates as a generalized computing resource.2. Grid computers can run on different operating systems (OS) and have different hardware.3. A grid can make use of spare computing power on a desktop computer.4. Grids can use high-performance or standard interconnects.5. Here, the individual systems are not specialized but work based on standard machines and OS.

11.11 ADVANTAGES AND DISADVANTAGES OF GRIDS

Advantages :

1. It gives substantial computing power for simulations, complex computations and analysis.
2. The existing resources may be better exploited.
3. It may offer scalability via incremental increase of resources.

Disadvantages :

1. Indeterminate quality of service (QOS).
2. Rapidly changing infrastructure.
3. Grid technologies are VARIED, some are IMMATURE. Some require additional specialized high-capacity communication links.

SUMMARY

In this chapter, we discussed the basic principles of multithreading, latency hiding techniques, issues related to multithreading, its architecture. After this we took a short tour of cluster computing, Neuro computing and Grid computing.

MULTIPLE CHOICE QUESTIONS (MCQS) WITH ANSWERS

1. The basic principles behind multithreading is to hide :

(a) Memory cycles	(b) Memory storages
(c) Memory latency	(d) None of the above.

2. Small-scale multiprocessor systems use :

 - (a) Snoopy protocol
 - (b) Directory protocol
 - (c) TCP
 - (d) UDP

3. The multithreaded processor is also known as :

 - (a) Multiple context processor
 - (b) Many process processor
 - (c) More process processor
 - (d) None of the above.

4. Rate of requests (p) is given by :

 - (a) $p = R$
 - (b) $p = R \times R$
 - (c) $p = \sqrt{R}$
 - (d) $p = \frac{1}{R}$

5. In saturation state, the efficiency of the processor remains :

 - (a) Variable
 - (b) Constant
 - (c) Random
 - (d) None of the above.

6. An example of decoupled architectures can be :

 - (a) * T
 - (b) USC (Mc Gill MGDA)
 - (c) EM-4
 - (d) None of the above.

7. HPC stands for :

 - (a) High performance cluster
 - (b) High performance computer
 - (c) High probability cluster
 - (d) None of the above.

8. Webservers, mailservers are examples of :

 - (a) Hybrid clusters
 - (b) Load balancing clusters
 - (c) HPC
 - (d) None of the above.

9. The largest problem in clusters is :

 - (a) Software skew
 - (b) Software slowdown
 - (c) Software crisis
 - (d) None of the above.

10. Many inputs and many outputs trained are involved in :

 - (a) Classification
 - (b) Transformation
 - (c) Minimization
 - (d) None of the above.

11. Which of the following is a major European project in field of neural network :

 - (a) Pygmalion
 - (b) ARPANET
 - (c) ARCNET
 - (d) PICONET

12. A software package that is dedicated to the development of neural network with complex structure is :

 - (a) Brainteaser
 - (b) Brainstorming
 - (c) Braintracer
 - (d) None of the above.

13. A programming environment used for developing modular neural networks is:

 - (a) ARPANET
 - (b) STARNET
 - (c) PICONET
 - (d) METANET

14. Neuroshell is a :
- (a) Menu driven software (b) Shell programming construct
(c) Programming language (d) None of the above.
15. Clusters are homogeneous whereas Grids are :
- (a) heterogeneous (b) homogeneous
(c) distributed (d) None of the above.

:: ANSWERS ::

- | | | | | |
|---------|---------|---------|---------|---------|
| 1. (c) | 2. (a) | 3. (a) | 4. (d) | 5. (b) |
| 6. (b) | 7. (a) | 8. (b) | 9. (a) | 10. (b) |
| 11. (a) | 12. (c) | 13. (d) | 14. (a) | 15. (a) |

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

1. What are the four context-switching policies for multi-threaded architectures ?

Ans. There are four switching policies :

1. **Switch on cache miss** : This policy corresponds to the case where a context is preempted when it causes a cache miss. In this case, R is taken to be the average interval between misses (in cycles) and L, the time required to satisfy the miss. Hence, the processor switches contexts only when it is certain that the current one will be delayed for a significant number of cycles.
2. **Switch on every load** : This policy allows switching on every load independent of whether it will cause a miss or not. In this case, R represents the average interval between loads. A general multithreading model assumes that a context is blocked for L cycles after every switch but in case of a switch-on-load processor, this happens only if the load causes a cache miss.
3. **Switch on every instruction** : This policy allows switching on every instruction, independent of whether it is a load or not. Please note that it interleaves the instructions from different threads on a cycle-by-cycle basis. Successive instructions become independent which will benefit pipelined execution.
4. **Switch on block of instructions** : Block of instructions from different threads are interleaved. This will improve the cache-hit ratio due to locality. It will also benefit single-context performance.

2. What are multithreaded processors ? Explain its operation ?

Ans. A multithreaded processor is a processor designed to handle context switching. The state of a thread is called its **context**. Each context is represented by various registers in form of the register file, PC, PSW and stack. Thus, the multithreaded processor provides the hardware storage space for each context. The figure below shows a multithreaded processor model.

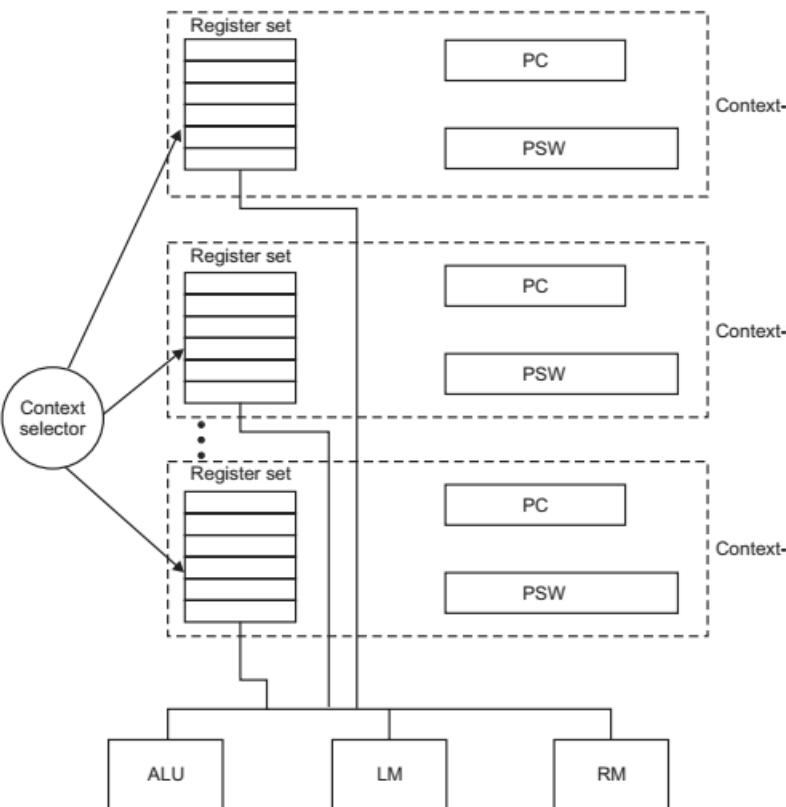


Fig. Multithreaded Processor Model

Legends :

- PC = Program counter
- PSW = Processor status word
- LM = Local memory
- RM = Remote memory

Operation of Multithreaded Processor

A single threaded processor's operation is as follows : It starts executing a single sequential thread. While executing if that thread requires some instruction or data from remote memory, then the processor waits for some time. This time comprises of the communication network delay, processor memory inter connection network delay, memory response time. Thus, the processor becomes idle for this much time. This slows down the efficiency of a single threaded processors.

A multithreaded processor's operation is as follows : A multithreaded processor is always kept busy in doing computation by switching between threads. When the current thread is suspended because of remote memory reference then the processor switches to a new thread and starts its execution. Now, the processor is no more idle. But, still few cycles are lost when processor does thread switch-

ing. This time is called as the context switching overhead. To achieve high efficiency, context switching overhead should be minimum. Please note that as compared to a single threaded processor, the efficiency of multithreaded processor is high as this processor is always busy.

During multithreading, the context goes through the following states :

Ready —> Leaving —> Blocked —> Running.

When the context is running, then the processor is busy. When the processor switches from context-1 to context-2, then context-1 is said to be in 'leaving' state. Basically, the processor switches from context-1 because context-1 may need remote memory reference.

Hence, context-1 goes from 'leaving' state to 'blocked' state. Again, when the remote memory reference is available, the context-1 goes from blocking state to ready state.

Finally, when the processor switches to the ready context and starts executing it, then the context-1 goes into 'running' state.

3. What is a thread ? What are its different types ?

Ans. A thread is a basic unit of execution. Basically, it is a sequence of instructions which can execute independently. The state of the thread is called its context. It is represented in terms of the number of processor registers, program counter, stack, and processor status word.

There are two types of threads, namely :

- (a) Kernel threads
- (b) User threads.

Kernel threads are managed by kernel.

User threads are managed in the user space.

4. What are the differences between a process and a thread ?

Ans.

Process	Thread
<ul style="list-style-type: none">1. Process can't share the same memory area (address space).2. It takes more time to create a process.3. It takes more time to terminate.4. Execution is very slow.5. It takes more time to switch between two processes.6. Process is loosely coupled.7. It requires more resources to execute.	<ul style="list-style-type: none">1. Threads can share memory files.2. It takes less time to create a thread.3. It takes less time to terminate.4. Execution is very fast.5. It takes less time to switch between two threads.6. Threads are tightly coupled.7. Requires fewer resources, so it is also called as a light weight process.

5. Why is thread a light weight process ?

Ans. In Unix, which is a heavy weight process, a complete copy of the process with its own memory allocation, variables, stack etc is created even though the execution starts from the forked position only. Often a complete copy of a process is not required. So, thread provides us a means of not creating a complete copy again and again, instead it allows sharing of the same memory space and global variables. Hence, it is a light weight process.

EXERCISE QUESTIONS

1. What is the basic concept of multithreading ? Explain the following performance measuring parameters :

- (a) Latency
- (b) Context switching overhead
- (c) Number of threads
- (d) Interval between switches ?

[Pune Univ., B.E. 2nd sem ; May. 2005 & Dec. 2006]

2. Explain different context switching policies adopted by multithreaded architectures.

[Pune Univ., B.E. 2nd sem ; May. 2005]

3. What is basic multithreading ? Explain multithreaded architectures and its computational model for parallel processing system.

[Pune Univ., B.E. 2nd sem ; Dec. 2004 & Dec. 2005]

4. Write short notes on :

- (a) Latency hiding techniques.
- (b) Features of parallel processing languages.

[Pune Univ., B.E. 2nd sem ; May. 2004 & Dec. 2005]

5. Write short notes on :

Principles of multithreading.

[Pune Univ., B.E. 2nd sem ; May. 2001]



CHAPTER

12 OPERATING SYSTEM ISSUES

12.0 INTRODUCTION

Operating system requirements of a multiprocessor and multiprogramming are conceptually very less different. But some additional complexity is there when multiple processor must work simultaneously.

The functional capabilities required in an OS for multiprogrammed computer includes:

- (a) Resource allocation and management schemes.
- (b) Memory and data set protection.
- (c) Prevention of system deadlocks.
- (d) Abnormal process termination or exception handling.

It also needs techniques for efficient utilization of resources and hence must provide I/O and processor load balancing schemes.

One of the main reasons for using a multiprocessor system is to provide some effective reliability and graceful degradation in the event of failure. Thus, O.S. must also be capable of providing system reconfiguration schemes to support graceful degradation.

Such extra capabilities and multiprocessor execution environment places heavy load on O.S. Thus, it is important for multiprocessor computer to design the O.S. efficiently.

12.1 CLASSIFICATION OF MULTIPROCESSOR O.S.

User programs have variety of requirements. Accordingly, there are variety of architectures and O.S. In general, the O.S. for advanced architectures are put under 4 categories as shown below in Fig. 12.1.

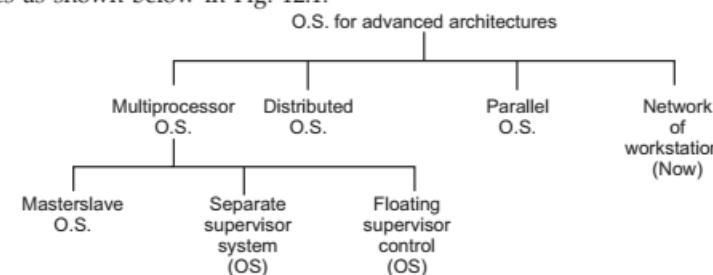


Fig. 12.1. Types multiprocessor O.S.

But, we are interested in multiprocessor O.S. mainly. There are three organizations that have been utilized in the degree of O.S. for multiprocessors, namely :

- I. Master slave configuration.

- II. Separate supervisor for each processor.
 - III. Floating supervisor control.
- We shall explain these one by one now.

I. Master Slave Configuration

This is the mostly used O.S. mode. It's main features are :

- 1. The supervisor is always run on the same processor and hence it is simplest to implement.
- 2. It may be designed by making some simple extensions to a uniprocessor operating system that includes full multiprogramming capabilities.
- 3. It is normally quite inefficient in its control and utilization of total system resources.

In **Master-slave mode**, one processor called **Master** maintains the status of all processors in the system and allocates work to all the slave processors. It means operating system is executed by one peripheral processor M_0 , all the other processors are treated as slaves to M_0 .

Example cyber 170.

Working with supervisor routine, it is always executed in the same processor thus slave request via, supervisor call instruction or trap for execution service that must be sent to master which acknowledges the requests and performs the appropriate service. **Some of the other characteristics of master-slave operating systems are :**

Table conflict and lockout problems for system control tables are simplified by forcing a single processor run the executive, but this operating system mode causes the system to be very susceptible to catastrophic failures which require operation intervention to restart the master processor when irrecoverable errors occurs.

The master-slave mode is most effective for special applications where the work load is well defined or for asymmetric systems where slaves have less capability than the master processor.

Master Slave Operating System :

- 1. The executive routine is always executed in the same processor. If the slave needs service that must be provided by the supervisor, then it must request that service and wait until the current program on the master processor is interrupted and the supervisor is dispatched. The supervisors and the routines that it uses do not have to be reentrant since there is only the one processor using them.
- 2. Having the single processor executing the supervisor simplifies the table conflict and lock out problem for control, tables. The overall software is comparatively inflexible. This type of system requires comparatively simple software and hardware.
- 3. The entire system is subject to catastrophic failures that require operator intervention to restart when the processor designated as the master has a failure or irrecoverable error.
- 4. Idle time on the slave system can built up and become quite appreciable if the master cannot execute the dispatching routines fast enough to keep the slave (s) busy.

5. This type of operating system is most effective for special applications where the work load is well defined or for asymmetrical system in which the slaves have less capability than the master processor.

II. Separate Supervisor System

Operating system characteristics are very different from master-slave system when there is separate supervisor system (kernel) running in each processor. This is similar to the concept of computer network where each process contains separate kernel.

Each processor services its own needs. However, as there is some interaction between the processors it is necessary for some of the supervisory code to be replicated to provide separate copies for each processor.

Each processor has its own set of input/output devices and files and any configuration of input/output usually requires manual intervention and possibly manual switching. And each supervisor has its own set of private tables. Some tables are common and shared by the whole system. This creates table access problems. The method used in accessing the shared resources depends on the degree of coupling among the processors. The separate supervisor operating system is not as sensitive to a catastrophic failure as a master-slave system.

Unfortunately the replication of the kernel in the processors would demand a lot of memory which may be underutilized, especially when compared with the utilization of shared data structure.

Thus, static form of caching could be used to buffer frequently used portions of operating system code, while infrequently used code could be centralized in shared memory, but determination of which portions of operating system are frequently executed is relatively difficult to make and is dependent of application workload.

Separate supervisor in each processor :

1. Each processor services its own needs. In effect, each processor (supervisor) has its own set of input/output equipment, files etc.
2. It is necessary for some of the supervisory code to be reentrant or replicated to provide separate copies for each processor.
3. Each processor (actually each supervisor) has its own set of private tables, although some tables must be common to the entire system and that creates table access control problems.
4. The separate supervisor operating system is as sensitive as is the master-slave system, however the restart of an individual processor that has failed will probably be quite difficult.
5. Because of the point immediately above, the reconfiguration of input/output usually requires manual intervention and possible manual switching.

III. Floating supervisor control

In this mode the supervisor routines float from one processor to another, although several of the processors may be executing supervisory service routines simultaneously.

Service request conflicts are resolved by priorities that are either set statically or dynamically but in this system, table access conflicts and table lock-out delays cannot be avoided.

This type of system can attain better load balancing over resources and it has advantages of providing graceful degradation and better availability of reduced capacity system. As well as, it provides true redundancy and makes the most efficient use of available resources.

Example : MVS operating system in IBM 3081 on C.MMP. and VM operating system in Hydra on C.MMP.

Many operating systems are not pure example of three classes discussed above.

Floating supervisor operating system :

1. The "master" floats from one processor to another, although several of the processors may be executing supervisor service routines at the same time.
2. This type of system can attain better load balancing over all types of resources.
3. Conflicts in service requests are resolved by priorities that can be set statistically or under dynamic control.
4. Most of the supervisory code must be reentrant since several processors can execute the same service routine at the same time.
5. Table access conflicts and table lock-out delays can occur, but there is no way to avoid this with multiple supervisors, **the important point is that they must be controlled in such a way that system integrity is protected.**

12.2 SOFTWARE REQUIREMENT OF MULTIPROCESSOR O.S.

The software for multiprocessor system differs from that of a uniprocessor systems on the following accounts :

1. **Architectural attributes**, which are unique to multiprocessors.
2. A new programming style, which is peculiar to parallel applications.

A multiprogrammed uniprocessor can simulate the multiple processor environments by creating multiple "**virtual processors**" for the users. **For example**, in a Unix O.S., each program can be thought of being executed on a separate virtual processor. However, the presence of real multiple processors demands extra amount of management software. This may be because of :

1. Non-homogeneity of processor in a multiple processing system. Because of this non-homogeneity, the software must treat the processors differently.
2. Asymmetric main memory *i.e.*, all PEs cannot access all the memory, complicates the O.S. software for resource management.
3. The PEs running concurrently may need to interact with each other.

The possibility of parallel processing in a program can be indicated by two ways
— (a) Explicitly (b) Implicitly.

For **explicit parallelism**, the user indicates the segments of the program, which can be run parallel.

Implicit parallelism is detected by the compiler. The approach is more suitable for **data-flow computers**.

Data dependency is the main factor for the interprocess detection of parallelism with sufficient granularity. The recogniser often puts on overhead, which may not be cost effective for analysing certain classes of programs in determining their parallel possibility.

12.3 OPERATING SYSTEMS REQUIREMENTS

There are many considerations to be taken care of when multiprocessor O.S. are designed :

1. **Defining the role of O.S.** : The O.S. can provide complete operating interface as in uniprocessor systems or allow the processors to manage some of the resources itself.
For example, a multithreading process may want to schedule its own data transfers and therefore manage the network card all by itself. For example, parallel applications often have this requirement.
2. **Modular design** : A modular design plays a major role in parallel O.S. also. As we know that independent modules force the interfaces to become standard. Thus, activities such as replacing a device driver can take place without affecting the overall functionality of other modules.
3. **Scalability** : It is the most important goal of parallel O.S. It makes economic decisions of purchase of hardware and software independent of the O.S. itself. More importantly, it requires scalable interfaces to the system modules, making O.S. programming independent.
4. **Portability** : Writing software for new system involves heavy investments. The achievable speed-ups often justify the cost of rewriting software with the same functionality. Instead, the O.S. designs should make the personality of the system independent to the base design. Each existing O.S. have an interface, which can be abstracted and implemented as independent personality. Also, newer versions of same O.S. should support older binaries. On the other hand, writing parallel software on parallel systems can have different standards and does not affect the O.S. design.
5. **Kernel space and user space** : In multi-user, multitasking environment, a processor should not be able to modify memory of another process or the kernel memory, unless explicitly shared. This feature ensures robustness of the systems. This is usually provided by using hardware support to switch the context in controlled manner when a process requests O.S. services. It introduces considerable overhead. In an O.S., for parallel processing, the application may want minimal overheads and since usually single parallel tasks run on all processors, the protection is not necessary. O.S. design should, therefore, provide incremental addition of required features such as protection memory space .

12.4 DISTRIBUTED SCHEDULER-PVM (A CASE STUDY)

Parallel Virtual Machine (PVM)–Case Study

PVM is self contained, public-domain software system that was originally designated to enable a network of heterogeneous unix computers to be used as a large scale, message passing parallel computer.

PVM has been implemented for non-unix platforms such as Windows NT and Windows 95. The programming languages supported include C, Fortran and Java.

With PVM user can construct a **virtual machine**. The set of computers used on a problem first must be defined prior to running the programs. This forms virtual parallel machine. The user can then dynamically create and manage a number of processes to run on this virtual machine. PVM provides library routines to support interprocess communication and other functions.

Virtual machine construction :

PVM system consist of two parts :

1. PVM daemon (called pvmd).

It resides on every computer of virtual machine.

2. User-callable library (called libpvm3.a)

This is linked to user application for process management, msg.passing and virtual machine management.

In PVM calls, a node is called as "host" and process is also called a "task".

The most convenient way of creating virtual machine is by creating a list of the names of the computers available in a host file. The host file is then read by PVM with command.

`pvm host_file`

Successful execution of this command will start pvm daemon on the invoking host and every host listed in host file. Invoking host, displays a prompt to indicate that, the host is in PVM console mode.

`pvm >`

The user types of number of commands to manage the virtual machine, to invoke a PVM application job and to monitor the execution of the job with this prompt which is similar to shell.

An alternative way is to start with one machine and add or delete others manually be using PVM control commands. This is known as **Dynamic Configuration**.

```
pum_addhost (hosts, nhost, infos) ;
pum_delhosts (hosts, nhost, infos) ;
```

where,

hosts : Names of host

nhost : Number of hosts to be added or deleted.

infos : integer array of length nhost; holds a status code returned for each host.

These two functions, number the hosts successfully added or deleted. When completely successful, information is equal to nhost.

The routing of message between computers is done by PVM daemon process installed by PVM on the computers that form the virtual machine. Each PVM daemon requires sufficient information to be able to select the routing path. This is shown below in Fig. 12.2.

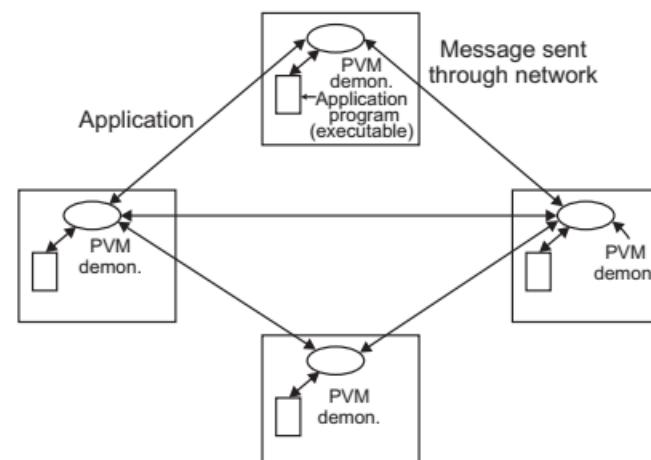


Fig. 12.2. Message Passing Between Workstations Using PVM.

Here, every process is related with one processor.

If number of processes is greater than the number of processors. PVM allows any number of process to be created without any relationship to the number of processors and it will allocate processors automatically.

Example : If there were eight processes and three processors, processes might be allocated to each processor as shown below :

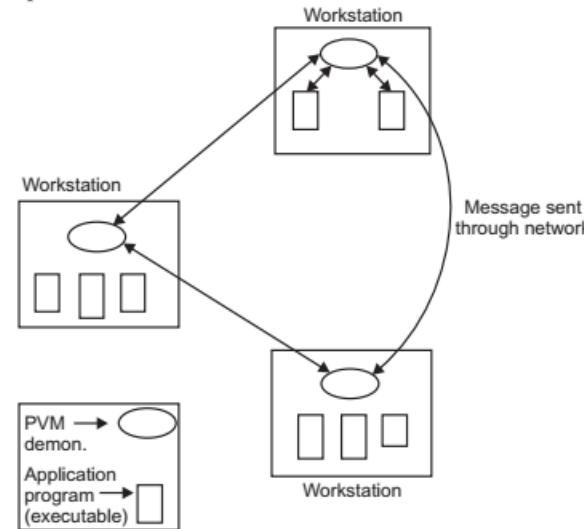


Fig. 12.3. Multiple Processes Allocated to Each Processor (Workstation)

Implementation

PVM programs are usually organized in a master-slave arrangement. The first pvmd started by hand is called the **master pumdd**. Other pvmd daemons started by master called **slaves**. The set of master and the slave daemons forms the virtual machine. Any virtual machine has exactly one master at a time.

Only the master can add or delete a slave from the virtual machine. However, requests to add or delete a slave come from outside of the master domain, that are transferred to master daemon which then starts the slave on specific host. PVM allows the master to start up a slave via rsh, rexec () etc.

Data structure called **host table** resides on every host. The host table has an entry called **host descriptor** which holds configuration information, as well as packet queues and message buffers for communication. The host table of master host is updated to include new entry for newly added slave. Information of updated host table is then broadcast to entire virtual machine, including newly added machine, thus all hosts of the virtual machine are synchronized and consistent.

Processes Creation and Execution

PVM enables both static and dynamic parallelism *i.e.*, process can be started and executed statically or dynamically.

To start executing one or more identical processes the PVM routine `pvm_spawn()` is used. To start a static parallel program user may execute,
 "spawn-count 2 hello" at PVM console.

This will create a parallel application of 2 processes to run on virtual machine each process executes same code hello.

To start dynamic process, the dynamic task creation function `pvm_spawn()` is used.

For mapping of processes and hosts, PVM provides an allocation algorithm. Multiple tasks are mapped to each host. PVM allows the user to explicitly specify a specific host or architecture type for each task.

One of the first steps a process must take when it executes is to be "enrolled" in PVM that can be done with the PVM system call `pvm_mytid()`, which returns task ID. Basically this routine is used only when task-ID is required otherwise any PVM routine can enroll the process if the process has not already been enrolled.

Some of the process management functions are listed below :

Table 12.1

PVM Function Call	Meaning
<code>tid = pvm_mytid();</code>	Get the task ID of the calling task.
<code>tid = pvm_parent();</code>	Get the task ID of the parent task.
<code>info = pvm_exit();</code>	The called task exits PVM. The task continues running as a unix process..
<code>mumt = pvm_spawn(...);</code>	Spawn a PVM task.
<code>info = pvm_kill(tid);</code>	Terminate a PVM task.
<code>tstat = pvm_pstat(tid);</code>	Get the status of a PVM task.
<code>info = pvm_task(...);</code>	Get the information of all tasks running on the virtual machine.
<code>mstat = pm_mstat(host);</code>	Get the status of a host.
<code>info = pvm_config(...);</code>	Get the configuration information of the entire virtual machine.

Grouping

The PVM uses a separate library called `libpvin3.a` to support collective operations. Separate daemon called the **group server** handles the grouping functions.

PVM supports **dynamic grouping** i.e., any task can join or leave a group at any time. There can be multiple groups, and a task can belong to different groups at the same time. A task can join or leave a group at any time without notifying other members of the group.

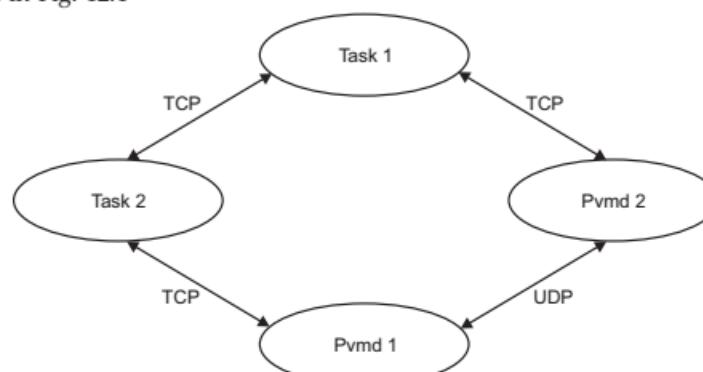
Dynamic grouping increases the non-deterministic behaviour of a program. A broadcast operation can have different results if a task is joining or leaving a group. The task may or may not get the broadcast message. An operation may deadlock if member tasks are leaving a group. Some of the grouping functions are listed below :

Table 10.2 : PVM Group Functions

PVM Function Call	Meaning
inum = pvm_joingroup ("World") ;	The calling task joins group World and is assigned an instance number inum. the instance number is like rank in MPI.
info = pvm_ivgroup ("World") ;	The calling task leaves group World.
tid = pvm_gettid ("World", inum) ;	Get task ID from instance number.
inum = pvm_getinst ("World", tid) ;	Get instance number from task ID.
gsize = pvm_gsize ("World") ;	Get Group size
info = pvm_barrier ("World", 10) ;	The calling task blocks (waits) until 10 members of World have called pvm_barrier.
info = pvm_bcast ("World", tag) ;	Broadcast a message identified by tag to all members of World (excluding self)
info = pvm_reduce (...) ;	Similar to reduction in MPI.

Programs communicate by message passing using PVM library routines. A PVM program running on a virtual machine involves daemons and tasks. Different kind of communication are possible like communication between two tasks, two daemons and a daemon and a task.

For communication, PVM chooses protocols i.e., TCP and UDP. These protocols are chosen as they are standard and used widely. The communication protocols are used as shown in Fig. 12.4

**Fig. 12.4. Protocols used**

Communication between pvmd and tasks are through TCP as it is reliable.

Communication between pvmd and pvmd uses UDP protocol because of following reasons :

- Scalability i.e., single UDP socket can communicate with any number of remote UDP sockets.
- UDP is connection protocol so it is inexpensive.
- It is easier to set time-outs in UDP to detect host pvmd and network failures.

Communication Function :

In PVM, to send a message three steps are :

1. Send buffer must be initialized by a call to pvm_initsend or pvm_mkbuf.
2. The msg must be packed into this buffer using any number and combinations of pvm_pk* routines.
3. The complete message is sent to another process by calling pvm_send routine or multicast with pvm_mast routine.

Some pvm communication functions are given below :

- int bufid = pvm_initsend (int encoding)
creates a new send buffer and makes it the current active buffer.
- int info = pvm_pkint (int *p, int nietm, int stride)
actual packing of data.
- int info = pvm_send (int tid, int tag)
send the message in active send buffer to the task denoted by tid and tags the message with tag.
- int info = pvm_mcast (int *tids, int ntasks, int tag)
to broadcast message to several tasks, where tids_list of destination task and n tasks is number of tasks.

A message is received by calling either a blocking (pvm_recv) or non-blocking (pvm_nrecv) receive routine and then unpacking each of the packed items from receiver buffer.

- int lpufl id = pvm_recv (int tid, int tag)
for blocking receives.
- int bufid = pvm_nrecv (int tid, int tag)
for unblocking receives.
- where tid and tag indicate any task and any tag respectively.
- int bufid = pvm_probe (int tid, int tag)
examine whether message is arrived or not.
- int bufid = pvm_bufinfo (int bufid, int *bytes, int *tag, int *tid)
refer to buffer information.
- int info = pvm_upkint (int *p, int nitem, int stride)
to unpack the packed item.

12.5 PTHREADS IN SHARED MEMORY SYSTEM (A CASE STUDY)

In shared memory multiprocessor architectures, threads can be used to implement parallelism. Historically, hardware vendors have implemented their own proprietary versions of threads, making portability a concern for software developers. For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard, implementations that remain to this standard are referred to as POSIX threads, or Pthreads.

Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system.

Pthreads

Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult

for programmers to develop portable threaded applications. In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required. For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995). Implementations, which adhere to this standard, are referred to as POSIX threads , or Pthreads. Most hardware vendors now offer Pthreads in addition to their proprietary API's. Pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library – though this library may be part of another library, such as libc.

The primary motivation for using Pthreads is to realize potential program performance gains. When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes. All threads within a process share the same address space. The primary motivation for considering the use of Pthreads is to achieve optimum performance.

Designing Threaded Programs

In order for a program to take advantage of Pthreads, it must be able to be organized into independent tasks, which can execute concurrently.

For example, if routine 1 and routine 2 can be interchanged or overlapped in real time, they are candidates for threading.

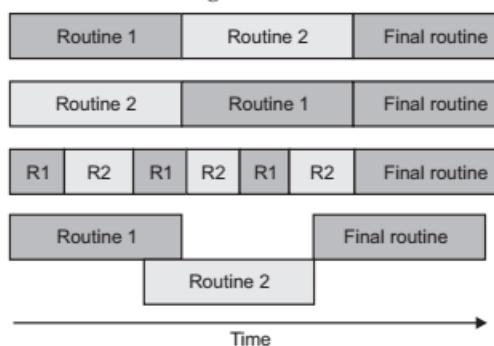


Fig. 12.5. Threading

Tasks that may be suitable for threading include tasks that :

- Block for potentially long waits.
- Use many CPU cycles.
- Must respond to asynchronous events.
- Are of lesser or greater importance than other tasks.
- Are able to be performed in parallel with other tasks.

The Pthreads API

The subroutines, which comprise the Pthreads API, can be informally grouped into three major classes :

Thread management : The first class of functions work directly on threads – creating, detaching, joining etc. They include functions to set/query thread attributes (joinable, scheduling etc.)

Mutexes : The second class of functions deals with synchronization, called a “mutex”, which is an abbreviation for “**mutual exclusion**”. Mutex functions provide for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

Condition variables : The third class of functions address communications between threads that share a mutex. They are based upon programmer specified conditions. This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

Thread Management

Naming conventions : All identifiers in the threads library begin with pthread-

Routine Prefix	Functional Group
Pthread_	Threads themselves and miscellaneous subroutines.
Pthread_attr_	Thread attributes objects.
Pthread_mutex_	Mutexes.
Pthread_mutexattr_	Mutex attributes objects.
Pthread_cond_	Condition variables.
Pthread_condattr_	Condition attributes objects.
Pthread_key_	Thread-specific data keys.

The concept of **opaque objects** supports the design of the API. The basic calls work to create or modify opaque objects – the opaque objects can be modified by calls to attributes functions, which deal with opaque attributes. The Pthreads API contains over 60 subroutines. This tutorial will focus on a subset of these – specifically, those which are most likely to be immediately useful to the beginning Pthreads programmer. The pthread.h header file must be included in each source file using the Pthreads library. For some implementations, such as IBM’s AIX, it may be needed to first include file. The current POSIX standard is defined only for the C language. Fortran programmers can use wrappers around C function calls. Also, the IBM Fortran compiler provides’ a Pthreads API.

Creating Threads

Initially, our main () program comprises a single, default thread. The programmer must explicitly create all other threads. Pthread_create creates a new thread and makes it executable. Typically, threads are first created from within main () inside a single process. Once created, threads are peers, and may create other threads.

Pthreads create arguments :

- * **thread** : An opaque, unique identifier for the new thread returned by the subroutine.
- * **attr** : An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
- * **start-routine** : The C routine that the thread will execute once it is created.
- * **arg** : A single argument that may be passed to start routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

- By default, a thread is created with certain attributes. The programmer via the thread attribute object can change some of these attributes.
- Pthread_attr_init and pthread_attr_destroy are used to initialize/destroy the thread attribute object.
- The maximum number of threads that may be created by a process is implementation dependent.

Terminating Threads :

- There are several ways in which a pthread may be terminated :
 - * The thread returns from its starting routine (the main routine for the initial thread).
 - * The thread makes a call to the pthread_exit subroutine (given below).
 - * The thread is canceled by another thread via the pthread_cancel routine (not covered here).
 - * The entire process is terminated due to a call to either the exec or exit subroutines.
- Pthread_exit is used to explicitly exit a thread. Typically, the pthread_exit () routine is called after a thread has completed its work and is no longer required to exist.
- If main () finishes before the threads it has created, and exits with pthread_exit(), the other threads will continue to execute. Otherwise, they will be automatically terminated when main () finishes.
- The programmer may optionally specify a termination status, which is stored as a void pointer for any thread that may join the calling thread.
- Cleanup : the pthread_exit () routine does not close files; any files opened inside the thread will remain open after the thread is terminated.

Example : Pthread Creation and Termination

- This simple example code creates 15 threads with the pthread_create () routine. Each thread prints a "Hello World!" message, and then terminates with a call to pthread_exit ().

Example Code – Pthread Creation and Termination

```
# include<pthread.h>
# include<stdio.h>
#define NUM_THREADS 15
Void *Print Hello (void *threaded)
{
    printf ("\n%d: Hello World! \n", threaded) ;
    Pthread_exit (NULL);
}
int main (int argc, char *argv [])
{
    Pthread_t threads [NUM_THREADS]
    int rc, t ;
    for (t = 0; t<NUM_THREADS; t++) {
        printf ("Creating thread %d\n", t);
        rc = pthread_create (&threads (t), NULL, Print Hello, (void
*)t) ;
```

```
if (rc)
{
    printf ("ERROR; return code from pthread_create ( ) is %d\n",
rc) ;
    exit(-1) ;
}
}
Pthread exit (NULL) ;
```

Mutex Variable

- Mutex is an abbreviation for “**mutual exclusion**”. Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.
- Mutexes have two basic operations, **lock** and **unlock**. If a mutex is unlocked and a thread calls lock, the mutex locks and the thread continues. If, however, the mutex is locked, the thread blocks until the thread ‘holding’ the lock calls unlock.

There are 5 basic functions dealing with mutexes.

1. Note that you pass a pointer to the mutex, and that to use the default attributes just pass NULL for the second parameter.
int pthread_mutex_init (pthread_mutex_t *mut, const
pthread_mutexattr_t
*attr) ;
2. Locks the mutex :
int pthread_mutex_lock (pthread_mutex_t *mut);
3. Locks the mutex :
int pthread_mutex_unlock (pthread_mutex_t *mut) ;
4. Either acquires the lock if it is available, or returns EBUSY.
int Pthread mutex try lock (pthread_mutex_t *mut) ;
5. Deallocates any memory or other resources associated with the mutex.
int Pthread mutex try lock (pthread_mutex_t *mut) ;

Condition Variables

- Conditions variables provide yet another way for threads to synchronize. While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
- A Condition variable is always used in conjunction with a mutex lock.
- A representative sequence for using condition variables is shown below.

Main thread

- Declare and initialize global data/variables which require synchronization (such as “count”)

SUMMARY

In this chapter, we have studied about the operating systems that are needed for multiprocessor. We have classified various types of O.Ss. We have also taken a CASE study on PVM and P threads that help in parallel programming.

MULTIPLE CHOICE QUESTIONS (MCQS) WITH ANSWERS

1. Which type of parallelism is most suited for data flow computers (DFCs)
(a) Explicit (b) Implicit
(c) Both (a) and (b) (d) None of the above
 2. The host file is read by PVM using command :
(a) pvm (b) pvm > host-file
(c) pvm host-file (d) None of the above
 3. In master - slave mode, the status of all PEs is maintained by :
(a) Master PE. (b) Slave PE.
(c) Both (a) and (b). (d) None of the above
 4. In PVM, a data structure that resides on every host is known as :
(a) Host data (b) Host name.
(c) Host table. (d) None of the above
 5. Primary means of implementing thread synchronization is :
(a) Mutex (b) Semaphore.
(c) Monitors. (d) None of the above

ANSWERS

1. (b) 2. (c) 3. (a) 4. (c) 5. (a)

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

- ### 1 What do you mean by :

- (a) Distributed O.S.

- (b) Parallel O.S.**

Ans. (a) Distributed O.S.

The processors have independent memory. They are managed in such a way that the users get single monolithic view of complete system. However, the resources like disks are central.

- (b) Parallel O.S.

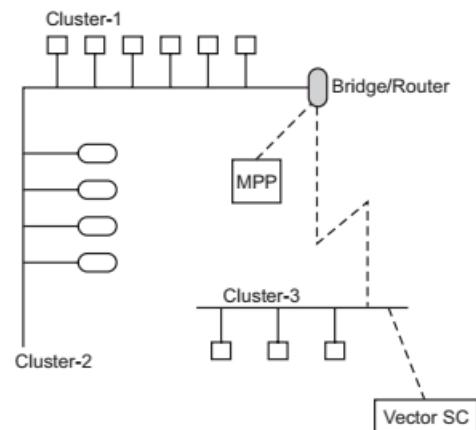
Here, the main emphasis is to support parallel applications on the distributed memory architectures. The OS itself provides very few services and allows the hardware services (such as data transfer) to be managed by the applications itself.

- ## 2. What is 'Network of workstation's' i.e., NOW?

Ans. A distributed system in which processors are usually heterogeneous and the OSes are independent of each other. However, they achieve global sharing of data by passing using an external network. The network itself spreads over large areas and can connect hundreds of systems.

- ### 3. What is the PVM architectural overview?

Ans.

**Fig.****4. Give some applications of PVM?**

- Ans.** (1) User programs are provided access to PVM through the use of calls to PVM library routines for functions such as process initiation, message transmission and reception.
 (2) PVM is a viable scientific computing platform that has been used for applications like molecular dynamic simulations, superconductivity studies, distributed fractal computations and matrix algorithms.
 (3) PVM has become the defacto standard for distributed computing.

EXERCISE QUESTIONS

1. Write short notes on :

- (a) Parallel virtual machines
- (b) Multiprocessor O.S.
- (c) Pthreads

[Pune Univ; B.E. 2nd sem; Dec., 2001, May 2002, Dec 2003, May–2006]

2. Explain operating system features for multiprocessor configuration .

[Pune univ ; BE 2nd sem; , May 2004]

3. How PThreads act as a programming interface for parallel processing?

[Pune Univ; B.E. 2nd sem; Dec. 2005]

4. What are the requirements of multiprocessor operating system? What are the various types of OS for multiprocessor? Describe major characteristics of each?

[Pune Univ ; B.E. 2nd sem; May 2003]

5. Compare the requirements of operating systems for uniprocessor and multiprocessor O.S. ?

[Pune Univ; B.E. 2nd Sem; Dec. 2001]



CHAPTER

13 OPENMP AND MPI

13.0 INTRODUCTION

OPENMP Stands for open specifications for multiprocessing. It is defined as an Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism. It comprises of three primary API components. These components are as follows –

- (a) Computer Directives.
- (b) Runtime Library Routines.
- (c) Environment variables.

Please understand that openMP is not meant for distributed memory parallel systems and is not necessarily implemented identically by all vendors. Also note that it is not guaranteed to make the most efficient use of shared memory.

OpenMP is a collaborative work between the interested parties from the hardware and software industry, government and academia.

13.1 GOALS OF OPENMP

There are some common goals of OPENMP as follows –

- (1) **Standardization:** It provides a standard among a variety of shared memory architectures.
- (2) **Portability:** It supports FORTRAN'S all dialects like 77, 90 and 95 as well as C and C++.
- (3) **Easy to use:** It provides the capability to incrementally parallelize a serial program. Please note that openMP provides the capability to implement both course grain and time grain parallelism.
- (4) **Multiple platforms:** It supports most of the UNIX platforms and WIN-NT platforms.
- (5) **Limited directives:** It supports simple and limited set of directives to program shared memory machines. By using just 3 to 4 directives (say) we can implement significant parallelism.

13.2. FORK – JOIN MODEL OF PARALLEL EXECUTION

OpenMP uses the fork–join model of parallel execution. This model is basically useful for large array – based applications. It is based upon the existence of multiple threads in the shared-memory model. It is an explicit programming model providing the programmer full control over parallelization.

Principle: “All openMP programs begin as a single process called as a master thread, which executes sequentially until the first parallel region construct is encountered.

OpenMP supports programs that will execute correctly both as **parallel programs** (using multiple threads, full openMP supports library) as well as **sequential programs** (using a simple openMP stub library).

The **master threads** then creates a team of parallel threads. Those statements of the program that are enclosed by the parallel region construct are then executed in parallel. When the team of threads completes it, they synchronize and terminate. So, now only the master thread is left with to execute the sequential program. The fork-join model is shown in Fig.

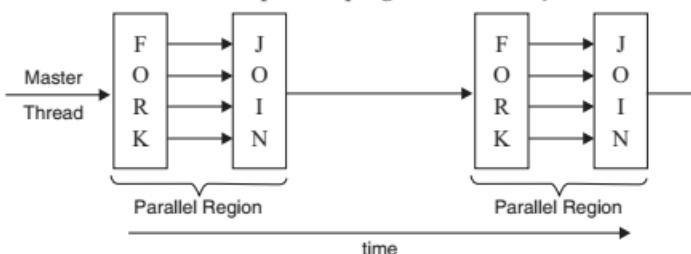


Fig.1 Fork-Join Model (openMP)

The main features of openMP parallelism are as follows-

- (1) Parallelism is specified with the use of **compiler directives**. These directives are embedded in C/C++ FORTRAN source code.
- (2) The three APIs of openMP provides for the **nesting** of parallel constructs and thus dynamically changing of threads is allowed here.
- (3) It does not specify anything about **parallel I/O**. What if the multiple threads try to read/write from the same file.
So, now the responsibility of the programmer increases.
- (4) It provides a **relaxed – consistency and temporary view** of thread memory. This means that **threads can cache their data** and are not required to maintain the exact consistency with real memory for whole of the time period. If the situation is critical i.e., all threads use and view a shared variable identically then the programmer must ensure that the variable is flushed (cleared) by all threads as needed.

General Code Structure (C/C++)

```
Its Syntax is
# include <omp.h>
main ()
{
    int V1, V2, V3;
    serial code
    :
    :
    Beginning of parallel section
    Fork a team of threads
    Specify variable scoping.
    # pragma omp parallel private (V1, V2) shared V3
    {
        Parallel section executed by all threads
        :
        :
        All threads joins master thread & disbands
    }
```

```

}
Resume serial code
:
:
}

```

Actually, a **compiler directive in C/C++ is called as a pragma**. It is a way to communicate information to the compiler. This information helps the compiler to produce correct and optimized program (object code). A pragma begins with a #.

For e.g.:

```
# pragma omp parallel default (shared) private (B, Pi)
```

These are some general rules for the directives format -

- (1) They are case sensitive.
- (2) They follow conventions of C/C++ standards.
- (3) Only one directive name may be specified per directive.
- (4) Each directive applies to at most one succeeding statement which may be a structure block.
- (5) Long directive lines can be continued on the succeeding lines by escaping the new line character with a backslash at the end of a directive line.

When to use OpenMP?

There are two reasons for this

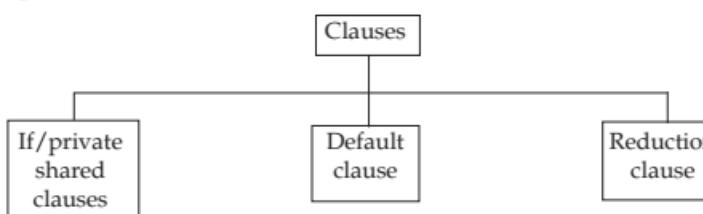
- (1) Parallelization by the compiler may not be the same as you wish like a loop is not parallelized, the granularity is not high.
- (2) when explicit parallelization comes into the consideration.

13.3 OPENMP CLAUSES

Many openMP directives support clauses. **For example,**

```
# pragma omp for private (a)
```

Here, private (a) is a clause to the 'for' directive.



Let us discuss these one by one

I. If clause : Its syntax is

```
If (scalar expression)
```

Execute in parallel only if the expression evaluates to true else execute serially.

II. Private clause: It tells the compiler to allocate a private copy of the variable for each thread executing the block of code, the pragma proceed.

For example,

```
# pragma omp parallel for private (i)
```

```

{
for (i = 0; i < n; i++)
{
    sort (a[i]);
}
// End of openMP parallel region.

```

III . Shared clause: Its *syntax* is

Shared (< variable - list>)

Here, all threads access the same address.

IV. Default clause: Its *syntax* is

default (none/shared)

'None' means no implicit defaults.

'Shared' means all variables are shared.

V. Reduction clause: Its *syntax* is

reduction (operation : variable)

Where '*variable*' is the name of the shared variable that will end up with the result of the reduction.

For example,

```

double avg ;
int i,n;
Sum = 0;
#pragma omp parallel for private (x) reduction (+ : sum)
for (i = 0; i < n; i++)
{
    sum = sum + i;
}
avg = sum/n;

```

13.4 PARALLEL REGION CONSTRUCT

A parallel region is a block of code that will be executed by multiple threads.

How it happens?

When a thread reaches a "parallel" directive, it creates a team of threads and becomes the master of the team. Please note that the master is the member of that team and has thread number 0 within that team. The code in the parallel region is multiplied so that all threads execute that code. There is a point after which only the master thread continues execution. This point is known as an implied barrier. It is at the end of the parallel section. Also note that if a thread terminates within a parallel region then all threads in the team will terminate. The work done until that point is undefined. The syntax of parallel region construct is

```
#pragma omp parallel [clause....] newline  
    if (scalar - expression)  
    private (var-list)  
    shared (var-list)  
    default (shared/none)  
    first private(var-list)  
    reduction (operator : var-list)  
    copy in (var-list)  
    num-threads (integer-expression)
```

The number of threads in a parallel region is determined by these factors (in order of precedence)

- (a) Evaluation of the IF clause.
- (b) Setting of the NUM-THREADS clause.
- (c) Use of `omp-set-num-threads()` library function.
- (d) Setting of the OMP- NUM- THREADS environment variable.
- (e) Implementation default-Number of processors (CPU) on a node, though it could be dynamic.
- (f) Threads are numbered from 0 (master thread) to N-1.

Omp-get-dynamic () and Omp-get-nested () library functions

`Omp-get dynamic()` is a library function to determine if dynamic threads are enabled.

`Omp-get-nested()` is a library function to determine if nested parallel regions are enabled.

Limitations of parallel threads (region)

A parallel region has some restriction also and they are as follows

- (1) It must be structured block that does not span multiple routines or code files.
- (2) It is not legal to branch into or out of a parallel region.
- (3) Only a single IF clause is permitted.
- (4) Only a single NUM-THREADS clause is permitted.

13.5 WORK-SHARING CONSTRUCTS

A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it. This construct does not launch any new threads. It must be enclosed dynamically within a parallel region. There are three types of work-sharing constructs:

- (a) **Do/for:** It shares iterations of a loop across the team. It represents a type of “**data parallelism**”.
- (b) **Sections:** It breaks work into separate and discrete sections. Each section is executed by thread. It can be used to implement a type of **functional parallelism**.
- (c) **Single:** It serializes a section of code. Let us discuss these directives one by one.

I. DO/FOR directive

It specifies that the iterations of the loop immediately following it must be executed in parallel by the team. The iteration of the loop are distributed over threads.

Its **syntax** is

```
# pragma omp for [clause....] newline
    Schedule (type [,chunk])
    Ordered
    private (<var-list>)
    first private (<var-list>)
    last private (<var-list>)
    shared (< var-list>)
    reduction (operator : var-list)
    for-loop
```

The following clauses have been used here

- (a) **Schedule:** It describes how iterations of the loop are divided among the threads in the team, (default schedule is implementation dependent).
- (b) **NOWAIT:** If it is specified then threads do not synchronize at end of the parallel loop.
- (c) **Ordered :** It specifies that the iterations of the loop must be executed as they would be in a serial program.

But this Do directive has some restrictions too. They are as follows

- (1) This loop cannot be a do-while loop. The iterator of the loop must be an integer.
- (2) Program correctness must not depend upon which thread executes a particular iteration.
- (3) Branching out of a loop is illegal.
- (4) The chunk size must be specified .
- (5) **Ordered** and **schedule** clauses may appear once each.

II. Sections Directive

The main features of this directive are

- (1) It is a non – iterative work sharing construct.
- (2) It specifies that the enclosed sections of the code are to be divided among the threads in the team.
- (3) Nesting of section directives is done .
- (4) Each section is executed once by a thread.
- (5) Different sections may be executed by different threads.

Its **syntax** is

```
# pragma omp sections [clause....] newline
    private (<var-list>)
    first private (<var-list>)
    last private (<var-list>)
    reduction (<operator: var-list>)
    nowait
{
    # pragma omp section newline
    structured-block
    # pragma omp section newline
    structured-block
```

```
}
```

But please note that it is illegal to branch into or out of section blocks.

III. Single directive

It specified that the enclosed code is to be executed by only one thread in the team. It is basically used when sections of code are not thread safe.

```
# pragma omp single [clause....] newline
    private (<var-list>
        first private (<var-list>
            nowait
            structured-block
```

Please note, however, that it is illegal to branch into or out of a SINGLE block. Also note that the SINGLE pragma tells the compiler that only a single thread should execute the block of code the pragma proceeds.

Parallel work-sharing constructs

OpenMP includes two directives

- (a) PARALLEL Do/Parallel for. They are also known as **shortcuts**.
- (b) PARALLEL SECTIONS.

```
# pragma omp parallel
# pragma omp parallel
    for (...)

# pragma omp parallel
# pragma omp sections{...}
```

We are in a position to solve an example now.

```
// To sum array elements.
# include <omp.h>
# define N 100
# define CHUNK SIZE 10
main ()
{
    int i, chunk;
    float a [N], sum = 0.0;
    chunk = CHUNK SIZE;
# pragma omp parallel for shared (a, b, c, chunk) private (i) \ Schedule (static, chunk)
    for (i = 0; i < n; i+ +)
        sum = sum + a[i];
}
```

Constructs for synchronization

Assume that there are two threads on two different processors. Both t1 and t2 increment a variable x at the same time. Assume x is initialized to 0. That is,

t_1 increment (x1)	t_2 increment (x1)
-------------------------	-------------------------

```

{
    x1=x1+1;
}
Load A, (x1 address)
Add A,1
Store A,(x1 address)
Now, one possible execution sequence is as follows :-  

(1) Thread -1 (t1) loads the value of x1 into register A.  

(2) Thread -2 (t2) loads the value of x1 in register A.  

(3) t1 adds 1 to register A.  

(4) t2 adds 1 to register A  

(5) t1 stores register A at location x1.  

(6) t2 stores register A at location x1.

```

Now, the resultant value of x_1 is 1 but it should be 2. To avoid this problem, the incrementation of x must be synchronized between t1 and t2 to ensure that the correct result is produced. Openmp provides such constructs. Some of these directives are as follows :-

I. Master Directive

It specifies a region that is to be executed only by the master thread of team. All other threads on the team, skip this section of code. **Please note that it is illegal to branch into or out of MASTER block.**

Its **syntax** is

```
# pragma omp master newline
structured-block.
```

II. Critical Directive

If specifies a region code that must be executed by only one thread at a time.

If a thread is currently executing in its **critical region** and another thread reaches that critical region then the second thread will be blocked until first one exits.

Its **syntax** is

```
# pragma omp critical [name] newline
structured-block
```

Please note that it is illegal to branch in or out of a critical block. Also note that all threads in a team will attempt to execute in parallel but whenever a CRITICAL construct encloses some statements then only one thread will be able to work at any time.

III. Barrier Directive

It synchronizes all threads in the team. A thread waits at this directive until all other threads have reached that barrier. All threads then resume executing in parallel, the code that follows the barrier. Its **syntax** is

```
# pragma omp barrier newline
```

But it has some limitations also. All threads in a team or none must execute the BARRIER region. The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.

IV. Atomic Directive

It specifies that a specific memory location must be updated atomically, rather than letting multiple thread attempts to write to it. In nutshell, this directive provides a mini-CRITICAL section.

Its syntax is

```
# pragma omp atomic newline  
state-expression
```

This directive, however, applies to only a single, immediately following statement. An atomic statement must follow a specific syntax.

V. Flush directive

It provides a synchronization point at which the implementation must provide a consistent view of memory. All thread-visible variables are written back to memory at this point. This directive is necessary to instruct the compiler that the variable must be written or read from the memory system. **Please note that it means that the variable cannot be kept in local CPU registers over the flush statement in the code.** Its syntax is

```
# pragma omp flush (<list>) var newline
```

The optional list contains a list of named variables that will be flushed in order to avoid flushing all variable. **Please note that for pointers in the list, pointer itself is flushed and not the objects it points to.**

Also note that this directive is not implied if a NOWAIT/nowait clause is present.

VI. Ordered Directive

It specifies that the iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor. Thread's chunk of iterations will not be executed if previous iterations haven't yet completed. This directive provides a way to fine tune where ordering is to be applied within a loop. Else it is not required.

Its syntax is

```
# pragma omp for ordered [clauses...]  
(loop region)  
# pragma omp ordered newline  
structured-block  
(end of loop region)
```

It has some restrictions also as follows :-

- (1) Only one thread is allowed in an ordered section at any time.
- (2) It is not legal to branch in or out of an ORDERED block.
- (3) The same ORDERED directive must not be executed more than once in an iteration of a loop. Also, it must not execute more than one ORDERED directive.
- (4) A loop having an ORDERED directive must be a loop with an ORDERED clause.

VII. Thread Private Directive

It is used to make global file scope variables local and persistent to a thread through the execution of multiple parallel regions.

Its syntax is

```
# pragma omp thread private (<list>) newline
```

This directive must appear after the declaration of listed variables. Each thread then gets its own copy of the variable, so data written by one thread is not visible to other threads. **Please note that THREAD PRIVATE variables are different from PRIVATE variables because they are able to persist between different parallel sections of a code.**

It has some limitations as well. Data in THREAD PRIVATE objects is guaranteed to persist only if the number of threads in different parallel regions remain constant. Also the dynamic threads mechanism is turned off. This directive must appear after every declaration of a thread private variable.

Summary

Each of the directive may/may not support all of the clauses. Let us summarize them in a tabular form :-

Clause	Directive					
	Parallel	Do/for	Sections	Single	Parallel Do/for	Parallel Section
If	✓				✓	✓
Private	✓	✓	✓	✓	✓	✓
SHARED	✓	✓			✓	✓
DEFAULT	✓				✓	✓
FIRST PRIVATE	✓	✓	✓	✓	✓	✓
LASTPRIVATE		✓	✓		✓	✓
REDUCTION	✓	✓	✓		✓	✓
COPYIN	✓				✓	✓
SCHEDULE		✓			✓	
ORDERED		✓			✓	

13.6 DIRECTIVE BINDING AND NESTING RULES

Directive Binding

- (1) The Do/for, SECTIONS, SINGLE, MASTER and BARRIER directives bind to the dynamically enclosing PARALLEL, if one exists. If no parallel region is currently being executed, the directives have no effect.
- (2) The ORDERED directive binds to the dynamically enclosing Do/for.
- (3) The ATOMIC directive enforces exclusive access with respect to ATOMIC directives in all threads, not just the current team.
- (4) The CRITICAL directive enforces exclusive access with respect to CRITICAL directives in all threads not just the current team.
- (5) A directive can never bind to any directive outside the closest enclosing PARALLEL.

Directive Nesting:

- (1) A parallel directive dynamically inside another PARALLEL directive logically establishes a new team, which is composed of only the current thread unless nested parallelism is enabled,
- (2) Do/for, SECTIONS and SINGLE directives that bind to the same PARALLEL are not allowed to be nested inside of each other.
- (3) Do/for, SECTIONS, and SINGLE directives are not permitted in the dynamic extent of CRITICAL, ORDERED and MASTER regions.
- (4) CRITICAL directives with the same name are not permitted to be nested inside of each other.
- (5) BARRIER directives are not permitted in the dynamic extent of Do/for ORDERED, SECTIONS, SINGLE, MASTER and CRITICAL regions.

- (6) MASTER directives are not permitted in the dynamic extent of Do/for, SECTIONS and SINGLE directives.
- (7) ORDERED directives are not permitted in the dynamic extent of CRITICAL regions.
- (8) Any directive that is permitted when executed dynamically inside a PARALLEL region is also legal when executed outside a parallel region.

OpenMP Library Routines (functions)

The OpenMP standard defines an API for library calls that perform variety of functions:-

- (a) Query the number of threads/processors
- (b) General purpose locking routines (semaphores)
- (c) Portable wall clock timing routines
- (d) Nested parallelism and dynamic adjustment of threads. For C/C++, we may have to specify "omp.h" include file.
- (e) The lock variable must be accessed only through the locking routines.

13.7 MESSAGE PASSING INTERFACES (MPI)

Parallel programming models are divided into implicit parallel programming model and explicit parallel programming model and message passing is an explicit parallel programming model. **Explicit parallelism means that parallelism is explicitly specified in the source code by the programmer using special language constructs, compiler directives or library function calls.**

It consists of processes, with multiple address space and control flow. Users must explicitly allocate data and work load to processes in message-passing programs. These programs are multithreading and asynchronous requiring explicit synchronization's to maintain correct execution order. These progresses have their own separate address space. In message passing two widely standard libraries are used. *i.e.* PVM and MPI which are implemented in virtually all types of parallel computers and greatly help the portability of message passing programs. Here we are elaborating the concept of MPI.

MPI is standard specification for a library of message passing functions, which is based on consortium of parallel computer vendors, library writers and specialists.

MPI achieves portability by providing a public - Domain, platform-independent standard of message passing library.

MPI specifies this library in language independent form and provides FORTRAN and C bindings, that we cover in next section. This specification does not contain any feature that is specific to any particular vendor, operating system or network. MPI has been implemented on IBM PCs, on windows, all main unix workstation and all major parallel computers. This means that a parallel program written in standard C or FORTRAN, using MPI for message passing could run without change on a single PC, a workstation, a network workstation, an MPP from any vendor, on any operating system.

MPI provides functionality based on four concepts that are, message data types, communicators, communication operations and virtual topology.

Parallelism Issues in MPI:

If static processes are assumed, all processes are created when a parallel program is loaded and they stay alive until the entire program terminates. All these processes are grouped together and identified by MPI-COMM-WORLD. Np process can be created/

terminated in middle of a program execution.

13.8 MPI IMPLEMENTATIONS

MPI is not a stand-alone, self contained software system. It serves as message passing communication layer on top of the native parallel program environment, which takes care of process management and input/output.

MPI Messages:

Basically a message is like a letter. We need to specify the contents of message and the intended recipient of the message. The former is called message buffer and latter a message envelope.

MPI processes are heavy-weighted, single threaded process and they have separate address spaces. Thus, one process cannot directly access variables in another process address space. Interprocess communication is realized by message passing.

General Syntax of send routine for MPI is

`MPI_send (buffer, count, datatype, destination, tag, communicator);`

For example: `MPI_send (&N, 1, MPI_INT, i, i, MPI_COMM_WORLD);`

means, it sends to label `1` Integer stored at location `&N` with tag `i` and send it to process `i`.

This routine has six parameters. The first three parameters specify the message address, message count, and message datatype respectively. Last three parameters specify the destination process id of the message, tag and communicator respectively. **These last three parameters together constitute the message envelope.**

Now, every parameter is discussed in detail.

The first argument *i.e.* `&N` indicates the starting address of the message buffer *i.e.* the memory area holding the data to be sent.

- A buffer refers to an application memory area specified by the programmer, where data value of message are stored.
- A buffer could also mean some memory area created and managed by message-passing system which temporarily stores the message while it is being sent.
- User may set aside a memory area of a certain size, to be used as an intermediate buffer to hold arbitrary messages that could appear in users application.

Three types of message buffer are used in MPI.

1. The message is directly transferred between user-level buffers A and B, both of which are declared in users application as shown in figure 2.



Fig.2

2. In second case, the message is firstly temporarily copied into a dynamically created system buffer S, and then deposited in receiver buffer B.

Two problems are there for this case, first, an extra coping introduces addition overhead. Second, a system buffer with sufficient size is not guaranteed.

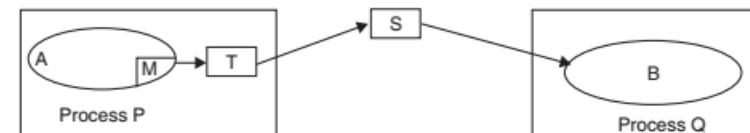


Fig. 3

3. In third case, user first declares a temporary buffer T which is large enough to hold any message that needs to be buffered. During message passing operation the message is first copied into buffer T and then deposited to receiver B. The application program generate an error message and terminate the application if system cannot accommodate buffer T.

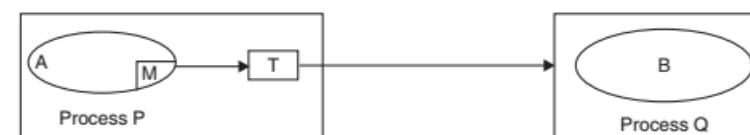


Fig. 4

2. Second argument specifies how many items of certain data type, which is given in third argument are contained in the message.
3. The data type could be either a basic data type or a derived type.

MPI introduces data type attribute to support heterogeneous computing and to allow messages from non-contiguous, non-uniform memory sections.

In order to send a message with non-contiguous data items of possibly mixed datatypes wherein message could consist of multiple data items of different datatypes, MPI introduces the concept of derived datatypes. MPI provides a set of routines to build complex datatypes.

Example:

```

double A[100];
MPI_Data_type Even Elements;
...
...
MPI_Type_vector (50, 1, 2, MPI_DOUBLE, & Even Elements);
MPI_Type_Commit (& EvenElements);
MPI_Send (A, 1, EvenElements, destination, ...);

...
...
MPI_Type_free (& EvenElements);

```

Here, a derived datatype EvenElements is constructed using the routine MPI_type_vector whose format, is as follows:

`MPI_Type_vector (count, blocklength, stride, oldtype, &newtype)`

The derived newtype consist of count copies of blocks, each of which in turn consists of blocklength copies of consecutive items of the existing datatype oldtype. **Stride specifies the number of oldtype elements between two consecutive blocks.** Thus, stride blocklength is the gap between two blocks.

Here only the even numbered elements are considered, skipping the odd numbered elements using the gap, as MPI_type_vector routine creates a derived datatype EvenElements which consists of one double precision number.

MPI-Type-Commit routine is used to commit the derived type before it is used by the MPI_Send routine.

MPI_Type_Free is used to release datatype when it is not required.

4. The fourth argument is the destination process ID (rank).

5. Fifth argument is message 1 tag and sixth argument identifies a process group and a context *i.e.* communicator.

Message Envelope:

MPI uses three parameters to specify a message envelope, the destination process id, the tag, and the communicator.

Destination process id specifies the process to which the message is to be sent.

Message Tag

It is an integer used by the programmer to label different types of messages and to restrict message reception. A **message tag also known as a message type**.

Suppose two send routines and two service routines are executed in sequence in node A and node B respectively.

Process X	Process Y
Send (M, 64, Y)	recv (P, X, 64)
Send (N, 32, Y)	recv (Q, X, 32)

Here tags are not used and the intent is to first send 64 bytes of data which is to be stored in P of process Y and then send 32 bytes of data which is to be stored in Q. If N, although sent later, reaches earlier than M at process Y from process X then there would be an error. Thus, tags are used,

Process X	Process Y
Send (M, 64, Y, tag 1)	recv (P, X, 64, tag 1)
Send (N, 32, Y, tag 2)	recv (Q, X, 32, tag 2)

Now, message M (with tag 1) is guaranteed to be received first. If message N arrives at Q first, it will be queued (buffered) until recv (Q, X, 32, tag 2) is executed.

Message Communicator

A **communicator is a process group plus a context**. A process group is a finite and ordered set of processes. A group has a finite number n of processes, where n is called the **group size**. And the ordering means that the n processes are ranked by integers 0, 1, ... n-1. A process is identified by its rank is communicator (group).

Most MPI users only need to use routines for communication within a group called **intra-communicators in MPI**.

Context in MPI are like system designated supertags that separates different communications from adversely interfering with each other. Also some problems are there with tags, to overcome that communicators are used like, tags are integer values specified by users and user may make mistakes entering the integer values. Even if user does not make mistakes it is difficult to ensure that the value of tag 1 is different from tag 2. Even if user could always know the value of tag 2, errors could still occur because the MPI-Recv routine may decide use a wildcard tag MPI-Any-tag.

Example:

```
Process 0 :
MPI_send (msg 1, count 1, MPI_INT, 1, tag 1, comm 1);
```

```
parallel_findroot ( );
```

Process 1 :

```
MPI_Recv (msg 1, count 1, MPI_INT, 0, tag 1, comm 1)
parallel_findroot ( );
```

The aim of this program is process 0 will send msg 1 to process 1 and then both execute a subroutine parallel= findroot (). Now, process 1 has another send routine with tag 2 value and comm2 value. But in this case, if communicator is not placed, it is possible that MPI-Rec = V may receive msg2 as tag2 may have same value as that of tag1.

MPI has a predefined set of communicators.

Also MPI also provides several routines for building user-defined communicators.

Most MPI users only need to use routines for point-to-point or collective communication within a group. *i.e.* intra-communicators in MPI.

We are in a position to write a message passing program with MPI routines now.

```
// MPI program to calculate  $\Sigma x(i)$ , where
// i = 0, 1, 2, ..., N
# include "mpi.h"
int x(i), i;
int main (argc, argv)
int argc;
char * argv [ ];
{
int i, tmp, sum = 0, group_size;
int my_rank, N;
MPI_Init (large, & argv);
MPI_comm_size(MPI_COMM_WORLD, & group_size);
MPI_comm_rank (MPI_COMM_WORLD, & my_rank);
if (my_rank== 0)
{
printf ("Enter value of N:");
scanf ("%d", &N);
for (i= l; i < group_size; i++)
    MPI_send (&N, 1, MPI_INT, i, i, MPI_COMM_WORLD);
for (i= my_rank; i<N; i = i+ group_size)
    sum = sum + x(i);
for (i = l; i< group_size; i++)
{
    MPI_Recv(&tmp, 1, MPI_INT, i, i, MPI_COMM_WORLD,&Status);
    sum = sum + tmp;
}
printf ("The result = %d", sum);
}
```

```
else
{
    MPI_Recv (&N, 1, MPI_INT, 0, i, MPI_COMM_WORLD, &status);
    for (i=my_rank; i<N, i = i+ group_Size)
        sum = sum + x (i);
    MPI_Send (&sum, 1, MPI_INT, 0, i, MPI_COMM_WORLD);
}
MPI_Finalize( );
}
```

How to run this program?

The following steps are followed

- (1) Firstly, compile this program using parallel C compiler (say, mpcc)–mpcc myfile.c–o myfile.o
- (2) Then, the executable myfile is loaded as n-processes in n-nodes (processes) using this command
MPIRUN myfile –np n

These n-processes which stay alive till the end of the program forms a default process group *i.e.*, MPI_COMM_WORLD.

For process, K (K= 0, 1, 2, n – 1),

MPI_COMM_rank returns K and MPI_COMM_SIZE returns n. Since n is a load-time parameter, the same program can be run on varying number of nodes.

In the above program, we have assumed that N>n. The process with rank 0 reads the input number N and sends it to all other processes. Then, a process with rank K computes the partial sum *i.e.*,

$$\text{sum} = K + x(n + k) + x(2n + k) + \dots$$

All the processes other than the one with rank 0 send their resulting partial sums to process 0, process 0 then adds these partial sums to get the total sum.

MPI assumes static processes *i.e.*, all processes are created when a parallel program is loaded. Please note that no process can be created or terminated in the middle of program execution. Also note that all processes stay alive till the program terminates.

13.9 ADVANTAGES OF MPI

MPI is not a complete programming environment as some aspects of parallel computing are not included here. It's advantages are

- (1) It allows point-to-point communications.
- (2) Message passing using MPI is reliable.
- (3) Good communicators – message passing done by some parts of a software package do not get mixed up with the message passing done by other parts.
- (4) MPI is extensible – debugging, profiling and visualization of parallel programs is possible.
- (5) MPI supports the mapping of the application topologies into virtual topologies and vice – versa. For example, a cartesian mesh topology is a virtual topology. MPI provides routines for defining, examining and manipulating this topology.



CHAPTER

14 OPTICAL COMPUTING- A CASE STUDY

INTRODUCTION

Image processing is an important topic today. Real-time computer vision requires processing images of 1000×1000 data elements within a very small time frame. To achieve the required effect, it is found that the time frame duration should be 16.7 msec. This time suggests processing rate of 10-1000 Gops and input approaching 1Gbytes/sec. These applications require a high degree of data parallelism in which simple arithmetic and logic operations must simultaneously take place across all data points. Computing these applications with high throughput rates requires massively parallel processing.

A massively parallel processor (MPP) was custom designed with sole purpose of image processing. It has a peak computing rate of 6 billion 8-bit integer operations/sec which leads in providing a real-time, time varying scene analysis. This MPP consists of 16,384 (128x128) bit slice microprocessors and is controlled by an ARRAY CONTROLLER. It is a single instruction multiple-data (SIMD) system.

Unique features of optics that makes its use in high-speed parallel processing are

- Speed.
- Parallelism.
- Communications.
- Architectural flexibility.

DEFINITION:

"OPTICAL COMPUTING" refers to use of optics to perform arithmetic, logic & symbolic operations as encountered in modern digital computers.

To achieve optical computing, one demands major improvements in

- Optical materials & devices.
- Memory arrays.
- Interconnection networks.
- Arithmetic/Symbolic algorithms.
- New programming paradigms.

REQUIREMENTS OF FUTURE COMPUTERS

- | | |
|-----------------|---|
| Flexibility | - to run a wide range of algorithms and languages. |
| Extendability | - so that machines can be extended as per user needs. |
| Scalability | - so that one product line using (the same software has machines from small to large (manufacturer's view). |
| Reliability | - failure free operations |
| Fault tolerance | - permits system to continue operating at a reduced capabil- |

Software environment must be efficient and easy to use for appropriate levels of skills of user.

COST EFFECTIVENESS

These features influence computer architecture as follows.

Reconfigurability in parallel machine is required for fault tolerance, flexibility & extendability. Such machines suggests complex interconnection networks.

Massive parallelism is required for extendability, scalability, fault tolerance & high performance. It suggests that interconnection networks have a high bandwidth.

Symbolic computation capability for string manipulation, reasoning and expert systems are required to provide good software environment.

Therefore, there is a move to parallelism and symbolic computation.

LIMITATIONS OF CURRENT COMPUTER TECHNOLOGY

These includes

- Electronic limitations.
- Architectural limitations.
- Software limitations.

ELECTRONIC LIMITATIONS:

Computers have increased their capabilities by shrinking the size of active electronic elements such as transistors on semiconductor chips.

Cost of manufacturing chip doesn't change, so cost per element decreases with increasing density. Small elements have increased speed.

It is increasing difficult to shrink the elements further, because the switching speed is similar to communication speed between elements.

Communication speed doesn't decrease with size, because RC time constant of interconnecting conductor remains constant.

As link shrinks, its capacitance decreases, but resistance increases. Thinner wire presents greater resistance current flow.

Progress in shrinking is hampered by

- * Number of pins for accessing device &
- * Bandwidth per pin are severely limited.

Sol: Optical interconnection to chips.

ARCHITECTURAL LIMITATIONS :

In sequential computers instructions are executed in sequence which involves separating memory, computation, interconnection and control.

High cost of electronic arithmetic units favored channeling everything through a single unit. Sequential architecture uses RAM to store information.

interconnections. Electronic interconnections are expensive in terms of reliability, cost and power as electrons are charged particles which interfere with one another.

In RAM, each item is stored in an address represented in binary form. So n bits can be selected to reference 2^n addresses. A decoder can be used to translate bits to permit selection of address.

- ⇒ Total number of connections to memory is only $n + 1$ for addressing 2^n memory cells.
 - Serial nature of RAM is disadvantage for parallel computing.
 - Tightly coupled processors are needed for a large scientific computation. Massively parallel tightly coupled machines are difficult to construct with electronic sequential architecture.
 - Even using all techniques of pipelining and overlap, improvement in super computer performance with time was flattening out until CRAY moved to replication of system to 2 processors.
- Performance of super computers with Time.

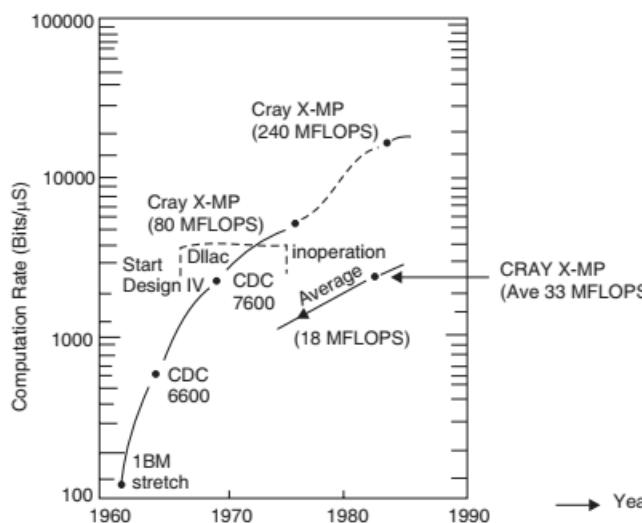
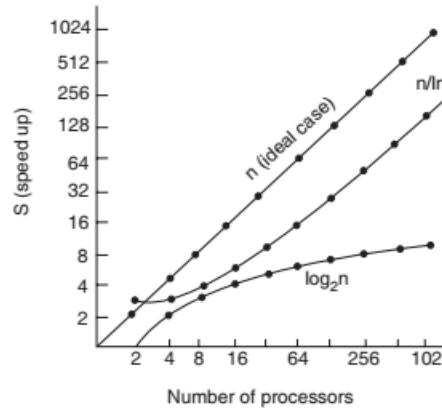


Figure shows the difficulty of increasing performance on a single problem as more processors are used.



For a given computation a parallel algorithm is often possible, that provides greater than 8 speed-up with 256 processors. High performance is rare, because as more processors are used, more time is lost in communication and overhead.

Long computations, for which high performance machines are needed, often have sections of code that are not very parallelizable because they involve gathering data together

from the parts.

SOFTWARE LIMITATIONS :

- Cost of developing software is a major crisis.
- Software accounts 80 % of cost of a new machine and similar % of maintenance cost
- Softwares are people intensive, and capability has not increased exponentially for same cost with time.

Therefore, there is an increasing mismatch between hardware and software resulting in lack of system and application software for many new architectures, parallel machines and new operating system.

- Attempts to overcome the software crises are responsible for emergence of more systematic methodology in organisation and procedures. This is supported by progress towards standardisation to a few languages such as C for real-time, and UNIX-based systems.
- Efforts at providing higher level languages such as LISP and PROLOG have demonstrated that for some applications, such as rapid prototype software development, more of the burden of details may be successfully placed on the computer in order to simplify the effort of programmer.

DIRECTION FOR COMPUTERS :

- 30% of the brain is associated with interfacing to the outside world. Learning is used rather than programming, 10^{11} neurons perform distributed memory and operation functions, and each neuron has upto 10,000 interconnections to other neurons.

This represents high level of parallelism.

Another direction is to use learning by repetition. This provides averaging which avoids the exactness required in programming & represents an approach to solving software crises.

It operates on system inputs and outputs. Associative memory is expensive in electronics, but is a good match in optics.

WHY USE OPTICS IN COMPUTERS ?

The advantages of OPTICS are:

1. **Higher bandwidth that enables more information to be carried.**

This arises because electronic communication along wire requires charging of capacitance that depends on length. Optical signals, Optical integrated circuits & free space don't have to charge a capacitor & are therefore faster.

Faster transmission permits faster computational elements to be used.

2. **Very high speed machines use additional power to provide speed and have elements located close to one another to limit transmission time.**

Transferring heat out of system is a limiting factor.

Ex : CRAY2 uses exotic fluids to transfer heat

3. Faster optical links permits computer elements to be spread farther which relieves the difficulty and cost of heat transfer.
4. Photons are uncharged & don't interface with one another as readily as electrons. So light beams may pass through one another without distorting information carried.

5. Optical memory may be able to avoid difficulties of memory contention, at least during reads.
6. Loops of connection are difficult to avoid in massively parallel system.
In electronics, loops will generate noise voltages spikes whenever electromagnetic fields through loop changes & high frequency or fast switching pulses will cause interference in neighboring wires.
7. Signals in adjacent fibres or in optical integrated channels don't interfere with each other, nor do they pick up noise due to loops.
8. In Optics, images or arrays of pixels may be handled in parallel. Lack of interference between photons made it difficult to use a small signal to control a larger signal for producing gain, as in a transistor.
9. Now, very high-speed, low-switching energy devices increase the non-linearity using GaAs quantum.
Therefore optical switches have comparable performance.
10. Superior storage and accessibility of optical materials over magnetic materials.
Magnetic disks require floating of pick-up coils within one micron of surface Optical disks use focussed laser beams to read the information, so that the light source doesn't have to be as close to storage material.

OPTICAL COMPUTING DEVICES :

Optically bistable devices can play a part in memory because they are essentially single-bit memories that can be both written to and read.

Another way to build such an optical flip-flop is by using liquid crystal light valves (LCLV). Light shines through the polariser, hits the LCLV and is then reflected back out through the polariser. If a voltage is applied to LCLV, polarisation of incoming beam is rotated so that it is perpendicular to the polariser, and it can't make it back out - it looks black. This is how visible digits are displayed on digital displays.

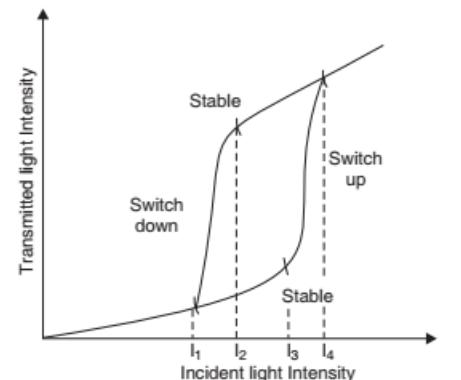
Optical bistable gates can perform logical AND,NOR or memory functions over inputs and outputs represented by light intensities as characterised in figure.

The device is nonlinear with positive feedback to ensure bistability. The I/O characteristics follow a hysteresis cycle in which

logical 0 - incident light intensity from I_1 to I_3

logical 1 - incident light intensity from I_2 to I_4

Switching states take place between 1 and 0 for 0 \rightarrow 1 and between 1 to 0 for 1 \rightarrow 0



Input-Output characteristics of an optical bistable device



Functionally, an optical bistable device with two light inputs x and y is shown. Control signal c decides the function, and power input p establishes the intensity bias value. The outputs q and \bar{q} are related to inputs by equation:

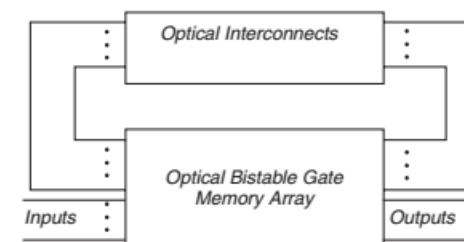
$$\text{New } q = \bar{c} \cdot q + c \cdot (\overline{x + y})$$

$$\text{New } \bar{q} = \bar{c} \cdot \bar{q} + c \cdot (\overline{x + y})$$

So device can be used as a logical OR or a logical NOR gate when control signal $c = 1$ and as pure memory device latch with New $q = q$ and New $\bar{q} = \bar{q}$ when $c = 0$.

Besides optical bistable devices and arrays, spatial light modulators (SLMs) are needed for storing, modifying and amplifying digital optical signals. The SLMs can be electrically and optically controlled and can process optical signals in one-dimensional, two-dimensional or three-dimensional formats.

ARCHITECTURE OF OPTICAL COMPUTER :



Optical bistable gate/memory arrays are used to build the CPU, the memory and I/O interfaces. These functional arrays are interconnected by some form of optical network. These are called **optical interconnection networks** which provide interprocessor and interprocessor-memory-I/O communications.

OPTICAL INTERCONNECTION NETWORKS

Characteristics of Optical Interconnection Networks :

Networks	Bandwidth	Reconfiguration Speed	Broadcast Capability	Parallelism	Rearrangability
Matrix-vector multiplication	Medium	Fast, control matrix	Yes	High	Yes, control matrix
VGM crossbar	Low	Slow, VGM device	No	High	Yes, voltage vector
Parallel optical bus	High	NA	No	Medium	Not possible
Hologram	Medium	Fast, computer generated hologram	No	High	Yes, computer generated hologram
Optical perfect shuffle	Medium	Medium, dependent on number of stages	No	Low	O (log N) limited permutations

Important issues in designing optical interconnection networks includes

- parallel vs volume (bulk)
- packet vs circuit switched

- guide wave vs free space
 - unidirectional vs bidirectional
 - optical vs electronic control
 - asynchronous vs synchronised operations
- Other important parameters that affect the performance and efficiency includes
- Reconfiguration time
 - Communication bandwidth
 - Propagation delay time
 - Cascadability
 - Broadcast capability

Electronic technologies are approaching their limits in terms of switching speed and package, requirements. As microcircuits become denser, the problems of clock skew, bandwidth, crosstalk and capacitive loading are intensified.

Using Optical networks many of these limitations can be lifted to allow construction of large scale networks to support massive parallelism.

Optical networks may consume less power, are immune to crosstalk and interference and lend themselves to, multiplexing and switching. There are two broad categories of optical interconnections.

Index guided: Based on use of optical fibres or integrated optical waveguides.

Free space: Simple optical-broadcast or imaging interconnections based on use of holographic optical elements.

An optical interconnect requires

1. **Source** converts electrical signal to an optical signal, either by direct electrical modulation of source or by external modulation of optical source.
2. **Receiver** consists of a detector that transforms the optical energy into electrical charge,
3. and an **amplifier** that produces the required digital electrical levels.
4. Optical waveguide which directs signal from source to receiver.

Optical beams need to use drivers and detector such as laser diodes, LEDs and opto-electrical transducers. Transmission media includes free space, fibers or planar waveguides.

Optical buses cannot be used as replacement to electrical buses. Significant increases in control complexity and power consumption may be encountered in using multiple optical buses. Time division multiplexing allows us to carry $(m \times n)$ signals in an m-channel optical fiber. Multiplexing signals onto an optical bus reduces the number of interconnections, uses less power, and offers higher application flexibility. Multiplexed optical bus include fixed connections and no broadcast capability.

OPTICAL CROSSBARS AND HOLOGRAM NETWORKS :

At the level of processing elements, the interconnection networks used range from shared bus to crossbar networks. A **crossbar** is a network that allows arbitrary one-to-one interconnection of **n** inputs to **n** outputs without contention.

There are several features of optics that make optical switching networks an attractive alternative to their electronic counterparts. These includes—

Speed : Speed of propagation of an optical signal does not depend upon number of components receiving it.

Parallelism : Optical systems are inherently parallel and 2-dimensional.

Bandwidth : Optical systems can deliver large bandwidths in both temporal and spatial domains.

Optical crossbar networks have been recently developed using

- directional optical couplers.
- spatial light modulators.

Directional optical couplers are 2×2 crossbar switches where two input signals can be either directly connected to output or swapped and then connected to output. The connection obtained can be controlled by applying a dc voltage across the coupler.

A **spatial light modulator** is a real time reconfigurable device capable of modifying the amplitude, phase or polarisation of an optical wavefront. These are used in building a whole range of optical crossbars. Here function of crossbars can be realised by matrix-vector multiplication.

An optical crossbar switch can be constructed with free-space optics, input laser diodes, variable grating mode (VGM) devices, Fourier transform lens, spatial filters and detector arrays. The VGM device performs an intensity-frequency conversion over a two-dimensional field. VGM directs input signals to one of several spatially separate output channels. The spatial filter masks off the unwanted diffraction orders.

VGM network consists of two fourier transform lenses and a switching matrix made from a liquid crystal light valve (LCLV). The input vector consists of a row of LEDs. The control of LCLV generates a matrix of light or dark spots. The input vector is multiplied by LCLV to form output vector. The LCLV crossbar network has lower reconfiguration overhead and broadcast capability.

Hologram based networks use 3D for free-space optical transmissions. A hologram is a beam-steering array used to establish arbitrary interconnections. Holograms can be used as **defractor** or a **deflector**. Defractor is mainly used for board-board interconnections. Deflector is used for chip-chip interconnections.

COMPARISON BETWEEN ELECTRONIC & OPTICAL COMPUTERS :

Characteristics	Electronic computers	Optical Computers
Switching speed	$O(10^{-2})$ - $O(10^{-11})$ s in SLGaAs, JJ, HEMT	$O(10^{-12})$ - $O(10^{-15})$ s in optical gate arrays
Processor granularity and parallelism	Large-grain and $O(1)$ - $O(10^4)$ processor per system	Fine-grain and massive parallelism
Communication Bandwidth	$O(10^1)$ - $O(10^3)$ Mbits/s	$O(1)$ - $O(10^2)$ Gbits/s
Physical integration	$O(10^4)$ - $O(10^5)$ per CMOS chip	$O(10^2)$ - $O(10^6)$ gate per optical array
Control Complexity	Synchronous with digital clocking	Synchronous/Asynchronous (absence of clock-skewing)
Power requirements	$O(10^2)$ W per board at $O(10)$ MHz clock	$O(10^{-2})$ W per array at $O(10)$ MHz clock rate
Reliability & Maintenance	Architecture and Technology dependent	Less interference and could be very costly

Deflective hologram uses the surface of a chip as a matrix of pin-outs interconnecting internal circuits. Hologram interconnections are reconfigurable. Major drawback is its low tolerance for misalignment of light beams and deflections.

Optical shuffle-network can be constructed using free-space optics or integrated optics. Holograms can also be used to construct the free-space shuffle network.

Integrated perfect shuffle is implemented with a waveguide network using hybrid optical/electrical devices.

OPTICAL SYMBOLIC ARITHMETIC

Methods proposed for performing digital arithmetic operations on optical data.

1. **Residue number system** : for carry-free addition, subtraction and multiplication. It suffers from difficulties in performing division, comparing two numbers, and converting a residue number to a conventional representation.
2. **Symbolic substitution** : for arithmetic and logic operations. Techniques involves the recognition of spatial patterns of binary coded data and replaces them according to a set of predefined symbolic substitution rules. These rules are based on conventional number representation which demands carry propagation and requires an $O(n)$ addition time, where n is operand length.
3. **SD number representation can be used in conjunction with Symbolic substitution.** By doing so, we can significantly improve the speed, accuracy and dynamic range of digital arithmetic. It limits carry propagation only to adjacent digital position for addition. Thus totally parallel addition becomes possible with a digital set of $\{1, 0, \bar{1}\}$.

The ability of implementing a symbolic substitution scheme with high-speed optical components and the provision of a carry-constrained arithmetic system permit us to develop a faster **optical arithmetic processor**.

OPTICAL NUMERICAL PROCESSING

Here numerical values are represented by analog light intensities. When coherent light passes through a transparency, its amplitude is multiplied at each point by the transmittance of that transparency.

ADD:

Addition is performed by modulating the separate beams and then combining them with a beam splitter. A beam splitter is just a semi-transparent mirror. Part of the light that hits it is reflected. The rest goes through it without changing direction. To join beams, the two input beams are directed in such a way that the reflected part of one of the beam and the transmitted part of other beam come together.

Thus the output contains both. As the light is coherent, the light beams interfere and so their complex amplitudes are added.

SUBTRACTION:

By phase shift. To do so a phase plate is inserted into path of one beam. This is just a flat plate of glass or a crystal whose thickness is such that light passing through it suffers a delay of $1/2$ a wavelength relative to light that does not.

MULTIPLICATION:

When light passes through a transparency, it is multiplied by the transparency's transmittance. Therefore to multiply 2 data images, a light beam is passed through a sandwich of their two transparencies.

DIVISION:

Division can always be expressed as two multiplications and can be accomplished with 3 transparencies.

APPLICATIONS:

Computation intensive problems that can be more effectively solved by optical computers includes

- * Signal/Image Processing
- * Computer vision
- * Pattern Recognition
- * Robotics Control
- * Artificial Neural Computing
- * Machine Intelligence
- * Supercomputing in Scientific Simulations
- * Optimisations.

LIMITATIONS:

Digital optical device developments are still in their earlier stages.

Data must be converted to optical form to use an optical-based communication medium. Signal conversions may cause performance degradation and increase power consumption

In optical devices, it is theoretically possible to achieve switching speeds in sub-picosecond range, but this falls outside thermal transfer limit and makes these device impractical.

Having fewer photons would reduce the power requirements, but, at the most practical wavelengths, the wave energy equivalent of at least 1000 photons needs to work in concert to avoid statistical fluctuations that would lead to switching errors.

INDIAN SCENARIO:

The Optocomputing project at C-DAC is visualised as one of the components and technology goals to be realised under India's National Photonic Initiative. The project goal is specifically directed to integrate photonic technology into the national project of development of a parallel super computer that is being pursued at C-DAC. The second objective is to integrate optical interconnect technology specifically optical crossbars within the PARAM architecture.

REFERENCES:

Advanced Computing	:	C-DAC
IEEE Micro	:	Ahmed Louri
Optical Computing	:	Atister McAulay



A= APPENDIX

A. GLOSSARY

1. **Address** : The number of a particular memory or peripheral storage location.
2. **ARM** : An ARM is a RISC processor invented by Advanced RISC machines, currently owned by Intel.
3. **Asynchronous** : Non-clocked systems.
4. **Broadcast** : It refers to sending the same message to all processes.
5. **Buffer** : Specialized memory area to store temporary data.
6. **Cache coherence problem** : The problem of inconsistent data in caches and main memory in multiprocessor system.
7. **Cluster** : Two or more computers used together to solve a problem.
8. **Cluster of workstations (CoW)** : It is a very large-scale computer system designed for parallel computing.
9. **Collective communication** : When all the processes in a group participate in a global communication.
10. **Control strategy** : It is the procedure to choose a latency sequence.
11. **Clock**: A periodic electrical signal.
12. **Deadlock** : When packets cannot be forwarded to the next node due to blocking.
13. **Failover** : The process of taking over a failed node's and/or resources.
14. **Initiation** : It means the start of a single function evaluation on a pipeline.
15. **Initiation rate** : Average number of initiations per clock unit is known as initiation rate.
16. **Light weight** : Something that will not consume computer resources.
17. **Live lock** : A situation in which a packet keeps going around the network without ever finding its destination.
18. **Monitor** : It is a suite of procedures that provides the only method to access a shared resource.
19. **Multicast** : It refers to sending the same message to a defined group of processes.
20. **Multitasking** : The running of two or more programs in one computer at the same time.
21. **Multithreading** : Multitasking within a single process.
22. **Multiuser** : A system which allows more than one user. Eg. Unix, NT.

23. **Nu Bus** : Expansion slots on MAC which accepts intelligent, self-configuring boards.
24. **Oct tree** : A tree in which each node has eight children.
25. **Parallelizing compiler** : A compiler that converts a sequential code into parallel code.
26. **PA-RISC** : It is a RISC processor developed by HP.
27. **Port** : A connection socket or jack on the MAC.
28. **Profile** : A profile of a program is a histogram or graph showing the time spent on different parts of the program.
29. **Quad tree** : A tree in which each node has four children.
30. **Semaphore** : A shared space for inter-process communications (IPC) controlled by “wake-up” and “sleep” commands.
31. **Stage utilization** : It is given by the busy time span for the stage divided by total time span.
32. **Temporal** : It means pertaining to time.
33. **Vector length** : It determines the termination of vector instruction.
34. **Word** : 1 word = 2 bytes.
35. **Wormhole Routing** : It is a way of pipelining packet transmission in a multihop network.

B APPENDIX

ACADEMIC PLAN FOR VII SEMESTER (FOR YEAR 2008-09)

[GGSIPU, DELHI]

SUBJECT: ADVANCED COMPUTER ARCHITECTURE

Branch. B. Tech. (CSE); 7th Sem.

Subject Code: ETCS 403

Total Lectures: 42

Total Tutorials: 14

Total teaching weeks in semester: 14 weeks

	TOPICS TO BE COVERED	Total No. of Lecture/Tutorial	
		Lecture	Tutorial
	First Term		
1.	Parallel computer models	1	
2.	The state of computing	1	
3.	Multiprocessors and multi-computers	1	1
4.	Multi-vector and SIMD computers, Architectural development tracks	1	
5.	Program and network properties		
6.	Conditions of Parallelism Data and resource dependencies.	1	
7.	Hardware and software parallelism	2	1
8.	Program partitioning and scheduling, Grain size and latency	1	
9.	Program flow mechanisms	1	
10.	Control flow versus data flow, Demand driven mechanisms	1	1
11.	Data flow architecture	2	
12.	Comparisons of flow mechanisms	1	1
13.	System Interconnect Architectures		
14.	Network properties and routing, Static interconnection networks	1	
15.	Dynamic interconnection Networks	2	1
16.	Multiprocessor system interconnects, Hierarchical bus systems, Crossbar switch and multi-port memory, Multistage and combining network.	3	1
	Second Term		
17.	Processors and Memory Hierarchy		
18.	Advanced processor technology, Instruction-set Architectures	1	

15.	CISC Scalar Processors, RISC Scalar Processors	1	
16.	Superscalar Processors, VLIW Architectures	2	1
17.	Vector and Symbolic processors	1	
Memory Technology			
18.	Hierarchical memory technology, Inclusion Coherence and Locality, Memory capacity planning	2	1
19.	Virtual Memory Technology	2	1
Back plane Bus System :			
20.	Back plane bus specification, Addressing and timing protocols		
21.	Arbitration transaction and interrupt	1	
22.	Cache addressing models, Direct mapping and associative caches	1	
Pipelining			
23.	Linear pipeline processor	1	
24.	Nonlinear pipeline processor	1	
25.	Instruction pipeline design, Mechanisms for instruction pipelining, Dynamic instruction scheduling, Branch handling techniques	2	
26.	Arithmetic Pipeline Design, Computer arithmetic principles, Static arithmetic pipeline, Multifunctional arithmetic pipelines		
Third Term			
Vector Processing Principles			
27.	Vector instruction types	1	
28.	Vector-access memory schemes	1	
Synchronous Parallel Processing			
29.	SIMD Architecture and Programming Principles, SIMD Parallel Algorithms	2	
30.	SIMD Computers and Performance Enhancement	1	

Text Books:

1. Rajiv Chopra , " Advanced Computer Architecture",— A Practical Approach; S.Chand & Company Ltd. , 3rd Revised Edition , 2013

C **APPENDIX**

B. GGSIPU's

End-term Exam.

Question Paper (2003–2012)

END-TERM EXAMINATION

FIRST SEMESTER [M.TECH. (IT) – WEEKEND PROGRAMME] – DECEMBER 2003

Paper Code : ITW-601 Subject : Advanced Computer Architecture

Time : 3 Hours

Maximum Marks : 60

Note : Attempt any five questions. All questions carry equal marks.

1. Indicate where each of the following statements is true or false and justify your answers with reasoning and supportive of counter examples :
 - (a) The CPU computations and I/O operations cannot be overlapped in a multiprogrammed computer.
 - (b) Synchronization of all PEs in an SIMD Computer is done by hardware rather than by software as often done in most MIMD computers.
 - (c) As far as programmability is concerned, shared-memory multiprocessor often simpler interprocessor communication support than that offered by message-passing multicomputer.
 - (d) In a MIMD computer, all processor must execute the same instruction at the same time synchronously.
2. Answer the following questions related to multistage networks :
 - (a) How many legitimate states are there in a 4×4 switch module, including both broadcast and permutation ? Justify your answer with reasoning.
 - (b) Construct a 64-input Omega network using 4×4 switch modules in multiple stages. How many permutations can be implemented directly in a single pass through the network without blocking ?
 - (c) What is the percentage of one-pass permutations compared with the total number of permutations achievable in one or more passes through network.
3. Answer the following questions on designing scalar RISC or super scalar RISC processor :
 - (a) Why do not RISC integer units use 32 bit general purpose register ? Explain the concept of register windows implemented in the SPARC architecture.
 - (b) What are the design trade-offs between a large register file and a large D-case ?

- (c) Explain the relationship between the integer unit and the floating point unit in most RISC processors with scalar or super scalar organization.
4. (a) Explain the following terms associated with cache and memory architectures.
(i) Low-order memory interleaving.
(ii) Physical address cache versus virtual address cache.
(b) Explain the following terms associated with cache design.
(i) Write through versus write-back cache.
(ii) Cache flushing policies.
5. Consider the five stage pipelined processor specified by the following reservation table :

	1	2	3	4	5	6
S ₁	X					X
S ₂		X			X	
S ₃			X			
S ₄				X		
S ₅		X				X

- (a) List the set of forbidden latencies and the collision vector.
(b) Draw a state transition diagram showing all possible initial sequences without causing a collision in the pipeline.
(c) List all the simple cycles from the state diagram.
(d) Identify the greedy cycle among the simple cycle.
(e) What is the MAL of this pipelines.
(f) What will be maximum throughout of this pipeline ?
6. (a) Prove that the number of legitimate states in a $K \times K$ switch module equals K^K .
(b) Determine the percentage of permutation that can be realized in one pass through a 64-input Omega network built with 2×2 switch modules.
7. Explain structural and operational differences between register-to-register and memory-to-memory architectures in building multipipelined super computers for vector processing. Comment on the advantages and disadvantages in using SIMD computers as compared with the use of pipelined super computers for vector processing.
8. Write short notes (any two)
(a) Omega Network
(b) Cross bar Network
(c) Hierarchical Bus System



END-TERM EXAMINATION

FIRST SEMESTER [M.TECH.] – MAY 2003

Paper Code : IT-310 Subject : Advanced Computer Architecture**Time : 3 Hours****Maximum Marks : 60****Note : Attempt any five questions. All questions carry equal marks.**

1. (a) Characterise the architectural operations of SIMD and MIMD computers. Also distinguish between a multiprocessor and multicomputers based on interprocessor communications.
(b) Indicate whether each of the following statement is true or false, justify your answer.
 - (i) CPU computations and I/O operations cannot be overlapped in a multiprogrammed computer.
 - (ii) As far as programmability is concerned, shared memory multiprocessor offer simplex interprocessor communication support than that offered by message passing multicomputers.
 - (iii) In an MIMD computer, all processors must execute the same instruction at the same time synchronously.
 - (iv) As far as scalability is concerned, multicomputers with distributed memory are more scalable than. Shared memory multicomputers.

2. What do you understand by “grain packing approach introduced by kruatrachue and Lewis for parallel programming applications. Apply this approach to reduce from 17 nodes in fine grain to 5 nodes is coarse grain for the below given operations:

- | | | |
|--------------|--------------|--------------|
| 1. a: = 1 | 2. b: = 2 | 3. c: = 3 |
| 4. d: = 4 | 5. e: = 5 | 6. f: = 6 |
| 7. g: = axb | 8. h: = cxd | 9. i: = dxe |
| 10. j: = exf | 11. k: = dxz | 12. l: = jxk |
| 13. m: = 4xl | 14. n: = 3xm | 15. o: = nxz |
| 16. p: = oxl | 17. q: = pxq | |

Note : nodes 1 to 6 are memory reference (data fetch) operations each of one cycle to address and 6 cycles to fetch. Other nodes 7 to 17 are all CPU operation requiring Z cycles

3. (a) Define the following terms for various system interconnect architecture :
 - (i) Digital bus
 - (ii) Network diameter
 - (iii) Non blocking network
 - (iv) Multistage networks.
(b) Differentiate between static and dynamic connection networks, give a few examples.
4. Explain the inclusion property and memory coherence requirements in a multilevel memory hierarchy. Distinguish between write through and write back policies in maintaining the coherence in adjacent levels. Also explain the basic concepts of

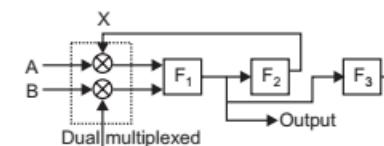
paging and segmentation in managing the physical and virtual memories in a hierach.

5. Based on your understanding of advanced processor, answer the following questions on RISC, CISC, super scalar and VLIW architecture.
 - (a) Compare the instruction set architecture in RISC and CISC processors in terms of instrcution format, addressing modes and CPL.
 - (b) Distinguish between scalar RISC and super scalar RISC in terms of instruction issue, pipeline architecture and processor performance.
 - (c) Explain the difference between super scalar and VLIW architecture in terms of hardware and software requirement.
6. Three functional pipelines f_1 , f_2 and f_3 are characterised by the following reservation tables :

	1	2	3	4
S_1	X			
S_2		X		
S_3			X	X

	1	2	3	4
T_1	X			
T_2		X		
T_3			X	

	1	2	3	4
U_1	X		X	
U_2				X
U_3	X			



- (a) Complete the composite reservation table for the composite pipelines with 9 rows ($S_1, S_2, S_3, T_1, T_2, T_3, U_1, U_2, U_3$) and 12 columns (1, 2, , 12).
- (b) Write the forbidden list and the initial collision vector.
- (c) Draw a state diagram clearly showing all latency cycles.
- (d) List all simple cycles and greedy cycles.
- (e) Calculate mal and maximal throughout of this composite pipeline.
7. Answer the following questions related to data flow computers.
 - (a) Distinction between static data flow computers and dynamic data flow computers.
 - (b) Draw a data flow graph showing the computations of the roots of sequence of quadratic equations.

$$A_1x_2 + B_1x + C_i = 0 ; i = 1, 2 \dots N$$
8. Write short note on ANY THREE of the following :
 - (a) What is non-vewmann architecture computer ? Compare von-neumann architecture vis-a-vis data flow architecture computer. Which architecture is more suitable for parallel algorithm and why ?
 - (b) Describe data routing system in Omega network.
 - (c) Draw block diagram of SISD and MIMD computers. Which configuration is ideal for future supercomputing ?
 - (d) Describe cache flushing policies.



END-TERM EXAMINATION

FIRST SEMESTER [M.TECH.] – DECEMBER 2004

Paper Code : ITW-601 Subject : Advanced Computer Architecture

Time : 3 Hours

Maximum Marks : 60

Note : Attempt any five questions. All questions carry equal marks.

1. (a) With the help of block diagrams, explain Flynn's classification of computer architectures.
(b) What are the system attributes to performance of a computer system. Discuss in brief.
 2. (a) Classify parallel computers based on their memory access mechanisms.
(b) Explain Bernstein's conditions for parallelism.
 3. (a) Discuss the concept of grain packing and scheduling with the help of an example.
(b) Compare control flow and data flow computers.
 4. (a) With the help of diagrams discuss chordal ring, barrel shifter, mesh and torus interconnection networks with respect to node degree, network diameter and bisection width.
(b) With the help of diagram, explain the switching process in a 16×16 Omega network.
 5. (a) Compare CISC and RISC architectures.
(b) With the help of pipeline diagrams, explain superscalar and VLIW processors.
 6. (a) Describe inclusion, Coherence and locality of reference properties of a memory hierarchy.
(b) Explain various address translation mechanism in a virtual memory.
 7. Consider a five stage pipelined processor specified by following reservation table:

	1	2	3	4	5	6
S ₁	X					X
S ₂		X			X	
S ₃			X			
S ₄				X		
S ₅	X					X

**MINOR - I****M.TECH. (IT WEEK END) ITW (601)****Advanced Computer Architecture****Max. Marks : 30****Note : Attempt question 1 which is compulsory any two questions from the rest.**

1. **(Compulsory) attempt any five parts.**
 - (a) Differentiate UMA and NUMA multiprocessor model.
 - (b) Describe Bemsteins conditions, how can you use these to decide, if two processes can be executed in parallel ?
 - (c) Distinguish data flow computer from conventional von Neumann computer.
 - (d) Is Omega network blocking or non blocking ? What is the difference between blocking & non blocking network ?
 - (e) What do you understand by the term hit ratio ? In a memory hierarchy of n level, what is the value of hit ratio at the nth level ?
 - (f) Distinguish CISC & RISC instruction architecture ? (2×5)
2.
 - (a) What do you understand by the term grain size ? How does grain packing improve the performance ?
 - (b) In a 16×16 omega network built with 2×2 switches, how many stages and total number of switches will be required ? If instead of 2×2 switch we use 4×4 switch will the number of stages remain the same ? State the relation. (4+6=10)
3.
 - (a) Describe super scalar, super pipe line and vector processor in comparison with a scalar processor.
 - (b) In a 2 level memory system there are 8 virtual pages on a disk to be mapped on to 4 page frames (PF's) in the main memory, A certain program generates the following page trace.
1, 0, 2, 7, 1, 7, 6, 7, 0, 1, 2, 0, 6
Show the successive virtual pages residing in the four page frames with respect to the above page traces using the LRU replacement policy. Compute the hit ratio in the main memory. Assume the PF's are initially empty. (4+6=10)
4. Write notes on the following :
 - (a) Explain the use of overlapped windows for parameters passing between the calling and called procedures.
 - (b) Describe TLB as a part of virtual memory technology. What do you think will change with 64 bit unified processing ?
 - (c) Describe Flynn's classification, specially compare SISD with MIMD structure. (3+4+3=10)



M. Tech. MINOR - II

Subject : Advanced Computer Architecture

Time : 3 Hours

Max. Marks : 60

Note : Q 1 is compulsory attempt any two questions from the rest

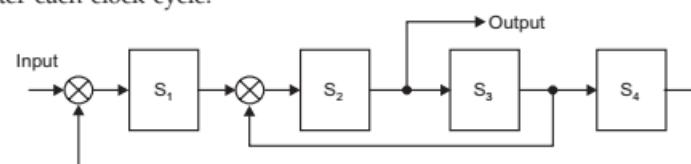
I. Write briefly on the following

1. Synchronous Versus Asynchronous data bus timing protocols.
2. Physical versus virtual addressing.
3. Cache Memory interleaving, bandwidth and fault tolerance.
4. Pipeline "Reservation Table" and how it helps to study the performance of pipeline.
5. Explain Tomasulo's Algorithm. $(1 \times 5 = 5)$

II. s

- (a) The main memory of a computer is organized as 64 blocks each block of 8 words. The cache has 8 blocks frames. Draw all lines showing mapping as clearly as possible for the sector mapping with four block per sector and the address bits that identify the sector number the block number and the word number.
- (b) Consider a two-level memory hierarchy, M1 and M2. Denote the hit ratio of M1 as h. Let c_1 and c_2 be the costs per kilobyte, s_1 and s_2 the memory capacities, and t_1 and t_2 the access times, respectively. Under what conditions will the average cost of the entire memory system approach c_2 ? $(2 \times 2.5 = 5)$

III. Consider the following pipelined procedure with four stages. This pipeline has a total evaluation time of six clock cycles. All successor stages must be used after each clock cycle.



- (a) Specify the reservation table for this pipeline with six columns and four rows.
- (b) List the set of forbidden latencies between task initiations.
- (c) Draw the state diagram which shows all possible latency cycles.
- (d) What is the value of the minimal average latency?
- (e) What is the maximum throughput of this pipeline? (5)

IV. Write note on any two :

- (a) Use of non-compute stages to lower MAL.
- (b) Effect of branching and degradation of performance.
- (c) Daisy chain V/s distributed arbitration scheme.
- (d) Optimal number of pipeline stages. $(2 \times 2.5 = 5)$
- (e) What is the maximum throughput of this pipeline?



END-TERM EXAMINATION

SEVENTH SEMESTER [B.TECH.] – DECEMBER-2007

Paper Code: ETCS 403 (Batch 2004) Subject : Advanced Computer Architecture
Paper ID: 27403

Time : 3 Hours **Maximum Marks : 75**

Note : Attempt one question from each unit Q. No. 1 is compulsory

1. (a) Give five performance factors and explain four system attributes.
(b) Mention the number legitimate states and permutation connections for $n \times n$ and 8×8 switch modules.
(c) Discuss arbitration, transaction and interrupt w.r.t. backplane bus system.
(d) Describe any three vector instructions types in cray-like computers.
(e) Draw a 3×3 mesh, illiac and torus network. (25)

UNIT - I

- Explain the architectural operations of SIMD and MIMD computers. Distinguish between multiprocessors and multicomputers based on their structures, resource sharing and interprocessor communications. (12.5)
 - Use Bernstein's conditions to detect the parallelism embedded in this code.

- P1 : $C = D \times E$
- P2 : $M = G + C$
- P3 : $A = B + C$
- P4 : $C = L + M$
- P5 : $F = G \div E$

Also draw a dependence graph showing both the data dependence and resource dependence. (12.5)

UNIT - II

4. Compare K-ary n-cube, hypercube and star network type based on the following static network characteristics.

 - (a) Node degree
 - (b) Network diameter
 - (c) No. of links
 - (d) Bisection width
 - (e) Symmetry

5. Discuss Memory Hierarchy Technology. Explain inclusion, Coherence and Locality properties w.r.t. memory hierarchy. (12.5)

UNIT - III

6. For synchronous model (pipelines) derive the formulas for calculating clock skewing, speedup, efficiency, throughout and optimal number of pipeline stages. What is performance/cost ratio ? (12.5)

7. Compare the relative merits of the three cache memory organizations.
 (a) Fully Associative Cache
 (b) Set Associative Cache
 (c) Direct Mapping Cache. (12.5)

UNIT - IV

8. Write short notes on **any two** :
 (a) SIMD parallel algorithms
 (b) Vector-Access Memory Schemes
 (c) VLIW architectures. (12.5)
9. Explain the following terms related to vector processing.
 (a) Vector and Scalar Balance point
 (b) Gather and Scatter instructions
 (c) Vectorisation compiler
 (d) Vectorisation ratio in user code (12.5)

**B. KARNATAK UNIV. DHARWAD***End-term Exam.**Question Papers (1996-1997)*

Eighth semester B.E. (CS & E) degree examination, 1996

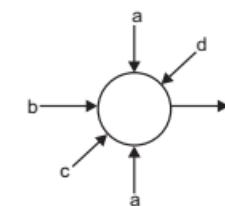
ADVANCED COMPUTER ARCHITECTURE**Time : 3 Hours****Maximum Marks : 100**

Note : Answer any two questions from Unit I, any two from Unit II and any one from Unit III.

UNIT - I

1. (a) What is computer architecture ?
 (b) What is meant by temporal parallelism and spatial parallelism in computers ? How these parallelism are achieved ?
2. (a) Explain how the throughput in a pipelined processor increases compared to a non-pipelined processor.
 (b) Prove that a K-state linear pipeline can be atmost K times faster than that of a non-pipelined serial processor.
3. (a) Explain with diagram a SIMD computer.
 (b) Following is a cell whose inputs and outputs are indicated.
 $d = a * b + c.$

Use this cell to multiply two $[2 \times 2]$ matrices. Calculate how many cells like this are required to get the result.



UNIT - II

4. (a) Draw the schematic of Illiac-IV, the SIMD computer and explain where data routing and manipulation functions are done.
(b) Draw a single-stage recirculating mesh connected network for $N = 16$ processing elements (PEs). Give the four permutations $R + 1, R - 1, R + 4, R - 4$ and the upper bound on steps to route data from PP_i to PE_j .
5. (a) Explain the different bus connections existing on a MULTIBUS-II bus.
(b) Describe Message Passing Co-processor (MPC) used in multibus-II bus structure.
6. (a) Give the block schematic of a static data flow computer.
(b) Differentiate between Static data flow computers and Dynamic data flow computers.

UNIT - III

7. (a) Describe the basic principle of systolic array and explain for which type of applications, they are more suited.
(b) Compare RISC architecture with CISC architecture.
8. (a) Explain characteristics of an optical bistable device.
(b) Compare the components used in a conventional computer with that used in an optical computer.



BE's CS Degree Examination, 1997

ADVANCED COMPUTER ARCHITECTURE

Time : 3 Hours

Maximum Marks : 100

Instructions: Answer any two questions from unit I, two questions from unit II, and one from unit I.

UNIT - I

1. (a) Briefly discuss the current trends in computer architecture towards (i) parallel processing (ii) cheap computing. (10)
(b) Describe pipeline computer structure and prove that a K-stage linear pipeline can be at most K times faster than that of a nonpipeline & serial processor. (10)
2. (a) Describe the functional structure of a SIMD array processor with concurrent scalar processing in the control unit. (10)
(b) Discuss the performance characteristic features of array processor and pipeline processor. (10)
3. (a) With an example explain the following :
(i) Loosely coupled system
(ii) Tightly coupled system

- (iii) Multiprocessor system
 - (iv) Multi computer system. (10)
- (b) Write short notes :
- (i) Pipelined multiplier using CSA.
 - (ii) Flynn's classification. (10)

UNIT - II

4. (a) What are the requisites of interconnecting network ? Explain the fundamental decisions to determine appropriate architecture of an interoonecting network. (10)
- (b) Describe multistage network by the following characterising features :
 - (i) Switch box
 - (ii) Network topology
 - (iii) Control structure (10)
5. (a) What are the salient requisites of image processing and pattern recognition ? Explain to what extent these are met by M.P.P. (10)
- (b) With neat sketch describe the architecture of neural network suitable for applications. (10)
6. (a) Describe the following :
 - (i) Data driven computation
 - (ii) Event drive computation
 - (iii) Static data flow graph
 - (iv) Dynamatic data graph. (10)
- (b) Draw data flow graph to represent the following computatons :
 - (i) If ((a = b) & (C < d)) then C = C - a
else C = C + a
 - (ii) For (i = 0 ; i \geq 10 ; i++) {Cout \rightarrow i} (10)

UNIT - III

7. (a) Define reconfigurability and explain it with reference to reconfigurable arrays and pipelines. (10)
- (b) Discuss the salient features and important applications of RISC and C and CISC architectures (10)
8. (a) Explain the working and characteristics of optical bistable device. (8)
- (b) Write short notes on :
 - (i) Systolic array architecture
 - (ii) Optical computer
 - (iii) Pattern embedding. (12)



PARALLEL ALGORITHMS (Elective)**Time : 3 Hours****Maximum Marks : 100**

Answer one question from Unit I. Answer any two questions from Unit II. Answer any two questions from Unit III. All questions carry equal marks.

UNIT - I

1. (a) What is a Supercomputer ? Explain the need for supercomputers.
(b) Define the following terms :
 - (i) Parallel processing ;
 - (ii) Speed-up ;
 - (iii) Scalability ;
 - (iv) Parallel computer ; and
 - (v) Pipelining.
2. (a) Why do you need highspeed computing ? List the points and explain in brief.
(b) List and explain the interesting features of parallel computers.

UNIT - II

3. (a) Define the terms Temporal and Data parallelism and explain how temporal parallelism can be exploited.
(b) Show that maximum speed-up of a pipeline is equal to its stages.
4. (a) Explain the following terms :
 - (i) Static schedule ;
 - (ii) Dynamic schedule ; and
 - (iii) Quasi-dynamic schedule.
(b) How does inter-task dependency affect in parallel processing ? Explain with suitable example.
5. (a) What do you understand by shared memory parallel computer ? Compute the maximum speed of processing for the following data with 10 PE's.
Processing speed of PE-1 MIPS
Data handled-64 bits.
Time to store result in memory-0.25 μsec.
Bus speed-40 Mega bytes/sec.
(b) What is a vector ? Explain vectorization process with respect to the following example :
Do 50 I = 1, 100
A (I) = B (I) * C (I)
D (I) = DULTT (A (I) * X (I)
E (I) = D (I)/B (I) + A (I)
50 CONTINUE.

UNIT - III

6. (a) What is an ideal situation for user in programming parallel computer ? Explain.
(b) Give a brief account of the salient features of FORTRAN 90.
7. (a) Explain sorting on SIMD-CC model with suitable example.
(b) Explain Ellis's algorithm for operations on AVL trees.
8. Write adequate notes on any *three* :
(a) Programming of message-passing systems.
(b) Profiling on vector computers.
(c) OCCAM for transputers.
(d) Fast Fourier transform.
(e) Parallel processing activity in the Indian context.

**B. DELHI UNIVERSITY'S (DCE)***End-term Exam.**Question Paper (2003)***MECTA****COMPUTER TECHNOLOGY AND APPLICATION****Paper – CS. 653 (Parallel Computing)****Time : 3 Hours****Maximum Marks : 100***(Write your Roll No. on the top immediately on receipt of this question paper.)*

(Question 1 is compulsory and answer any four questions from 2 to 6 Parts of the same question should be answered together and in the same sequence.)

1. (a) We know that an algorithm that satisfies the constraints of the EREW PRAM can be executed without any change on a CREW PRAM model. Is the converse true ? Justify your answer.
(b) How does a multiprocessor machine differ from a multi-computer machine ? Give two examples of each.
(c) What are the limitations of the speed up ? Can we have superlinear speed-up ? Justify your answer.
(d) Describe the place of the following computer system in the Flynn's classification scheme :
 - (i) Illiac IV
 - (ii) nCUBE
 - (iii) Cray T90 series
 - (iv) CM-5
 - (v) PARAM 10000
(e) Give the bisection width of the following Networks :
 - (i) Mesh Network

- (ii) Hypercube Network
- (iii) Shuffle-Exchange network
- (iv) De-bruijn Network

(Assume that number of nodes are - 2^k)

- (f) What do you mean by Broadcast operation and Reduction operation ? Illustrate their algorithmic implementation through example.
- (g) Give the Odd-even sorting network for sequence of eight elements.

(7 × 4 = 28)

2. (a) Give a parallel algorithm or finding the sum

$$S = \sum_{i=1}^n a(i)$$

where X = x₁, x₂, ..., x_n and Y = (y₁, y₂, ..., y_n)

Assume that you have p processors and that p divide n . explain your algorithm in the following steps.

- (i) Basic idea,
- (ii) Time complexity.

- (b) Define a n-cube or hypercube network, assuming $n = 2^m$. Give the simple routing algorihm that routes a packet from its source $S = (s_{n-1} \dots s_0)$ to destination $D = (d_{n-1} \dots d_0)$

(10 + 8 = 18)

3. (a) Describe the Cyclic Reduction algorithm to solve the first order linear recurrence.

$$x_1 = d_1$$

$$x_3 = a_j x_j + x d_1 \quad 2 < j < n$$

- (b) Suppose the best sequential algorithm for a problem requires $2n$ tog \square n steps for n data items. What is the minimum number of steps for a parallel algorithm to be cost optimal using n^2 processors ?

(12 + 6 = 18)

4. (a) Describe the parallel quicksort algorithm. What speedup can be expected from this parallel algorithm ?

- (b) What type of Fortran 90 programming model is ? Illustrate any four important features of Fortran 90 that are not found in Fortran 77.

(12 + 6 = 18)

5. (a) Given $v = 2^{22}$ and $a = (a_0, a_1, a_2, a_3, \dots, a_{n-1})^T$ be a complex vector and n processors, show that the Fourier transform of a can be computed in $O(\log(n))$ parallel steps requiring a total of $C(n \log(n))$ operations.

- (b) Describe the Graham's list scheduling algorithm on UMA multiprocessors. Does it yield an optimal scheduling ? If not give an example where it does not generate an optimal schedule.

(12 + 6 = 18)

6. (a) Write an illustrate the Hirschberg algorithm for finding the connected component of an undirected graph.

- (b) Write the pseudocode of back substitution lgorithm suitable for implementation on a UMA multiprocessor.

(12 + 6 = 18)



IGNOU'S ADCA/MCA (III YEAR)

SPECIAL TERM-END EXAMINATION, JUNE 2004

CS-12 : COMPUTER ARCHITECTURE

Time : 3 Hours

Maximum Marks : 60

Note : Question 1 is compulsory. Answer only three from the rest.

1. (a) Analyze the data dependencies among the following statements in a given program fragment :

Load R₁, M(242) / R₁ ← M(242) /Load R₂, M(240) / R₁ ← M(240) /MULTI R₃, R₁, R₂ / R₃ ← (R₁) × (R₂) /ADD R₂, R₁, R₃ / R₂ ← (R₁) × (R₃) /STORE M(240), R₂ / M(248) ← (R₂) /

Draw a dependency graph to show all the three types of dependencies and explain them.

- (b) Consider the main memory of a computer which has 128 blocks, where the size of each block is 16 words. The cache memory has 16-block-frames.

Sketch the fully associative mapping scheme. Also design the address format showing a tag field, block number and word number.

- (c) What is a hot spot problem ? Why does it occur ?

- (d) What is the cache coherence problem ? Describe two protocol approaches, with the help of a suitable diagram.

- (e) Explain the applicability and restrictions involved in using Amdahl's law to estimate the speed-up performance of n processor systems compared with that of a single processor.

- (f) Show the pipelined execution of successive instructions in two underpipelined cases, and explain each case.

2. (a) Explain the effect of the following on overall CPU performance in terms of program length, clock rate, and effective CPI.

- Instruction set
- CPU implementation and control
- Compiler technology
- cache and memory hierarchy

- (b) Compare and explain the instruction-set architecture in RISC & CISC processors in terms of instruction formats and CPI. Also write the two differences between scalar RISC and Super Scalar RISC.

3. (a) Consider the following pipeline reservation table.

	1	2	3	4
S ₁		X		
S ₂	X		X	
S ₃		X		X

- (i) What are the forbidden latencies ?
- (ii) Draw a state transition diagram.
- (iii) List all the greedy cycles.
- (iv) Determine the optimal constant latency cycle.
- (b) Design a 4-cube formed by interconnecting two 3-cubes, and also list the two features of hypercube interconnection architecture.
- 4. (a) With respect to superpipelined design :
 - (i) Show diagrammatically the superpipelined superscalar execution with degree $m = n = 3$ and explain the diagram.
 - (ii) Derive the speed-up factor over the base machine and explain each stage.
- (b) List the two key differences between multiprocessor and multicomputer systems, and explain.
- 5. (a) Draw a figure and explain the multithreaded architecture and its computation model.
- (b) Show, and explain, the two common problems caused by asynchrony and communication latency in massively parallel processing.
- (c) Consider the case of the Cray Multiprocessor model. Explain the following concepts with an example each :
 - (i) Macrotasking
 - (ii) Microtasking
- 6. (a) Define five factors which affect the performance of an interconnected network.
- (b) Describe the Hypercube Routing functions with the help of an example in each of the following cases :
 - (i) Routing by least significant bit
 - (ii) Routing by middle bit
- (c) Explain 'Inclusion' and 'Locality' properties in the context of memory hierarchy. Show them diagrammatically also.

(Please write your Roll No. immedeately)
Roll No._____

FIRST-TERM EXAMINATION

Seventh Semester [B.Tech.]
Paper code: ETCS 403
Time: 1 hr & 30mins.

Sep.:_____, 2008
Sub.:_____
Max. Marks : 30

Note: Attempt Q. No. 1 and any two more Questions.

1. (a) Distinguish between Multiprocessors and Multicomputers.
(b) Explain Feng's classification of computers with examples.
(c) State Amdahl's law. Show that the maximum speed-up achievable is -

$$S(n) = I/f \text{ Where } n \text{ is number processors.}$$

$$n \rightarrow \infty$$

- (d) Consider the following C statements -

$$\begin{aligned} a &= x + y; \\ b &= x + z; \end{aligned}$$

Do they satisfy Bernstein's conditions? Furthermore, if $a = x + y$; and $b = a + b$; then will they satisfy the same?

- (e) What are NORMA machines? (2*5 =10)
2. (a) How is Implicit parallelism different from Explicit parallelism. Give some examples and explain. (4)
- (b) Suppose that the same program is executed on two different machines-A and B. Compilers and data set on both the machines are identical. Given that:

Machine-A: Average CPI = 3.5

Cycle time = 1 .0ns

Machine-B: Average CPI = 1.2

Cycle time = 0.5ns

Which out of these two machines is slower? (6)

- (a) Consider the following statements-
`if x>y then (x-y)
else (x*y)
end if`
Data Flow Graph? (5)
- (b) 'Lower the level the finer will be the granularity of the software processes'. Explain. (5)

4. (a) Define the following networks with an example of each-

1. Blocking network.
2. Rearrangeable network.
3. Non-blocking network.
4. Recirculating network. (1*4=4)

- (b) With the help of diagrams discuss Ring, Star, Tree, Mesh, Torus and Hypercube interconnection networks with respect to node degree, network diameter and bisection width. (6)

(Please write your Roll No. immediately)
Roll No._____

**FIRST-TERM REAPPEAR EXAMINATION
FIRST-TERM EXAMINATION**

Seventh Semester [B.Tech.]

Paper code: ETCS 403

Time: 1 hr & 30mins.

Sep._____, 2008

Sub.:_____

Max.Marks:30

Note: Attempt Q. No. 1 and any two more Questions.

1. (a) Explain Flynn's classification of computers.
(b) What are Bernstein's conditions?
(c) What is an Eager machine?
(d) What are the factors affecting the performance of interconnection networks?
(e) What are systolic arrays? (2×5=10)
2. (a) Compare UMA.NUMA and COMA models.(5)
(b) How are loosely coupled systems different from tightly coupled systems? Give examples and explain.
(5)
3. (a) Explain the different levels of parallelism. (5)
(b) Discuss the role of compilers in parallel computing. (5)
4. (a) Draw a 3*3 Mesh, Illiac and Torus network. (5)
(b) What is a switch module? How can they be used to construct Multistage Networks. (5)

(Please write your Roll No. immediately)

Roll No._____

SECOND-TERM EXAMINATION**7th Semester [B.Tech.]****Paper code: ETCS 403****Time 1 hr & 30 mins.****Nov.,10,2008****Sub. : ACA****Max. Marks:30****Note Attempt Q. No. 1 and any two more Questions**

1. (a) Computer CISC and RISC processors in tabular form.
(b) Explain Sector-mapping cache addressing model.
(c) What will be the speed up gain of the vector pipeline over the scalar pipeline?
(d) Explain Ramamoorthy and Li's classification of pipeline with suitable example.
(e) What are Superscalar, Super-pipeline and Super-pipeline superscalar processors?
(*5=10)
2. (a) Prove that:
$$e = l/h + (l - h) r$$
where e , h and r represent the access efficiency, hit ratio and access time ratio respectively. Assume a two level memory hierarchy as M_1 and M_2 (5)
(b) You are asked to perform capacity planning for a two-level memory system.
 M_1 (cache) has 3 choices- 64KB, 128 KB and 256KB.
 M_2 (MM) has 4MB capacity.
Given that: $C_1 = 20 C_2$; $T_2 = 10 T_1$
Cache hit ratios are 0.7, 0.9 and 0.98 respectively for above three choices of M_1 .
(a) Find T_a in terms of $T_1 = 20\text{ns}$.
(b) Find average byte cost of entire system if $C_2 = \$0.2/\text{KB}$.
(c) Compare the three memory designs. Which out of these is the best option?
19.24, 12.16 10.15 3rd option.
(5)
3. (a) Discuss various cache performance issues. (4)
cycle cannot, hit ratio , block size set no.
(b) A block set associative cache memory consists of 128 block divided into four block sets. The main memory consists of 16,384 block and each block contains 256 eight-bit words.
(i) How many bits are required for addressing the main memory?
(ii) How many bits are needed to represent the TAG, SET, WORD fields? (6)
4. (a) Explain with a neat diagram, how will you design a pipelined, Two 6-bit Multiplier? (5)
(b) A processor (P_1) is non-pipelined and has a clock rate of 25 MHz. It has average CPI = 4. Another processor (P_2) has clock rate of 20 MHz. P_2 is designed with five stages. P_2 is improved successor of P_1 .
(i) If a program of 100 instructions is to be executed on both processor, what is the Speedup of P_2 compared to that of P_1 ?
(ii) Find MIPS rate of P_1 as well as P_2 during execution of the program. (5)

(Please write your Roll No. immediately)
Roll No._____

SECOND-TERM REAPPEAR (NEW) EXAM.

7th Semester [B.Tech.]

Paper Code: ETCS 403

Time 1hr & 30 mins.

Nov.,10,2008

Sub.: ACA

Max. Marks: 30

Note: Attempt Q. No. 1 and any two more Questions.

1. (a) Explain VLIW processor architecture.
 (b) Distinguish between superscalar and VLIW processors in tabular form.
 (c) Explain inclusion, coherence and locality properties of memory hierarchy.
 (d) Show that the maximum speed-up of a pipeline is equal to its stages.
 (e) What is the purpose of sequential, target and loop buffers? (2 *5=10)
2. (a) Consider a two-level memory hierarchy (M_1, M_2). M_1 is directly connected to the CPU. Determine the average cost per bit (C) and the average access time (t_a) for the data given below: $C = 0.01138 t = 1.09 \times 10^{-7}$

Memory level	Capacity (S_i)	Cost (C_i)	Access time (T_{av})	Hit (H)
M_1 (Cache)	1024	0.1000	10^{-8}	0.9000
M_2 (MM)	2^{16}	0.0100	10^{-6}	-----

- (b) Explain any two cache addressing models.
3. (a) What is a synchronous pipeline? Define the following with respect to this pipeline:
 (i) Clock-period.
 (ii) Speed-up.
 (iii) Efficiency.
 (iv) Throughput? (5)
- (b) Explain Handler's Classification of pipelining. (5)
4. Consider the following reservation table:

	T_1	T_2	T_3	T_4	T_5	T_6	T_7
S_1	X					X	
S_2			X				X
S_3		X		X			
S_4			X				
S_5			X		X		

- (a) Determine latencies in the forbidden list, F and the collision vector.
- (b) Draw the state transition diagram.
- (c) List all simple cycles and greedy.
- (d) Determine MAL.
- (e) If the clock period, $T = 20\text{ns}$, find the maximum throughput of the pipeline. (10)

(Please write your Roll No. immediately.....)

Roll No._____

FIRST TERM, EXAMINATION**B. Tech 7th semester, CSE****Sept._____,2009****Paper code: ETCS 403****Sub.: ACA****Time: 1 Hr & 30 mins.****M.M. : 30****Note: Q1 is compulsory and answer any 2 more Questions.**

1. (a) Discuss some interesting features of Parallel Processors.
 (b) State Bernstein's conditions.
 (c) Distinguish between Static Network and Dynamic Network.
 (d) What are systolic arrays?
 (e) How will you embed a mesh into a hypercube? (2×5=10)
2. (a) Distinguish between loosely coupled and tightly coupled systems. (4)
 (b) A program is run on a 40MHz CPU with the instruction mix and corresponding clock cycle count as given in the table below. Determine
 (a) Effective CPI
 (b) Execution time
 (c) MIPS rate for the program. (6)

The table is shown below:

Instruction type	Clock cycle count	Instruction count
1. Integer Arithmetic	1	45,000
2. Floating point	2	32,000
3. Data transfer	2	15,000
4. Control Transfer	2	8,000

3. (a) Compare control flow, dataflow and reduction computers in term of the program flow mechanism. (4)

- (b) It is desired to store the sum of all array elements, $a[i]$ for $i \leftarrow 0$ to n . that is,

$$\text{sum} = \sum_{i=1}^n a(i)$$

Draw its data flow graph. Also explain your results. (6)

4. (a) Say, two floating point numbers are to be added, Let the two numbers be a and b . Their floating point representation is as follows:

$$a = (\text{mant } a, \text{exp } a)$$

$$b = (\text{mant } b, \text{exp } b)$$

It takes T units of time to execute each task. What are the four steps that we follow to add these numbers in parallel? (4)

- (b) What is MPP? Explain its architecture in detail. What are its image processing applications? (6)

(Please write your Roll No. immediately.....)

Roll No.....

FIRST TERM, REAPPEAR EXAMINATION**B. Tech 7th semester, CSE****Sept._____,2009****Paper code: ETCS 403****Sub.: ACA****Time: 1 Hr & 30 mins.****M.M. : 30****Note: Q.1 is compulsory and answer any 2 more Questions.**

1. (a) Explain Fengls classification of computers.
(b) Define system throughput and CPU throughput. Which out of the two is greater?
(c) What is control dependence?
(d) What is fine grain, medium grain and coarse parallelism?
(e) What is the significance of bisection bandwidth? (2×5=10)
2. (a) Give some examples of India's supercomputers. (5)
(b) How does a multiprocessor machine differ from a multicomputer machine? Give two examples of each. (5)
3. (a) What is the commutativity property, non-transitivity property and associativity property of parallelism. (5)
(b) What are five main types of data dependencies? Explain each with an example. (5)
4. (a) What are multistage networks? Draw a multistage network with $a^* b$ switches. (5)
(b) Is omega network a blocking or a non-blocking network? What is the difference between a blocking and a non-blocking network? (5)

(Please write your Roll No. immediately)

Roll No._____

FIRST-TERM. (E) EXAMINATION**B. Tech 7th semester,IT****Paper code: ETCS 403****Time 1hr & 30 mins.****Sep._____,2009****Sub.: ACA(E)****M. M. :30****Note Attempt Q. No. 1 and any two more Questions.**

1. (a) Explain Handlers classification of computers.
(b) Explain Gustafson's law for scaled problem.
(c) What is a lazy machine?
(d) Define a non-blocking network.
(e) Write short notes on chip processing.
2. Consider the following C program:-

```
for (i = 0;i < i++) {  
    a[i] = b[i] + c[i];  
}  
sum = 0;  
for (j = 0;j < n;j++) {  
    sum = sum + a[j];  
}
```

This program is to be executed firstly on a sequential machine. How many machine cycles are required? Now if this program is run on a M-processor system, how many cycles are required? Compare your results. (10)

3. (a) Draw a dataflow graph showing the computations of the roots of a given quadratic equation. (5)
(b) What is grain packing? Explain. (5)
(d) Computer the maximum speed of processing for the following data with 10 PEs-
Processing speed of PE= 1MIPs, Data handled = 64 bits
Time to store results in memory = 0.25b usec.
Bus speed = 40 Megabytes/sec.? (5)
(b) Give bisection width of Mesh, hypercube and shuffle-exchange network. (5)

(Please write your Roll No. immediately.....)

Roll No._____

SECOND TERM. EXAMINATION

B. Tech 7ⁱⁿ semester, CSE

Nov.._____,2009

Paper code: ETCS 403

Sub.: ACA

Time: I Hr. & 30 mins.

M.M. : 30

Note: Q1 is compulsory and answer any 2 more Questions

1. (a) Compare RISC and CISC in a tabular form.
 (b) Compare Superscalar and VLIW processors in a tabular form.
 (c) Explain Ramamoorthy and Li's classification.
 (d) Explain the effect of block size, set number and cache size on cache performance.
 (e) Prove that the efficiency of a linear pipeline is given by the formula:

$$\eta = n/k + (n-1) (2 \times 5 - 10)$$
2. (a) Draw and explain VLIW architecture. Give its advantages, disadvantages, examples and applications.
 (b) Draw a timing diagram showing a superscalar processor with degree (m) = 3.
3. Consider a two-level memory hierarchy, M_1 and M_2 . Denote the hit ratio of M_1 as h . Let C_1 and C_2 be the cost per Kb , S_1 and S_2 be the memory capacities and t_1 and t_2 be the access times respectively.
 - (a) Under what conditions will the average cost of the entire memory system approaches C_2 .
 - (b) Find T_{eff} or t_a of this hierarchy.
 - (c) Let $r = t_2 / t_1$ be the speed ratios of two memories. Let $E = t_1 / t_a$ be the access efficiency of the memory system. Express E in terms of r and h .
 - (d) Plot E against h for $r = 5, 20, 100$ respectively.
 - (e) What is the required hit ratio (h) to make $E > 0.95$ if $r = 100$? (10)
4. (a) Discuss set associative cache addressing model. (4)
 (b) Consider the following reservation table:

S1/T1	1	2	3	4	5	6	7
S1	X		X				X
S2				X		X	
S3			X		X		

Applying Davidson's approach, find the MAL of his pipeline. Also find its throughput if the clock period is 20ns. (6)

(Please write your Roll No. immediately.....)

Roll No.....

SECOND TERM. REAPPEAR EXAMINATION**B. Tech 7th Semester, CSE****Nov.____, 2009****Paper code: ETCS 403****Sub.-ACA****Time: I Hr. & 30 mins.****M.M. : 30****Note: Attempt Q. No. 1 and any two more questions.**

1. (a) What will be the speed-up gain of the vector pipeline over scalar pipeline.
(b) How do we calculate the total cost of the memory hierarchy?
(c) State inclusion, coherence and locality of reference properties of memory hierarchy
(d) Explain sector mapping cache addressing model.
(e) What is loop unrolling? Explain with an example. (2×5 = 10)
2. (a) Say we have to multiply two numbers being stored in a 2D memory. Explain how CISC and RISC machine will solve this problem. Which is a better approach? (4)
(b) What are vector and symbolic processors? (6)
3. (a) Consider a two level memory hierarchy— M_1 and M_2 . M_1 is directly connected to the CPU. Determine the average cost per bit (C) and the average access time for the data given below:-

Memory level	Capacity	Cost	Access time	Hit
M_1 (Cache)	1024	0.1000	10^{-8}	0.9000
M_2 (MM)	2^{16}	0.0100	10^{-6}	-----

- (b) What is Daisy chained bus arbitration scheme. (5+5=10)
4. (a) Show that the throughput (w) of a pipeline is $1/x$ when efficiency tends to 1.
(b) Consider the following reservation table:

S1/T1	1	2	3	4
S1	X			X
S2		X		
S3			X	

Applying Davidson's approach, find the MAL of his pipeline. Also find its throughput if the clock period is 20ns. (6 + 4 = 10)

(Please write your Roll No. immediately.....)

Roll No.....

SECOND TERM (E) EXAMINATION

B. Tech 7th semester,IT

Nov. __,2009

Paper code: ETCS 403

Sub.: ACA(E)

Time: I Hr. & 30 mins.

M.M. : 30

Note: Q1 is compulsory and answer any 2 more Questions

1. (a) Give the advantages and disadvantages of using common/ separate caches.
(b) Explain memory hierarchy..
(c) Distinguish between symbolic and Numeric processing.
(d) What is memory interleaving?
(e) What are sequential, target and loop buffers used for? (2*5=10)
2. (a) Discuss RISC and CISC architectures. (5)
(b) What is pipeline underutilization and poor pipeline utilization? Discuss. (5)
3. (a) Discuss five parameters that characterize memory technology.
(b) Explain the role of TLB and PT during address translations. (5+5=10)
4. Draw and explain a 6-bit multiply pipeline unit. (10)

Please write your Roll No. immedeately
Roll No._____

END TERM EXAMINATION

SEVENTH SEMESTER [B.Tech] DECEMBER-2008

Paper Code: ETCS-403

Paper Id-27403

Time: 3 Hours

Subject:Advanced Computer Architecture

(Batch 2004-2005)

Maximum Marks: 75

Note : Question 1 is compulsory. Attempt one question from each unit.

1. (a) Explain how instruction set, compiler technology, CPU implementation and control, and cache and memory hierarchy affect the CPU performance and justify the effects in terms of program length, clock rate and effective CPI. (5)
- (b) Define the following terms: (5)
 - (i) Node degree
 - (ii) Network diameter
 - (iii) Bisection bandwidth
 - (iv) Static connection networks
 - (v) Dynamic connection networks.
- (c) Define the following basic terms associated with memory hierarchy design. (5)
 - (i) Virtual Address Space
 - (ii) Page fault Hit
 - (ii) Ratio
 - (iv) Multilevel Page Tables
 - (v) Hashing function
- (d) Explain physical address cache versus virtual address cache. (5)
- (e) Explain various types of vector instructions. (5)

UNIT-1

2. Consider the execution of an object code with 200,00 instruction on a 40-MHz processor. The program consists of four major types of instructions. The instruction mix land the number of cycles (CPI) needed for each instruction types are given below based on the result of a program trace experiment- (12.5)

Instruction Type	CPI	Instructions Mix.
Arithmetic and Logic	1	60%
Load/Store with cache net	2	18%
Branch	3	12%
Memory reference with cache miss	4	10%

- (a) Calculate the average CPI when the program is executed on a uniprocessor with the above trace results.
- (b) Calculate the corresponding MIPS rate based on the CPI obtained in Part (a).

OR

3. Consider the execution of, the following code segment. Detect the maximum parallelism using Bernsterns condition. Justify the portion that can be executed in parallel and the remaining portion that must be executed sequentially. (12.5)

$S1: A = B + C$
 $S2: C = D + E$
 $S3: F = G + E$
 $S4: C = A + F$
 $S5: M = G + C$
 $S6: A = L + C$
 $S7: A = E + A$

UNIT-II

4. (a) How many legitimate states are there in a 4×4 Switch module, including both broad cast and permutation? Justify your answer. (6)
- (b) Construct a 64-input Omager network using 4×4 switch modules in multiple stages. How many permutations can be implemented directly in a single pass through the network without blocking? (6.5)
5. Explain the structure and operational requirements of the instruction pipelines used in CICS, Scalar, RISC, Superscalar RISC, and VLIW processors. Comment on the cycles per instruction expected from these processor architectures. (12.5)

UNIT-III

6. Discuss synchronous Vs Asynchronous Bus timing protocol. (12.5)
OR
7. Consider the following reservation table for a four-stage pipeline, with a clock cycle $T = 20$ ns. (12.5)

	1	2	3	4	5	6
S1	X					X
S2		X		X		
S3			X			
S4				X	X	

- (a) What are the forbidden latencies and the initial collision Vector?
(b) Draw the state transition diagram for scheduling the pipeline.
(c) Determine the MAL associated with the shortest greedy cycle.
(d) Determine the pipeline throughput corresponding to the MAL and given above.

UNIT-IV

8. Discuss S-Access memory organization. (12.5)
OR
9. Discuss organization of SIMD computer. (12.5)

Please write your Roll No. immediately

Roll No. _____

END TERM EXAMINATION**SEVENTH SEMESTER [B.Tech] DECEMBER-2009****Paper Code: ETCS-403****Paper Id-27403****Time: 3 Hours****Subject: Advanced Computer Architecture****Maximum Marks: 75**

Note : Question 1 is compulsory. Attempt one question from each unit.

1. (a) What are the different methods of increasing the speed of computers? (4)
(b) Characterize the architectural operations of SIMD and MIMD computers. (4)
(c) Explain Cache addressing models. (4)
(d) Explain superscalar technique. (4)
(e) Discuss vector processing principle. (4)
(f) Define coherence and locality.(5)

UNIT-I

2. Perform a data dependencies analysis on each of the following Fortran program fragments. Show the dependence graphs among the statement with justification:- (12.5)

S1: $X = \text{SIN}(Y)$

S2 : $Z = X + W$

S3 : $-2.5 \times W$

S4 : $X = \text{COS}(Z)$

OR

3. (a) Perform a data dependences analysis on each of the following Fortran program fragments. Show the dependence graphs among the statements with justification: (8)

S1 : $A = B + D$

S2 : $C = A \times 3$

S3 : $A = A + C$

S4 : $E = A/2$

- (b) What are the differences between string reduction and graph reduction machines? (4.5)

UNIT-II

4. (a) What is the difference between a blocking switch and a non-blocking switch? Is an Omega Network block or non-blocking? (6)
(b) Suppose parallel issue is added to vector pipeline execution. What would be the further improvement in throughput, compared with parallel issue in a superscalar pipeline of the same degree? (6.5)

OR

5. Prove the following properties associated with multistage Omega Networks using different-size building blocks: - (12.5)

- (a) Prove that the number of legitimate states in a $K \times k$ switch modules equals K^k .

- (b) Determine the percentage of permutations that can be realized in one pass through a 64-input Omega built with 2×2 switch modules.
- (c) Repeat part (b) for 64-input Omega Network built with 8×6 switch modules.

UNIT-III

6. Discuss Synchronous vs Asynchronous Bus timing protocol. (12.5)

OR

7. A non pipelines processor X has a clock rate of 25 MHz and an average CPI of 4. Processor Y, an improved successor of X, is designed with a five-stage linear instruction pipeline. However, due to latch delay and clock skew effects, the clock rate of Y is only 20 MHz. If a program containing 100 instructions is executed on both processors, what is the speedup of processor Y compared with that of processors X? Calculate the MIPS rate of each processor during the execution of this particular program. (12.5)

UNIT-IV

8. Write short note: (any two) _____ (12.5)
(a) Enterprise memory subsystem architecture
(b) VLIW architecture
(c) Vector and symbolic processor

OR

9. Discuss S-Access memory organization. (12.5)

END TERM EXAMINATION

First Semester [B.Tech.] December 2007

Paper Code: IR605

Paper ID: 53605

Time: 3 Hours

Subject: Advanced Computer Architecture

Maximum Marks: 60

Note: Attempt any five questions in all including Q.1 which is compulsory

Q.1. Explain in brief:- **(2 × 10 = 20)**

- (a) What do you understand by the granularity of a parallel system ?
- (b) Define Flynn's classification.
- (c) What are the advantages of parallel processing over sequential computations ?
- (d) Differentiate between UMA, NUMA and COMA.
- (e) What is the significance of a bisection bandwidth ?
- (f) What are the differences between scalar and vector processing ?
- (g) If a super scalar processor of degree '3' is used in 4-stage pipeline instructions, then how many instructions will be executed in '7' clock cycles ?
- (h) List '3' data structures used for parallel algorithms.
- (i) Enumerate at least five applications of parallel programming.
- (j) Differentiate between CISC and RISC scalar processor.

Q.2. (a) Explain Bernstein's conditions and how is it useful to determine the maximum parallelism? **(5)**

(b) Determine the maximum parallelism between the following instructions:- **(5)**

S₁; X = Y + Z

S₂; Z = U + V

S₃; R = S + V

S₄; Z = X + R

S₅; Q = M + Z

Q.3. Consider the execution of an object code with 2,00,000 instructions on a 40 MHz processor. The program consists of four major types of instructions. The instructions mix and the number of cycles (CPI) needed for each instruction type are given below on the result of a program trace experiment: **(10)**

<i>Instruction type</i>	<i>CPI</i>	<i>Instr. Mix</i>
Arithmetic	1	60%
Load/Store	2	18%
Branch	4	12%
Memory reference	8	10%

(a) Calculate average CPI for uniprocessor.

(b) Calculate MIPS.

Q.4. (a) Discuss the differences between control flow, data flow and reduction computers. **(6)**

(b) Define the following terms 3 examples (i) Node degree (ii) Network diameter. **(4)**

Q.5. (a) Discuss the Design Space of modern processor facilities in detail. (6)

(b) How scalar RISC and Super Scalar RISC are different ? (4)

Q.6. (a) Define the following:- (2 x 3 = 6)

(i) Inclusion

(ii) Coherence

(iii) Locality of reference

(b) Virtual memory models. (4)

Q.7. Consider a two-level memory hierarchy M_1 and M_2 , with access time t_1 and t_2 cost per byte C_1 and C_2 and capacities S_1 and S_2 . The Cache hit ratio $h_1 = 0.95$ at first level.

(a) Derive a formula showing the effective access time t of this memory system. (5)

(b) Derive a formula showing the total cost of this memory system. (5)

Q.8. Write short notes on any two: (2 x 5 = 10)

(a) Non linear pipeline processor

(b) High performance signaling layer

(c) Patrol scrobbing.

END TERM EXAMINATION

First Semester [M.Tech.] December 2008

Paper Code: ITR605

Subject: Advanced Computer Architecture

Paper ID: 53605

Time: 3 Hours

Maximum Marks: 60

Note: Attempt five questions in all including Q.1 which is compulsory

Q.1. Explain in brief:- (2 × 10 = 20)

- (a) Define Node degree and Network diameter.
- (b) What do you understand by 'Page fault' ?
- (c) Define efficiency and throughout of a pipelined processor.
- (d) List the five generations of Electronic computers with their Representative Systems.
- (e) Explain MIMD and SISD machines with the help of example.
- (f) Differentiate between control flow and data flow computers.
- (g) Explain perfect shuffle permutation function.
- (h) What do you understand by 'Instruction pipeline cycle' and 'Instruction issue latency' ?
- (i) List the differences between RISC and CISC Architecture.
- (j) Differentiate between Super-scalar and Vector processor.

Q.2. Perform a data dependence analysis on $S_1: A = B + D$ $S_2: A = B + D$ $S_3: A = B + D$ $S_4: A = B + D$

Show the dependence graphs among the statements with justification. (10)

Q.3. Explain, how instruction set, compiler technology, CPU implementation and control, and cache and memory hierarchy affect the CPU performance. Justify the effects in terms of program length clock rate and effective CPI. (10)**Q.4.** Explain the following or a pipelined processor:- (2.5 × 4 = 10)

- (a) Internal Data forwarding
- (b) Prefetch buffers
- (c) Hazard avoidance
- (d) Multiple functional units

Q.5. (a) Design a pipeline unit for fixed point multiplication of 8-bit integers. (5)

(b) Discuss effect of Branching for a pipeline processor. Also compute the effective pipeline throughput with the influence of branching. (5)

Q.6. Answer the following questions for the K-ary n-cube network. (2 × 5 = 10)

- (a) How many nodes are there ?
- (b) What is the network diameter ?
- (c) What is the bisection bandwidth ?
- (d) What is the node degree ?
- (e) Draw a K-ary n-cube network.

Q.7. Consider a two level memory hierarchy M_1 and M_2 . M_1 is a cache with three capacity choices of 64KB, 128KB and 256KB. M_2 is a main memory with a 4MB capacity. Let C_1 and C_2 be the Cost/Byte and t_1 and t_2 the access times for M_1 and M_2 respectively. Assume $C_1 = 20C_2$ and $t_2 = 10t_1$. The cache hit ratio for the three capacities are assumed to be 0.7, 0.9 and 0.98 respectively.

- (a) What is the average access time t_g in terms of $t_1 = 20\text{ns}$ in the three cache design. (5)

- (b) Express the average byte cost of the entire memory hierarchy if $C_2 = \$0.2/\text{KB}$. (5)

Q.8. Write short notes on any two: (2 x 5 = 10)

- (a) Cache Coherence Protocols.

- (b) Enterprise Memory Subsystem Architecture.

- (c) VLIW Architecture

SECOND TERM EXAMINATION

Seventh Semester [B.Tech.] November 2011

Paper Code: ETCS403

Paper ID: 44201

Time: 1½ Hours

Subject: Advanced Computer Architecture

Maximum Marks: 30

Note: Q.1 is compulsory and answer any two questions from the rest.

Q.1. (a) Define write through and write back policy. $(2 \times 5 = 10)$

- (b) What is broad call ?
- (c) Define page fault rate.
- (d) What is CSA ?
- (e) What is simple latency ?

Q.2. (a) Discuss cache coherence. (4)

- (b) Consider the design of a three-level memory hierarchy with the following Specifications:

Memory Level	Access Time
Cache	$t_1 = 30\text{ns}$
Main memory	$t_2 = ?$
Second Memory	$t_3 = 2\text{ms}$

The design goal is to achieve an effective memory-access time $t = 800\text{ns}$ with a cache hit ratio $h_1 = 0.97$ and a hit ratio $h_2 = 0.99$ in main memory. Find the main memory access time. (6)

Q.3. Consider the following reservation table for a four-stage pipeline.

	1	2	3	4	5	6	7
S1	x		x				x
S2				x		x	
S3			x		x		

- (a) What are the forbidden latencies ?
- (b) Draw a state transition diagram showing all possible initial sequences (cycles) without causing a collision in the pipeline.
- (c) List all the simple and greedy cycles from the state diagram.
- (d) What is the MAL for this pipeline ?
- (e) If pipeline clock period is 20ns. Find the throughput. (8)

Q.4. (a) 45 tasks are processed in a 20-stage pipeline. The pipeline frequency is 1000MHz. (6)

- (i) Find the Speedup factor ?
- (ii) Find the efficiency.
- (iii) Find the throughput.

(b) Design a pipeline for floating point adder. (4)

SECOND TERM EXAMINATION

First Semester [M.Tech.] January 2011

Paper Code: ITR605

Time: 1½ Hours

Subject: *Advanced Computer Architecture*

Maximum Marks: 60

Note: Attempt all questions. Internal choice indicated.

- Q.1.** Define or briefly describe any ten from the following:- $(2 \times 10 = 20)$

- (a) Network diameter and Bisection bandwidth.
- (b) Computational granularity and communication latency.
- (c) Unicast, multicast and broadcast interconnect.
- (d) RAW, WAR and WAW hazards.
- (e) Perfect Shuffle mapping.
- (f) Write through and write-back cache.
- (g) Write-invalid, write-update and write-once protocols.
- (h) Message, packet and flit in message packing mechanisms.
- (i) Memory bandwidth and fault tolerance.
- (j) Cache flushing policies: FIFO and LRU
- (k) Gather and Scatter instructions related to vector processing.
- (l) Processor and coprocessor.
- (m) Cache hit mapping.

- Q.2.** Attempt either (a) or (b) of the following:

- (a) (i) Briefly explain the following terms related to processing on a digital computer:
Cycle time, cycles per instruction (CPI), Instruction count (I_C), Execution time,
MIPS rate and Throughput rate. (5)
- (ii) A 100MHz processor was used to execute a benchmark program with the
following instruction mix and clock cycle counts:

Instruction type	Instruction count	Clock cycle count
Integer arithmetic	48000	1
Data transfer	300000	2
Floating point	12000	3
Control transfer	10000	2

Determine the effective CPI, MIPS rate and execution time for this program. (5)

- (b) Sequential program consists of the following five statements. Considering each statement as a separate process clearly identify input set I_i and output set O_i of each process. Restructure the program using Bernstein's Conditions in order to achieve maximum parallelism between processes. If any pair of the process cannot be executed concurrently, specify which of the three conditions is not satisfied. (10)

S1: $A = B \times C$

S2: $C = B + D$

S3: $S = O$

S4: Do $I = A, 100$

$S = S + x(I)$

End Do

$$S5: IF(S.GT.1000)C = C \times 2$$

Q.3. Attempt either (a) or (b) of the following:

- (a) (i) Sketch an 8-input Omega Network using 2×2 switches as building blocks and show switch settings for messages from node 0 to node 6 and from node 4 to node 7. Indicate where blocking is taking place. How many permutations can be implemented in one pass through the above network ? $(4 + 1 + 1)$
- (ii) On a 16 node hypercube network sketch the multicast routes with minimum distance from the source the multicast routes with minimum distance from the source node (001) to seven destinations (0001, 0111, 1000, 1011, 1100 and 1111) (4)
- (b) (i) Compare the instruction set architecture in RISC and CISC processors in terms of instruction formats, addressing modes and CPI. (5)
- (ii) Explain the difference between superscalar and VLIW architecture in terms of hardware and software requirements. (5)

Q.4. Attempt either (a) or (b) of the following:

- (a) (i) How are bits of memory address mapped to different parts of cache, which is organized either as Directly-Mapped, Fully Associate or Set-Associate Cache ? (3)
- (ii) State the formula for obtaining average access time in a memory hierarchy.(1)
The access time of the cache memory is 5ns and hit rate is 75%. The access time of the main memory is 120ns and hit rate is 99.5%. The access time of virtual memory is 10ms. What is the average access time of this memory hierarchy ?(4)
- (iii) The computer system has a 64 bit virtual memory address and 42 bit physical address. Find out virtual page number and physical page number if the page size is 16KB. (2)
- (b) (i) Consider the following four stage reservation table:

	1	2	3	4	5
S1	x				x
S2		x			
S3			x		
S4				x	

State forbidden latencies and collision vector sketch the stage transition diagram. List simple and greedy cycles. Determine optimal constant cycle and minimum average latency (MAL) $(1 + 3 + 2 + 1)$

- (ii) Sketch a three stage linear pipeline and state the formulae for speed processing n tasks. (3)

Q.5. Attempt either (a) or (b) of the following:

- (a) Write short notes on any two of the following:- $(2 \times 5 = 10)$
 - (i) MESI protocol
 - (ii) Cloud computing
 - (iii) Vector access memory organizations: C-Access, S-Access and C/S-Access.
- (b) (i) In the context of instruction pipeline explain briefly the three methods for scheduling instructions: static scheduling, dynamic scheduling with Tomasulo's register tagging scheme and score-boarding. (5)
- (ii) Design and sketch a pipeline for fixed point multiplication of 12 bit integers, with first stage for partial product generation, the last stage with 24 bit look ahead adder and intermediate stages with carry-save adders. The sketch should show all line-widths and inter stage connections.

END TERM EXAMINATION

First Semester [M.Tech. (CSE/IT)] December 2011

Paper Code: ITR605

Subject: *Advanced Computer Architecture*

Time: 3 Hours

Maximum Marks: 60**Note:** Attempt any five questions including Q.1 which is compulsory.**Q.1.** (a) Define or discuss any 10 of the following in the context of computer architecture. (2×10=20)

- (i) SIMD model and MIMD model
- (ii) Data dependences
- (iii) Speed up gain of a multiprocessor machine
- (iv) Superscalar and VLIW Architecture
- (v) Static and Dynamic connection networks
- (vi) Pipeline Efficiency and Throughput
- (vii) Perfect Shuffle and Exchange function
- (viii) Hardwired and Micro-coded control
- (ix) Instruction issue latency and Instruction issue rate
- (x) Register-to-register architectures for multivector supercomputers
- (xi) Non-blocking and Blocking networks
- (xii) Node degree and network diameter
- (xiii) What is the number of 2×2 switches in a 16×16 baseline multistage network ?
- (xiv) MESI protocol

Q.2. (a) Discuss the concept of computational granularity and communication latency. Further, discuss how the concepts are related at various levels, e.g., instruction level, loop level, procedure level etc. (4)

(b) Consider the execution of a program of 15,000 instructions by a linear pipeline processor with a clock rate of 25 MHz. Assume that the instruction pipeline has five stages and that one instruction is issued per clock cycle. The penalties due to branch instructions and out-of sequence executions are ignored. (3 + 3)

- (i) Calculate the speedup factor using this pipeline to execute the program as compared with the use of an equivalent non-pipelined processor with an equal amount of flow-through delay.

- (ii) What are the efficiency and throughput of this pipelined processor ?

Q.3. (a) Construct an 8-input Omega network using 2×2 switch modules in multiple stages. Show the routing of the message from input 010 to output 110.

(b) A nonpipelined processor X has a clock rate of 25 MHz and an average CPI of 4. Processor Y, an improved successor of X, is designed with a five-stage linear instruction pipeline. However, due to the latch delay and clock skew effects, the clock rate of Y is only 20MHz. (3 + 3)

- (i) If a program containing 100 instructions is executed on both processors, what is the speedup of processor Y compared with that of processor X ?

- (ii) Calculate the MIPS rate of each processor during the execution of this particular program.

Q.4. (a) Compare control-flow, dataflow & reduction computers in terms of the program flow mechanism used.(b) State and use Bernstein's conditions to detect maximum parallelism embedded in the following code. Justify the portions that can be executed in parallel. (1 + 5 + 1)
P1: $X = Y * Z$

$$P2: P = Q + X$$

$$P3: R = T + X$$

$$P4: X = S + P$$

$$P5: V = Q / Z$$

Q.5. (a) Compute the diameter and bisection width for 3D mesh with $p = 64$ processors. (2)

(b) Consider the **five-staged pipelined** processor specified by the following reservation table: (8)

	1	2	3	4	5	6
S1	x					x
S2		x			x	
S3			x			
S4				x		
S5		x				x

(i) List the set of forbidden latencies and the collision vector. (1)

(ii) Draw the state transition diagram showing all possible initial sequence (Cycles) without causing a collision in the pipeline. (1.5)

(iii) List all the simple cycles from the state diagram. (1.5)

(iv) Identify the greedy cycles from the state diagram. (1)

(v) What is the MAL of this pipeline ? (0.5)

(vi) What is the minimum allowed constant cycle in using this pipeline ? (0.5)

(vii) What will be the maximum throughput of this pipeline ? (1)

(viii) What will be the maximum throughput if the minimum constant cycle is used ? (1)

Q.6. (a) Analyze the data dependences among the following statements in a program and draw a dependence graph to show all the dependences. Also indicate resource dependence if any. Rewrite the statements after eliminating false (pseudo) dependencies if any. (4)

$$S1: Y = X / C$$

$$S2: X = X / C$$

$$S3: Z = Y / C$$

$$S4: Y = C - Y$$

(b) Consider a cache (M_1) and memory (M_2) hierarchy with the following characteristics: M_1 : 16 K words, 50 ns access time, M_2 : 1 M words, 400 ns access time. Assume eight-word cache blocks and a set size of 256 words with **set-associative** mapping. (2 + 2 + 2)

(i) Show the mapping between M_2 and M_1 .

(ii) How is the memory address divided into various fields ?

(iii) Calculate the affective memory access time with a cache hit ratio of $h = 0.95$.

Q.7. (a) Discuss the architecture of SPARC processor. Explain the concept of overlapped register windows. (4)

(b) A two-level memory system has eight virtual pages on a disk to be mapped into three pages frames (FPs) in the main memory. A certain program generated the following page trace: 0, 1, 2, 4, 2, 3, 7, 2, 1, 3, 1. (4.5 + 1.5)

(i) Show the successive virtual pages residing in the three page frames with respect to the above page trace using the LRU, FIFO and Optimal replacement policy. Compute the hit ratio in the main memory. Assume the PFs are initially empty.

(ii) Compare the hit ratio in parts (i) and comment on the effectiveness of the

three policies with respect to this particular page trace.

END TERM EXAMINATION

Seventh Semester [B.Tech. December 2011]

Paper Code: ETCS403

Time: 3 Hours

Subject: *Advanced Computer Architecture*

Maximum Marks: 75

Note: Attempt any five questions

- Q.1. (a) Define a trace. How is it used in a VLIW processor? List the advantages and disadvantages of VLIW processor. (7.5)
(b) What do you understand by the term 'branch prediction'? Why do many superscalar processors include hardware to perform branch prediction? (7.5)
- Q.2. (a) Draw a 16 input omega network using 2×2 switches as building blocks. Show the switch setting for routing a message from node 1011 to node 0101 and from node 0111 to node 1001 simultaneously. Does the blocking exist in this case? (7.5)
(b) Describe a multiport memory without priority assignment. (7.5)
- Q.3. (a) Differentiate Hardware and Software parallelism. (5)
(b) How many switch points are there in a crossbar switch network that connects processors in memory modules? (5)
(c) Draw the space time diagram for a six segment pipeline showing the time it takes to process eight tasks. (5)
- Q.4. (a) A non pipeline system takes 50ns. to process a task. The same task can be processed in a six- segment pipeline with a clock cycle of 10ns. Determine the speedup ratio of the pipeline for 100 tasks. What is the maximum speedup that can be achieved? (7.5)
(b) Explain different types of addressing modes with suitable examples. (7.5)
- Q.5. (a) Describe associative caches. How they function? (6)
(b) What are the difference between super scalar processors and multiscalar processors and multiscalar processors? (5)
(c) What conditions must be satisfied by the statement appearing before a branch instruction so that it can be used in the delay slot? (4)
- Q.6. (a) Describe how Cache Coherence is maintained in Distributed Shared Memory parallel computers? (8)
(b) Explain handshake protocol. How it controls data transfer during an input operation? (7)
- Q.7. (a) Draw and explain MIMD architecture. How it is different from MISD architecture? (7.5)
(b) Describe various vector-access memory schemes. (7.5)
- Q.8. Write short notes on any three of the following: (5 \times 3 = 15)
(a) RISC Scalar Processors
(b) Program flow mechanisms
(c) Dynamic instruction scheduling

(d) Vector instruction types.

FIRST TERM EXAMINATION, SEPT., 2012

ACA-(403)

Time : 1 Hours

Maximum Marks : 30

Note : Question 1 is compulsory. Answer only two from the remaining.

- Distinguish between implicit and explicit parallelism with examples.
 - What is grain packing?
 - Draw a data flow graph to compute the following expression:-

$$(x + y) * (x - y)$$
. What happens when $x = 4$ and $y = 32$ appears on the input arcs x and y .
 - Explain the hypercube routing functions with example.
 - What is a systolic array? List its primary characteristics. $(2 \times 5 = 10)$
 - Explain how the degree of parallelism and number of processors affect the performance of a parallel computing system. Give Amdahl's law and find the expression for fixed load speed-up. (10)
 - Consider the execution of the following code segment consisting of 7 statements. Use Bernstein's conditions to detect the maximum parallelism embedded in this code. justify the portions that can be executed in parallel and the remaining portions that must be executed sequentially. Rewrite the code using parallel constructs such as C-begin and co-end. No variable substitution is allowed.

$$\begin{array}{lll} S1 : A = B + C & S2 : C = D + E & S3 : F = G + E \\ S4 : C = A + F & S5 : M = G + C & S6 : A = L + C \\ S7 : A = E + A & & \end{array}$$

$$S1 : A = B + C \quad S2 : C = D + E \quad S3 : F = G + E$$

$$S4 : C = A + F \quad S5 : M = G + C \quad S6 : A = L + C$$

$$S7 : A = E + A \quad (10)$$
 - With the help of diagrams, discuss chordal ring, barrel shifter, mesh and torus interconnection networks with respect to node degree, network diameter and bisection width.

- (b) What is MPP? Explain its architecture in detail. What are their images processing applications? (5 + 5 = 10)

END TERM EXAMINATION

Seventh Semester [B.Tech. December 2012]

Paper Code: ETCS403

Subject: Advanced Computer Architecture

Time: 3 Hours

Maximum Marks: 75

Note: Attempt any five questions.

- Q.1. (a) Explain the UMA, COMA & NUMA multiprocessor models. (7.5)
(b) Differentiate between SISD, SIMD, MIMD and MISD architectures. (7.5)
- Q.2. (a) What are the sysmtes attributes to performance of a computer system? (7.5)
(b) Explain five types of Data Dependence (7.5)
- Q.3. Explain Bernstein's conditions for parallelism. Consider the execution of the following code segment consisting of seven statements. Use Bernstein's conditions to detect the maximum parallelism embedded in this code. Justify the portions that can be executed in parallel and the remaining portions that must be executed sequentially. (5)
- P1 : A = B + C
P2 : C = D + E
P3 : F = G + E
P4 : C = A + F
P5 : M = G + C
P6 : A = L + C
P7 : A = E + A
- Q.4. (a) Compare CISC and RISC Architecture. (7.5)
(b) Explain static and dynamic interconnection networks. (7.5)
- Q.5. (a) What is multistage and combining network? (7.5)
(b) Explain Inclusion, coherence and locality in memory technology. (7.5)
- Q.6. (a) Discuss some cache addressing models. (7.5)
(b) What is virtual memory management? Explain some page replacement algorithms. (7.5)
- Q.7. Differentiate asynchronous with synchronous pipeline model. Give formula for calculating speedup, efficiency, throughput, optimal no. of stages for synchronous model. (15)
- Q.8. Write short notes on any three of the following:- (3 × 5 = 15)
(a) Vector processing and instruction types
(b) SIMD Parallel algorithms
(c) Crossbar and multiport memory
(d) VLIW architectures

D --- **APPENDIX**

EXPERIMENTS

Suggested experiments of ACA-lab / Parallel Programming lab. With solutions

Experiment No. 1

Write a program to show a demo of forkall and nodeid constructs.

Solution.

```
/* Demo of forkall and Nodeid */
#include< parc 2.h>
void main ()
{
    inf i;
    forkall ();
    i = nodeid ();
    printf ("\n This is processor number %d", i);
    if (i) terminate ();
}
```

Experiment No. 2

Write a program to add two matrices using fork () and nodeid ().

Solution.

```
/* addition of 3 * m matrices */
#include< parc 2.h>
main ()
{
    inf i ; nid, j, n, np a[3][3], b[3][3], c[3][3];
printf ("\n Enter number of columns of matrix a & b\n");
scanf ("%d", &n);
printf ("\n Matrix a\n");
for (i = 0 ; i < 3 ; ++i)
{
    for (j = 0 ; j < n ; ++j)
    {
        scanf ("%d", &a[i][j]);
    }
}
printf ("\n Matrix B\n");
```

```
for (i = 0 ; i < 3 ; ++i)
{
    for (j = 0 ; j < n ; ++j)
    {
        scanf ("%d ; & b[i][j]) ;
    }
}
fork (1);
fork (2);
nid = nodeid ();
if (! nid)
{
    for (j = 0 ; j < n ; ++j)
        c[0][j] = a[0][j] + b[0][j];
}
if (nid)
{
    for (j = 0 ; j < n ; ++j)
        c[nid][j] = a[nid][j] + b[nid][j];
}
if (! nid)
for (i = 1 ; i < 3 ; i++)
    formany 2((char far *) c[i], (n * 2), i);
if (! nid)
{
    printf ("\n Matrix C\n");
    for(i = 0 ; i <= 2 ; ++i)
    {
        printf ("\n");
        for (j = 0 ; j < n ; ++j)
            printf ("%d"; c[i][j]);
    }
}
terminate ();
}
```

Experiment No. 3

Write a program to multiply two matrices.

Solution. A parallel program for matrix multiplication is given below :

```
forkall ();
clrscr ();
nid = nodeid ();
np = numproc ();
```

```

/* compute block size for all PEs */
blk = N/np ;
bsize = blk * size of (int) * N ;
K1 = nid * blk ;
/* compute row wise */
if (nid == (np - 1))
    M = N ;
else
    M = (nid + 1) * blk ;
for (l = k1 ; l < M ; l++)
    for (i = 0 ; i < N ; i++)
    {
        for (j = 0 ; j < N ; j++)
        {
            C[i][j] += A[l][j] * B[j][i] ;
        }
        printf ("C[%d][%d] = %d", l, i, C[l][i] ;
    }
    if (np > 1)
        fpr (index = 0 ; index < np ; index++)
fromany ((char *) & C[index * blk] [0], bize, index) ;

```

Experiment No. 4

Write a program to solve a system of linear equations given, using Gaussian Elimination technique.

Solution. Consider a system of n-linear equations containing m-variable :

$$\left. \begin{array}{l} a_{11} x_1 + a_{12} x_2 + \dots + a_{1m} x_m = b_1 \\ a_{21} x_1 + a_{22} x_2 + \dots + a_{2m} x_m = b_2 \\ \vdots \\ a_{n1} x_1 + a_{n2} x_2 + \dots + a_{nm} x_m = b_n \end{array} \right\}$$

So, it's parallel code is :

```

/* upper triangularization */
forkall () ;
st-tm = timetick () ;
nid = nodeid () ;
if (!nid)
{
    for (col = 0 ; col <= m ; ++ col)
        u[0][col] = a[0][col] ;
}
for (k = 0 ; k < n - 1 ; ++k)
{
    if (nid > k)

```

```

{
for (col = 0 ; col <= m ; ++ col)
{
    if (a[k][k] != 0)
        u[nid][col] = a[nid][col] -
            (a[nid][k] * a[k][col]) / a[k][k];
}
for (i = 1 ; i <= n ; ++ i)
    fromany ((char far *) u[i], (4*(m + 1)), i);
    for (i = k + 1 ; i < n ; i++)
    {
        for (j = 0 ; j < m ; j++)
        {
            a[i][j] = u[i][j];
        }
    }
/* back tracking */
x[n-1] = a[n-1][m] / a[n-1][n-1];
for (k = n - 2 ; k >= 0 ; -- k)
{
    sum = 0.0;
    for (i = n - 1 ; i >= k ; -- i)
        sum += u[k][i] * x[i];
    x[k] = [u[k][m] - sum] / u[k][k];
}

```

Experiment No. 5

Write a program to implement simpson's 1/3rd rule.

Solution. Simpson's 1/3rd rule is stated as follows :

$$\int_{x_0}^{x_n} y dx = \frac{h}{3} (y_0 + 4(y_1 + y_3 + \dots + y_{n-1}) + 2(y_2 + y_4 + \dots + y_{n-2}) + y_n)$$

It's program is :

```

forkall ();
nid = nodeid ();
np = numproc ();
h = (u - 1) / (10.0 * np);      /* stepwise */
if (!nid)
{
    for(i = 0 ; i < 10 ; ++i)

```

```
        ysm = f ((1 + (i * h))) ;
    }
    if (nid)
    {
        for (i = 0 ; i < 10 ; i++)
            ysm = f(nid * (h * i)) ;
        if ((nid * h) != 1 && (nid * h) != 4)
        {
            if ((nid % 2) == 0)
                ysm *= 2 ;
            if ((nid % 2) != 0)
                ysm *= 4 ;
        }
    }
    gathreal (ysm, & sum) ;
    if (!nid)
    {
        sum = (h/3) * sum ;
        printf ("\n value of integral = %f", sum) ;
    }
    else
        terminate () ;
}
```

Note : Parallelism is achieved by first dividing the total interval into an even number of subintervals, then each processor is made to calculate the value of the function in a particular subinterval. The value of the function in all the subintervals is then gathered and summed up in processor zero (or PE-0).

E————**APPENDIX**

D. BIBLIOGRAPHY

1. Kai Hwang, "Advanced Computer Architecture", TMH, Edition 2001, Reprint 2007.
2. S.S. Jadhav, "Advanced Computer Architecture and Computing", Technical Publications, Edition 2008.
3. V. Rajaraman and C.S.R. Murthy, "Parallel Computers", PHI, Edition 2000.
4. H.G. Cragon, "Memory Systems and Pipelined Processors", Narosa Publishing, Edition 1996, Reprint 1998.
5. Arvind and Culler, D.E., Data flow Architectures in Annual Review of computer Science, Vol. 1, pp 225–53, Editor Traub, J.F. ; Annual Reviews Inc., California, USA.
6. Barry Wilkinson, "Parallel Programming", Pearson Education Asia, Edition 1999, Reprint 2002.
7. Dan Hammerstrom, 'Neural Networks at work' IEEE Spectrum June 1993.
8. **Websites :**
 - (a) <http://www.ibm.crm/servers/eserver/pseries/hardware/whitepapers/power/ppc.arc.2.html>
 - (b) <http://www.intel.com/products/processor>.
9. William Stallings, "Computer Organisation and Architecture", PHI 2012.
10. D.E. Culler, J. Singh, "Parallel Computer Architecture", Morgan Kaufmann Publishers.

F = APPENDIX

E. LATEST DEVELOPMENT

Supercomp sets new speed record

US SCIENTISTS unveiled the world's fastest supercomputer on 9/6/08, as \$100 million(Rs. 420 crore) machine that for the first time has performed 1,000 trillion calculations per second in a sustained exercise.

The technology breakthrough was accomplished by engineers from the Los Alamos National Laboratory and the IBM Corp. on a computer to be used primarily on nuclear weapons work, including simulating nuclear explosions.

The computer, named Roadrunner, is twice as fast as IBM's Blue Gene system at Lawrence Livermore National Laboratory, which itself is three times faster than any of the world's other supercomputers, according to IBM.



Roadrunner will help maintain the United States nuclear weapons stockpile.

"The computer is a speed demon. It will allow us to solve tremendous problems" said Thomas D'Agostino, head of the National Nuclear Security Administration which oversees nuclear weapons research and maintains the warhead stockpile.

But officials said the computer also could have a wide range of other applications in civilian engineering, medicine and science, from developing bio-fuels and designing more fuel-efficient cars to finding drug therapies and providing services to the financial industry. To put the computer's speed in perspective, if every one of the six billion people on earth used a hand-held computer and worked 24 hours a day it would take them 46 years to do what the Roadrunner computer can do in a single day.

The IBM and Los Alamos engineers worked six years on the computer technology. Some elements of the Roadrunner can be traced back to popular video games, said David Turek, vice-president of IBM's supercomputing programmes. In some ways, he said, it's "a very souped-up Sonyplay Station 3".

"We took the basic chip design (of a Playstation) and advanced its capability," said Turek.

But the Roadrunner supercomputer is nothing like a video game. The interconnecting system occupies 6,000 sq ft with 91.7 km of fibre optic and weighs 226,800 kg. Although made from commercial parts, the computer consists of 6,948 dual-core computer chips and 12,960 cell engines, and it has 80 terabytes of memory.

The Roadrunner computer, now housed at the IBM research laboratory in Poughkeepsie, New York, will be moved next month to the Los Alamos National Laboratory in New Mexico.

Along with other supercomputers, it will be key "to assure the safety and security of our (weapons) stockpile," said D. Agostino. With its extraordinary speed it will be able to simulate the performances of a warhead and help weapons scientists track warhead aging, he said.

But the computer – and more so that technology that it represents – marks a future for a wide range of other research and uses. "The technology will be pronounced in its employment across industry in the years to come," said Turek, the IBM executive.

Michael Anastasio, director of the Los Alamos lab, said that for the first six months the computer will be used in unclassified work. After that about three-fourths of the work will involve weapons and other classified government activities.

—AP

G APPENDIX EUROPE GET ITS FASTEST PC



Jugene — Fastest Europe's Supercomputer

On 26th May, 2009, Europe unveiled a new supercomputer with the power of 50,000 home PCs. It is *fastest in Europe* and the *third fastest worldwide*. This computer is named as "*Jugene*". It is capable of 1,000,000,000,000,000 (or 10^{15}) calculations per second. It is even faster than Road Runner and Jaguar computers in US, as said by Kosta Schinarakis from the Juelich research centre at Germany where Jugene is located. This machine is not an ordinary PC. It requires 2,95,000 processors located in 72 lockers, each of a size of telephone box.

Revised Edition

ADVANCED COMPUTER ARCHITECTURE

(A Practical Approach)

- Parallel Algorithms • Parallel Programming • Super Computers

