DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE MATERIALS

CS 405 COMPUTER SYSTEM ARCHITECTURE

VISION OF THE INSTITUTION

To mould true citizens who are millennium leaders and catalysts of change through excellence in education.

MISSION OF THE INSTITUTION

NCERC is committed to transform itself into a center of excellence in Learning and Research in Engineering and Frontier Technology and to impart quality education to mould technically competent citizens with moral integrity, social commitment and ethical values.

We intend to facilitate our students to assimilate the latest technological know-how and to imbibe discipline, culture and spiritually, and to mould them in to technological giants, dedicated research scientists and intellectual leaders of the country who can spread the beams of light and happiness among the poor and the underprivileged.

ABOUT DEPARTMENT

- Established in: 2002
- Course offered : B.Tech in Computer Science and Engineering

  M.Tech in Computer Science and Engineering

  M.Tech in Cyber Security
- Approved by AICTE New Delhi and Accredited by NAAC
- Affiliated to the University of A P J Abdul Kalam Technological University.

DEPARTMENT VISION

Producing Highly Competent, Innovative and Ethical Computer Science and Engineering Professionals to facilitate continuous technological advancement.

DEPARTMENT MISSION

1. To Impart Quality Education by creative Teaching Learning Process
2. To Promote cutting-edge Research and Development Process to solve real world problems with emerging technologies.
3. To Inculcate Entrepreneurship Skills among Students.
4. To cultivate Moral and Ethical Values in their Profession.

PROGRAMME EDUCATIONAL OBJECTIVES

PEO1: Graduates will be able to Work and Contribute in the domains of Computer Science and Engineering through lifelong learning.

PEO2: Graduates will be able to Analyse, design and development of novel Software Packages, Web Services, System Tools and Components as per needs and specifications.

PEO3: Graduates will be able to demonstrate their ability to adapt to a rapidly changing environment by learning and applying new technologies.

PEO4: Graduates will be able to adopt ethical attitudes, exhibit effective communication skills, Teamworkand leadership qualities.

PROGRAM OUTCOMES (POS)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES (PSO)

PSO1: Ability to Formulate and Simulate Innovative Ideas to provide software solutions for Real-time Problems and to investigate for its future scope.

PSO2: Ability to learn and apply various methodologies for facilitating development of high quality System Software Tools and Efficient Web Design Models with a focus on performance

optimization.

PSO3: Ability to inculcate the Knowledge for developing Codes and integrating hardware/software products in the domains of Big Data Analytics, Web Applications and Mobile Apps to create innovative career path and for the socially relevant issues.

COURSE OUTCOMES

CO1	To understand concepts of different parallel computer models.
CO2	To analyze the advanced processor technologies and understand the importance of memory hierarchy.
CO3	To analyze different multiprocessor system interconnecting mechanisms and discuss protocols for enforcing cache coherence.
CO4	To analyze different message passing mechanisms.
CO5	To analyze and design different pipe lining techniques.
CO6	To appraise concepts of multithreaded and data flow architectures.

MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES

	PO 1	**PO 2**	**PO 3**	**PO 4**	**PO 5**	**PO 6**	**PO 7**	**PO 8**	**PO 9**	**PO 10**	**PO 11**	**PO 12**
CO1	3	3	3	3	-	-	-	-	-	-	-	-
CO2	3	2	3	2	-	-	-	-	-	-	-	-
CO3	3	3	2	3	2	-	-	-	-	-	-	-
CO4	3	3	3	3	2	-	-	-	-	-	-	-
CO5	3	2	3	3	-	-	-	-	-	-	-	-
CO6	3	2	2	3	-	-	-	-	-	-	-	-

MAPPING OF COURSE OUTCOMES WITH PROGRAM OUTCOMES

	PSO1	PSO2	PSO3
CO1	2	-	-
CO2	3	3	-
CO3	3	2	-
CO4	2	-	-
CO5	3	-	-
CO6	3	2	-

Note: H-Highly correlated=3, M-Medium correlated=2, L-Less correlated=1

SYLLABUS

Course code	Course Name	L-T-P -Credits	Year of Introduction
CS405	COMPUTER SYSTEM ARCHITECTURE	3-0-0-3	2016

Course Objectives:

- To impart a basic understanding of the parallel architecture and its operations
- To introduce the key features of high performance computers

Syllabus:

Basic concepts of parallel computer models, SIMD computers, Multiprocessors and multi-computers, Cache Coherence Protocols, Multicomputers, Pipelining computers and Multithreading.

Expected outcome :

The Students will be able to :

i. summarize different parallel computer models
ii. analyze the advanced processor technologies
iii. interpret memory hierarchy
iv. compare different multiprocessor system interconnecting mechanisms
v. interpret the mechanisms for enforcing cache coherence
vi. analyze different message passing mechanisms
vii. analyze different pipe lining techniques
viii. appraise concepts of multithreaded and data flow architectures

Text Book:

- K. Hwang and Naresh Jotwani, Advanced Computer Architecture, Parallelism, Scalability, Programmability, TMH, 2010.

References:

1. H P Hayes, Computer Architecture and Organization, McGraw Hill, 1978.
2. K. Hwang & Briggs , Computer Architecture and Parallel Processing, McGraw Hill International, 1986
3. M J Flynn, Computer Architecture: Pipelined and Parallel Processor Design, Narosa Publishing House, 2012.
4. M Sasikumar, D Shikkare and P Raviprakash, Introduction to Parallel Processing, PHI, 2014.
5. P M Kogge, The Architecture of Pipelined Computer, McGraw Hill, 1981.
6. P V S Rao , Computer System Architecture, PHI, 2009.
7. Patterson D. A. and Hennessy J. L., Morgan Kaufmann , Computer Organization and Design: The Hardware/Software Interface, Morgan Kaufmann Pub, 4/e, 2010.

Course Plan

Module	Contents	Hours	End Sem. Exam Marks
I	Parallel computer models - Evolution of Computer Architecture, System Attributes to performance, Amdahl's law for a fixed workload. Multiprocessors and Multicomputers, Multivector and SIMD computers, Architectural development tracks, Conditions of parallelism.	6	15%

II	Processors and memory hierarchy - Advanced processor technology- Design Space of processors, Instruction Set Architectures, CISC Scalar Processors, RISC Scalar Processors, Superscalar and vector processors, Memory hierarchy technology.	8	15%
FIRST INTERNAL EXAM			
III	Multiprocessors system interconnects - Hierarchical bus systems, Cross bar switch and multiport memory, Multistage and combining networks. Cache Coherence and Synchronization Mechanisms, Cache Coherence Problem, Snoopy Bus Protocol, Directory Based Protocol, Hardware Synchronization Problem	7	15%
IV	Message Passing Mechanisms-Message Routing schemes, Flow control Strategies, Multicast Routing Algorithms. Pipelining and Superscalar techniques - Linear Pipeline processors and Nonlinear pipeline processors	8	15%
SECOND INTERNAL EXAM			
V	Instruction pipeline design, Arithmetic pipeline deign - Super Scalar Pipeline Design	8	20%
VI	Multithreaded and data flow architectures - Latency hiding techniques, Principles of multithreading - Multithreading Issues and Solutions, Multiple context Processors, Fine-grain Multicomputer- Fine-grain Parallelism. Dataflow and hybrid architecture	8	20%
END SEMESTER EXAM			

Question Paper Pattern (End semester exam)

1. There will be ***FOUR*** parts in the question paper - **A, B, C, D**
2. **Part A**
   a. **Total marks : 40**
   b. ***TEN*** questions, each have **4 marks**, covering **all the SIX modules** (***THREE*** questions from **modules I & II**; ***THREE*** questions from **modules III & IV**; ***FOUR*** questions from **modules V & VI**).
   All the TEN questions have to be answered.
3. **Part B**
   a. **Total marks : 18**
   b. ***THREE*** questions, each having **9 marks**. One question is from **module I**; one

question is from **module II**; one question ***uniformly*** covers **modules I & II**.

c. ***Any TWO*** questions have to be answered.

d. Each question can have ***maximum THREE*** subparts.

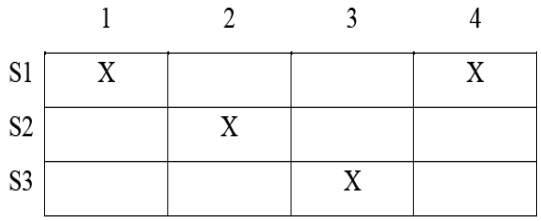
4. **Part C**
   a. **Total marks : 18**
   b. ***THREE*** questions, each having **9 marks**. One question is from **module III**; one question is from **module IV**; one question ***uniformly*** covers **modules III & IV**.
   c. ***Any TWO*** questions have to be answered.
   d. Each question can have ***maximum THREE*** subparts.
5. **Part D**
   a. **Total marks : 24**
   b. ***THREE*** questions, each having **12 marks**. One question is from **module V**; one question is from **module VI**; one question ***uniformly*** covers **modules V & VI**.
   c. ***Any TWO*** questions have to be answered.
   d. Each question can have ***maximum THREE*** subparts.
6. There will be ***AT LEAST* 60%** analytical/numerical questions in all possible combinations of question choices.

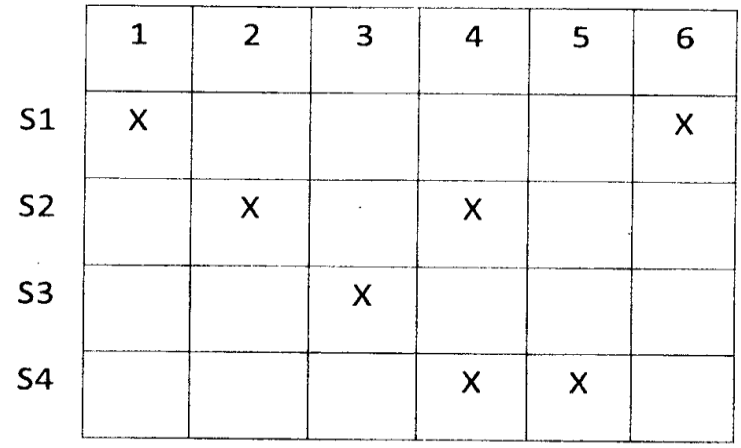
QUESTION BANK

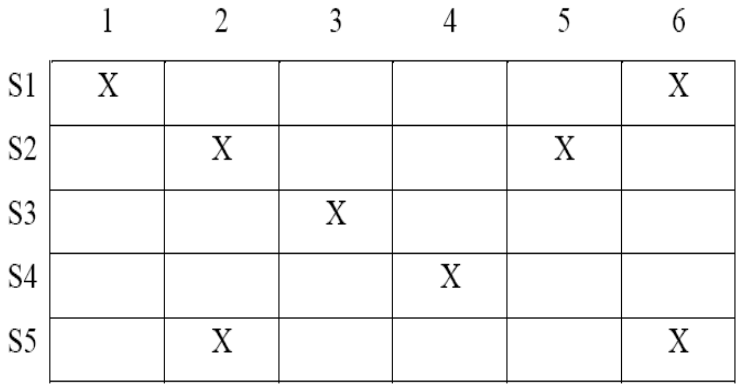
MODULE I				
Q:NO:	**QUESTIONS**	**CO**	**KL**	**PAGE NO:**
1	Explain Flynn's classification of computer architecture.	CO1	K5	3
2	Explain in detail about implicit and explicit parallelism in parallel programming.	CO1	K5	13
3	State Amdahl's law. Write an expression for the overall speed up.	CO1	K3	14
4	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. Rewrite the code using parallel constructs such as Cobegin and Coend. No variable substitution is allowed. All statements can be executed in parallel if they are declared within the same block of a (Cobegin and Coend) pair. S1: A=B+C S2: C=D+E S3: F=G+E S4: C=A+F S5: M=G+C S6: A=L+E S7: A=E+A	CO1	K2	42
5	Write about the Shared-Memory Multiprocessors.	CO1	K3	20
6	Write about the development layers for computer	CO1	K3	6

	system development.			
7	Write about the performance factors versus system attributes.	CO1	K3	10
	MODULE II			
1	Explain the terms (i) Hit Ratio (ii) Effective Access Time with proper equations.	CO2	K5	80
2	Compare the characteristics of CISC and RISC Architectures.	CO2	K5	63
3	Explain VLIW architecture. Also explain pipelining in VLIW processors.	CO2	K5	69
4	Draw memory hierarchy.	CO2	K1	74
5	Write about the inclusion property and data transfer between adjacent levels of a memory hierarchy.	CO2	K3	77
6	You are asked to perform capacity planning for a two-level memory system. The first level, M1, is a cache with three capacity choices of 64 Kbytes, 128 Kbytes, and 256 Kbytes. The second level, M_2, is a main memory with a 4-Mbyte capacity. Let c_1 and c_2 be the cost per 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 ratios for the three capacities are assumed to be 0.7, 0.9 and 0.98 respectively. (i) What is the average access time ta in terms of $t_1=20$ ns in the three cache designs? (Note that t_1 is the time form CPU to M_1 and t_2 is that from CPU to M_2) (ii) Express the average byte cost of the entire memory hierarchy if $c_2=\$0.2$/Kbyte.	CO2	K2	80
7	Explain the role of compilers in exploiting parallelism.	CO2	K5	83
8	Consider the design of a three level memory hierarchy with the following specifications for memory characteristics:	CO2	K3	82

	<table><tr><th>Memory level</th><th>Access time</th><th>Capacity</th><th>Cost/Kbyte</th></tr><tr><td>Cache</td><td>t1=25 ns</td><td>s1=512 Kbytes</td><td>c1=$1.25</td></tr><tr><td>Main Memory</td><td>t2=903 ns</td><td>s2=32 Mbytes</td><td>c2=$0.2</td></tr><tr><td>Disk array</td><td>t3=4 ms</td><td>s3 =39.8 Gbytes</td><td>c3=$0.0002</td></tr></table> Hit ratio of cache memory is h1=0.98 and a hit ratio of main memory is h2=0.9. (i) Calculate the effective access time. (ii) Calculate the total memory cost.			
	MODULE III			
1	Differentiate between crossbar network and multiport memory.	CO3	K4	90
2	Explain in detail about the network characteristics of multiprocessor system interconnects.	CO3	K5	84
3	Write about the schematic design of a cross point switch in a crossbar network with diagram.	CO3	K3	93
4	Explain hot spot problem.	CO3	K5	103
5	Design an 8 input omega network using 2X2 switches as building blocks. Show the switch settings for the permutations π_1=(0,7,6,4,2)(1,3)(5). Show the conflicts in switch settings, if any. Explain blocking and non-blocking networks in this context.	CO3	K6	100
6	Explain routing in Omega networks.	CO3	K5	97
7	Show an inter-processor-memory crossbar network.	CO3	K2	91
8	Explain bus system at board level ,backplane level and I/O level with diagram.	CO3	K5	88
9	Compare Omega and Butterfly multistage networks.	CO3	K5	96
	MODULE IV			
1	Differentiate between store and forward and wormhole	CO4	K4	134

	routing.			
2	Consider the following pipeline reservation table:  i) What are the forbidden latencies? ii) Draw the transition diagram. iii) List all the simple cycles and greedy cycles. iv) Determine the optimal constant latency cycle and minimal average latency (MAL) v) Let the pipeline clock period be τ=20ns. Determine the throughput of the pipeline.	CO4	K5	162
3	Write about dimension order routing. Consider a 16 node hypercube network. Based on E-cube routing algorithm, show how to route a message from 0010 to 1001. Find all intermediate nodes on routing path.	CO4	K3	140
4	Determine the frequency of the pipeline if the stage delays are $\tau1$ = 3ns, $\tau2$ = $\tau3$ =5ns and $\tau4$ =8 ns and the latch delay is 1 ns.	CO4	K5	155
5	Consider the five-stage pipelined processor specified by the following reservation table and answer the following: (S indicate the stages).	CO4	K5	162

	 i) List the set of forbidden latencies and the collision vector. ii) Draw the state transition diagram showing all possible initial sequences without causing a collision in the pipeline. iii) List all the simple and greedy cycles from the state diagram. iv) Determine the minimum average latency.			
6	Consider a 16-node hypercube network. Based on the E-cube routing algorithm, show how to route a message from node (0111) to node (1101). All intermediate nodes must be identified on the routing path.	CO4	K2	141
7	Differentiate between synchronous and asynchronous model of linear pipeline processors.	CO4	K4	154
8	Consider the following pipeline reservation table:	CO4	K5	162

	 i) What are the forbidden latencies? ii) Draw the transition diagram. iii) List all the simple cycles and greedy cycles. iv) Determine the optimal constant latency cycle and minimal average latency (MAL) v) Let he pipeline clock period be τ=20ns. Determine the throughput of the pipeline.			
9	Write about virtual networks and network partitioning.	CO4	K3	150
	MODULE V			
1	Write about the possible hazards that can occur between read and write operations in an instruction pipeline.	CO5	K3	178
2	Explain the Tomasulo's algorithm for the dynamic instruction scheduling.	CO5	K5	181
3	Compile the concept of in-order issue and out-of-order issue with respect to superscalar processor.	CO5	K6	203
4	Write short notes on internal data forwarding.	CO5	K3	176
5	Explain static branch prediction strategy and dynamic branch prediction strategy.	CO5	K5	187
6	Explain the effect of branching in instruction pipelining. Find the execution time and throughput of the pipeline for n instructions by considering the effect	CO5	K5	185

	of branching. How branch penalty is reduced using delayed branch strategy.			
7	With an example differentiate between the Carry-Save Adders (CSA) and Carry Propagate Adder (CPA).	CO5	K4	193
8	Construct a pipeline unit for fixed-point multiplication of 8-bit integers using CSA and CPA.	CO5	K3	197
	MODULE VI			
1	Elaborate any three latency hiding techniques used in distributed shared memory multi computers.	CO6	K6	221
2	With a neat diagram discuss the architecture of ETL/EM-4 dataflow architecture.	CO6	K2	265
3	Distinguish between static dataflow computers and dynamic dataflow computers.	CO6	K4	262
4	Write about the consistencies in strong and relaxed memory models for building scalable multiprocessors with distributed shared memory.	CO6	K3	235
5	Discuss any two latency hiding techniques used in distributed shared memory multicomputers.	CO6	K2	221
6	Write about any two context switching polices for multithreaded architecture.	CO6	K3	238
7	Write about Multithreading issues and solutions.	CO6	K3	238
8	Illustrate the scalable coherence interface (SCI) interconnect model.	CO6	K3	229
9	Write a short note on fine-grain parallelism.	CO6	K3	250

APPENDIX 1

CONTENT BEYOND THE SYLLABUS

MODULE NOTES

CS 405

COMPUTER SYSTEM ARCHITECTURE

Module I

PARALLEL COMPUTER MODELS

(1)

For higher performance, lower costs, and sustained productivity in real-life applications parallel processing is used.

Forms of parallelism include:

1. lookahead,
2. pipelining,
3. vectorization,
4. concurrency,
5. simultaneity,
6. data parallelism,
7. partitioning,
8. interleaving,
9. overlapping,
10. multiplicity,
11. replication
12. time sharing
13. space sharing
14. multitasking
15. multiprogramming,
16. multithreading and
17. distributed computing at different processing levels.

(2) Evolution of Computer Architecture

The study of computer architecture involves

1. Hardware organization

Abstract machine is organized with

a. CPUs
b. caches
c. buses
d. microcode
e. pipelines
f. Physical memory etc.

2. Programming / software requirements.

Over the past four decades, computer architecture has gone through evolutional rather than revolutional changes.

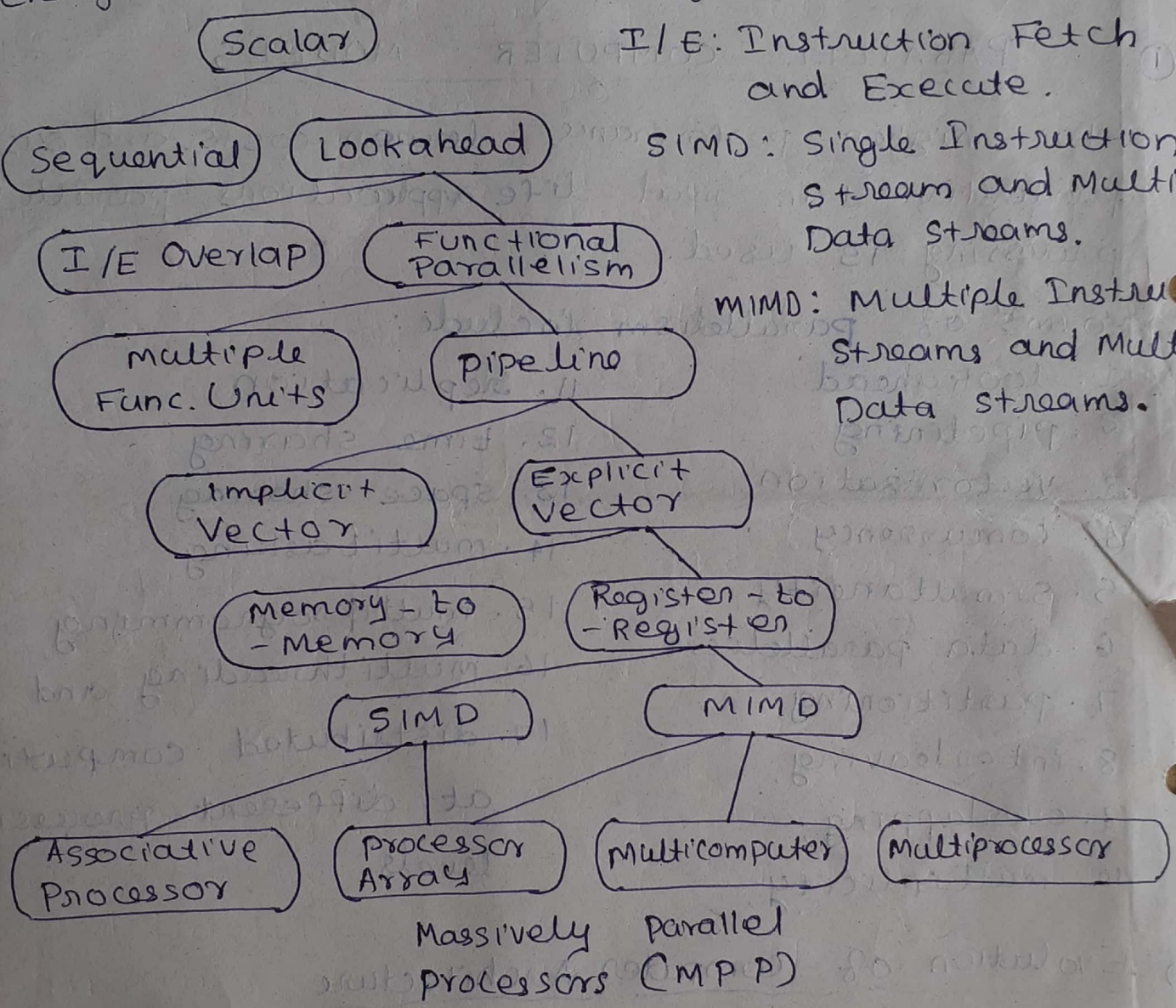


Figure 1.1 Tree showing architectural evolution from sequential scalar computers to vector processors and parallel computers.

Lookahead

1. To prefetch instructions in order to overlap I/E (instruction fetch / decode and execution) and
2. To enable functional parallelism.

Functional Parallelism

Two approaches

1. To use multiple functional units simultaneously and
2. To practice pipelining at various processing levels.

Pipelining

Include

1. Pipelined instruction execution,
2. pipelined arithmetic computations and
3. memory-access operations.

Flynn's Classification

Michael Flynn (1972) introduced a classification of various computer architectures based on

1. notions of instruction and
2. data streams.

1. Conventional sequential machines are SISD (single instruction stream over a single data stream) computers.
2. Vector computers are SIMD (single instruction stream over multiple data streams) machines.

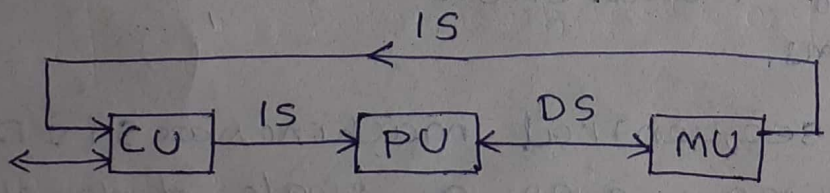
3. Parallel computers are MIMD (multiple instruction streams over multiple data streams) machines.
4. Systolic arrays for pipelined execution are MISD (multiple instruction streams and a single data stream) machines.

parallel / vector computers.

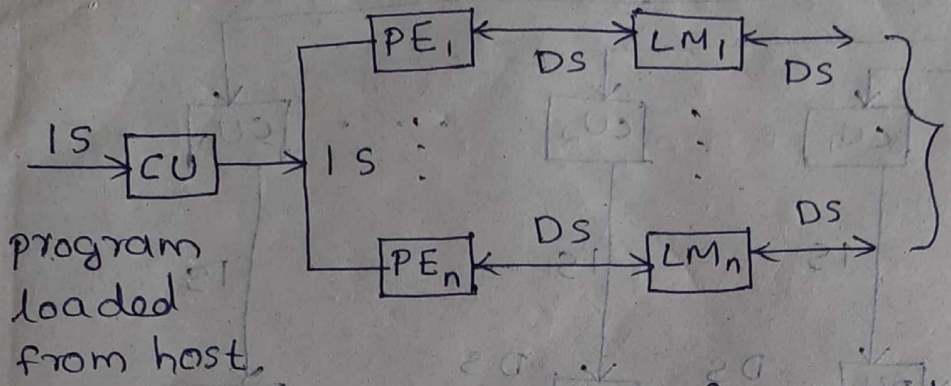
Two classes of parallel computers
1. shared-memory multiprocessors and
2. message-passing multicomputers.

Differs in:
i. memory sharing and
ii. mechanisms used for interprocessor communication.

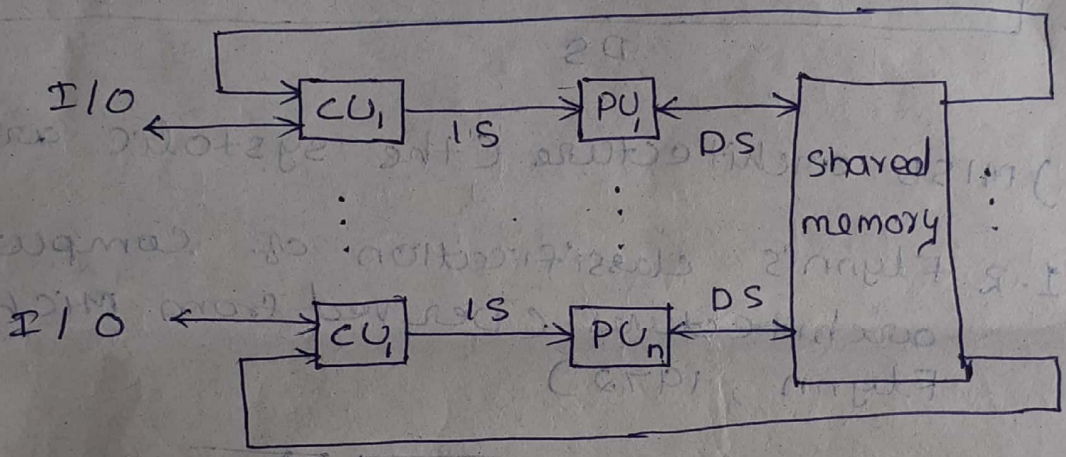
Two families of pipelined vector processors:
1. Memory-to-memory
pipelined flow of vector operands directly from memory to pipelines and back.
2. Register-to-register
Uses vector registers to interface between the memory and functional pipelines.



(a) SISD uniprocessor architecture



(b) SIMD architecture (with distributed memory)



(c) MIMD architecture (with shared memory)

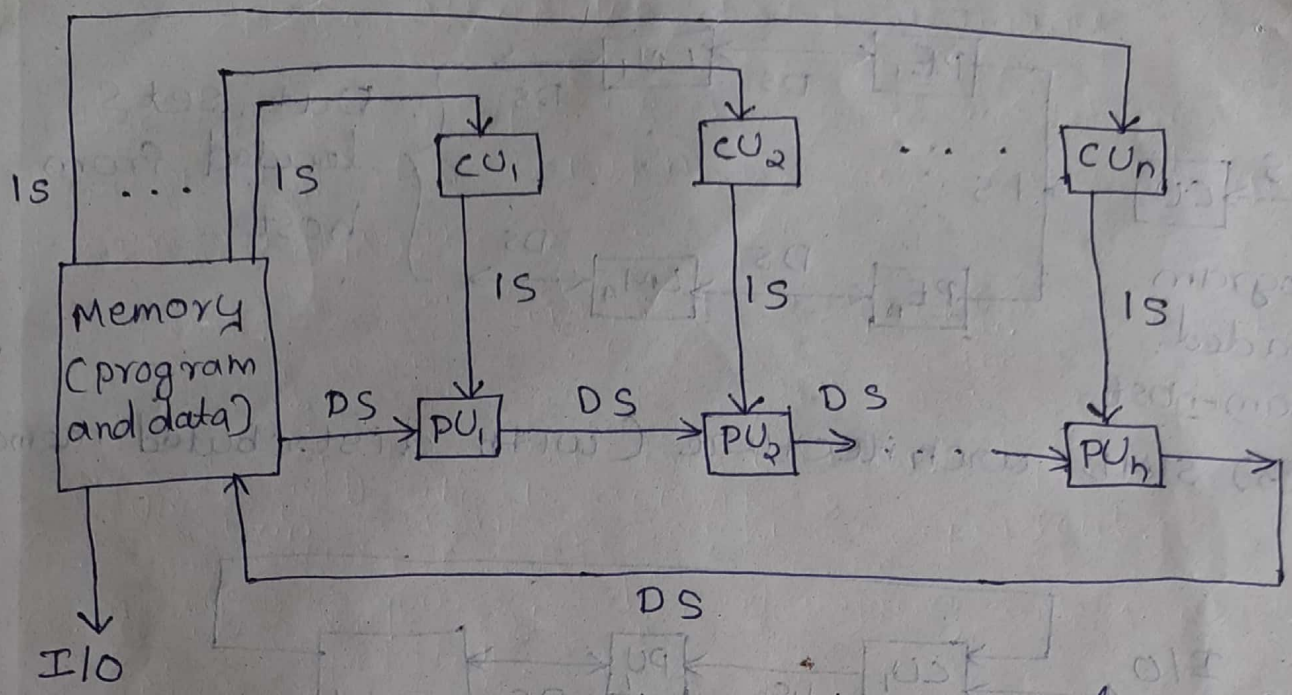
Captions:

CU = Control Unit PU = Processing Unit

MU = Memory Unit IS = Instruction stream

DS = Data stream PE = Processing Element

LM = Local Memory.



(d) MISD architecture (the systolic array)

Figure 1.2 Flynn's classification of computer architectures (Derived from Michael Flynn, 1972)

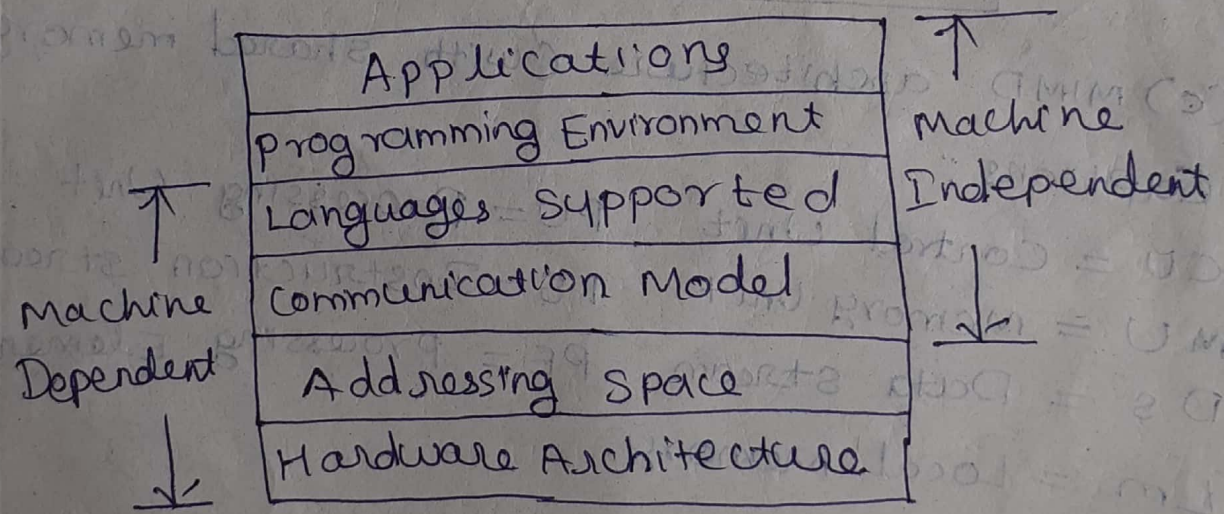


Figure 1.3 Six layers for computer system development.

Development Layers

Figure 1.3 illustrates a layered development of parallel computers based on a recent classification by Lionel Ni (1990).

1. Hardware configurations differ from machine to machine.
2. The address space of a processor in a computer system varies among different architectures - depends on memory organisation, which is machine-dependent.
3. Application programs and programming environments are machine-independent.

Independent of machine architecture means the user programs can be ported to many computers with minimum conversion costs.

4. High-level languages and communication models depend on the architectural choices made in a computer system.

New Challenges.

1. It is still very difficult and painful to program parallel and vector computers.
2. Need to strive for major progress in the software area.
3. A whole new generation of programmers need to be trained to program parallelism effectively.

Real-life problems include:
weather forecast modeling, computer-aided design of VLSI circuits, large-scale database management, artificial intelligence, crime control, and strategic defense initiatives.

(3) System Attributes to performance

The ideal performance of a computer system demands a perfect match between machine capability and program behavior.

The simplest measure of program performance is the turnaround time, which includes

i. disk and memory accesses,
ii. input and output activities,
iii. compilation time,
iv. OS overhead, and
v. CPU time.

In order to shorten the turnaround time, one must reduce all these time factors.

Clock Rate and CPI

CPU (processor) driven by a clock with a constant cycle time (T in nanoseconds).

clock rate $f = 1/T$ in megahertz.

The size of a program = instruction count (I_c)

cycles per instruction (CPI) - Different machine instructions require different numbers of clock cycles to execute.

The cycles per instruction (CPI) becomes an important parameter for measuring the time needed to execute each instruction.

An accurate estimate of the average CPI requires a large amount of program code to be traced over a long period of time.

Performance Factors

1. The CPU time (T in seconds / program) needed to execute the program is:

$$T = I_c \times CPI \times \tau \qquad (1.1)$$

where

I_c : be the number of instructions in a given program, or the instruction count.

CPI : cycles per instruction

τ : constant cycle time

2. The CPI of an instruction type can be divided into two component terms
   1. the total processor cycles and
   2. memory cycles

needed to complete the execution of the instruction.

3. The complete instruction cycle may involve one to four memory references:
   i. one for instruction fetch,
   ii. two for operand fetch, and
   iii. one for store results.

Therefore we can rewrite Eq. 1.1 as follows:

$$\boxed{T = I_c \times (P + m \times k) \times \tau} \qquad (1.2)$$

where,

P is the number of processor cycles needed for the instruction decode and execution,

m is the number of memory references needed,

k is the ratio between memory cycle and processor cycle,

I_c is the instruction count, and
τ is the processor cycle time.

System Attributes

The five performance factors (I_c, P, m, k, T) are influenced by four system attributes:

i. instruction-set architecture, (affects I_c & P);
ii. compiler technology, (affects I_c, P, and m)
iii. CPU implementation and control, and (affects P.T)
iv. cache and memory hierarchy (affects k.T).

I_c - program length
P - processor cycle needed
m - memory reference count
P.T - total processor time.
k.T - memory access latency.

Table 1.2 Performance Factors Versus System Attributes

System Attributes	Performance Factors				
	Instr. Count, I_c	Average Cycles per Instruction, CPI			Processor Cycle Time, T
		Processor Cycles per Instruction, P	Memory References per Instruction, m	Memory-Access Latency, k	
Instruction-set Architecture	X	X			
Compiler Technology	X	X	X		
Processor Implementation and Control		X			X
Cache and Memory Hierarchy				X	X

MIPS Rate

Let C be the total number of clock cycles needed to execute a given program.

Then the CPU time $T = C \times \tau = C/f$.

$$CPI = C/I_c$$

$$T = I_c \times CPI \times \tau$$

$$= I_c \times CPI / f.$$

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

MIPS rate varies with respect to

1. the clock rate (f),
2. the instruction count (I_c), and
3. the CPI of a given machine.

$$\text{MIPS rate} = \frac{I_c}{T \times 10^6} = \frac{f}{CPI \times 10^6} = \frac{f \times I_c}{C \times 10^6} \quad (1.3)$$

Eq. 1.2 can be written by using 1.3 as

$$T = I_c \times 10^{-6} / \text{MIPS}$$

MIPS rate of a given computer is directly proportional to the clock rate and inversely proportional to the CPI.

Throughput Rate

W_s : how many programs a system can execute per unit time, called the system throughput W_s. (in programs/second).

W_p : CPU throughput (1.4)

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

From Eq. 1.3

$$W_p = (MIPS) \times 10^6 / I_c$$

The unit for W_p is programs / second.

1. The reason why $W_s < W_p$ is due to the additional system overheads caused by the I/O, compilers, and OS when multiple programs are interleaved for CPU execution by multiprogramming or time sharing operations.
2. If the CPU is kept busy in a perfect program - interleaving fashion, then $W_s = W_p$. This will probably never happen, since the system overhead often causes an extra delay and the CPU may be left idle for some cycles.

Programming Environments

The programmability of a computer depends on the programming environment provided to the users.

sequential environment

Conventional uniprocessor computers are programmed in a sequential environment in which instructions are executed one after another in a sequential manner, using languages, compilers, and operating systems all developed for a uniprocessor computer.

parallel environment

parallelism is automatically exploited. All should support parallel activities - compilers, the operating system, parallel scheduling.

Two approaches to parallel programming :

1. Implicit Parallelism

2. Explicit Parallelism

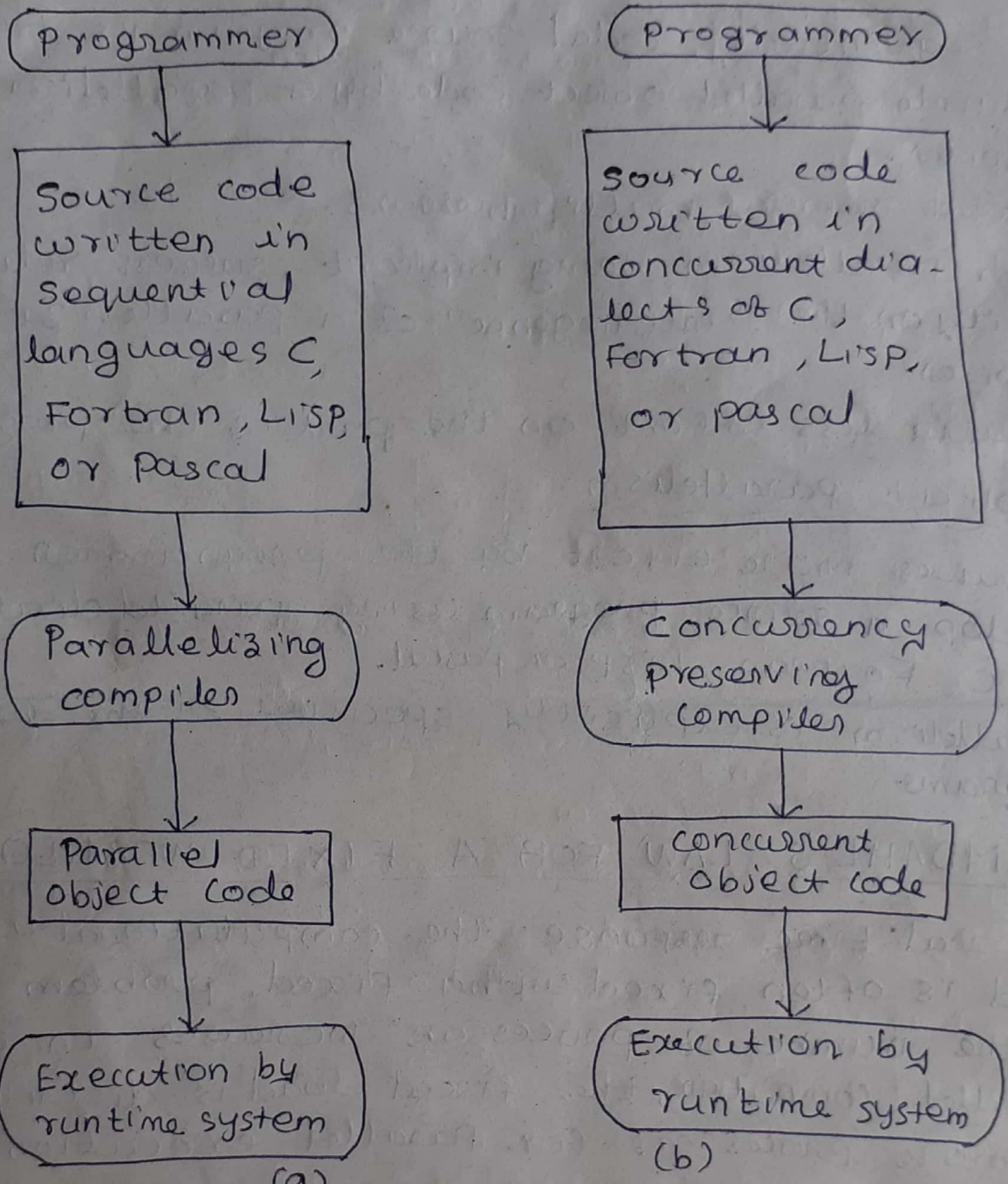


Figure 1.4 Two approaches to parallel programming. (a) Implicit parallelism (b) Explicit parallelism.

Implicit Parallelism

1. The sequentially coded source program is translated into parallel object code by a parallelizing compiler.
2. Shared-memory multiprocessors.
3. With parallelism being implicit, success relies heavily on the "intelligence" of a parallelizing compiler.
4. Requires less effort on the part of the programmer.

Explicit Parallelism

1. Requires more effort by the programmer to develop a source program using parallel dialects of C, Fortran, Lisp, or Pascal.
2. Parallelism is explicitly specified in the user programs.

④ ✓ AMDAHL'S LAW FOR A FIXED WORKLOAD

1. For real-time response, the computational workload is often fixed with a fixed problem size.
2. As the number of processors increases in a parallel computer, the fixed load is distributed to more processors for parallel execution.
3. The main objective is to produce the results as soon as possible.
   That is minimal turnaround time is the primary goal.
4. Speedup obtained for time-critical applications is called fixed-load speedup.

Fixed-Load Speedup

1. The execution of a program on a parallel computer use different numbers of processors at different time periods during the execution cycle.
2. For each time period, the number of processors used to execute a program is defined as the degree of parallelism (DOP).

Asymptotic speedup:

Denote the amount of work executed with DOP=i as $W_i = i \Delta t_i$ or

$$W = \sum_{i=1}^{m} W_i.$$

Δ - computing capacity of a single processor approximated by MIPS or Mflops without considering any latency.

The execution time of W_i on a single processor is

$$t_i(1) = W_i / \Delta.$$

t_i = total amount of time that DOP = i

The execution time of W_i on k processors is

$$t_i(k) = W_i / k\Delta.$$

With an infinite number of available processors,

$$t_i(\infty) = W_i / i\Delta \quad \text{for } 1 \le i \le m.$$

Thus the **response time**

$$T(1) = \sum_{i=1}^{m} t_i(1) = \sum_{i=1}^{m} \frac{W_i}{\Delta} \qquad (1.5)$$

$$T(\infty) = \sum_{i=1}^{m} t_i(\infty) = \sum_{i=1}^{m} \frac{W_i}{i\Delta} \qquad (1.6)$$

The asymptotic speedup S_∞ is defined as the ratio of $T(1)$ to $T(\infty)$:

$$S_\infty = \frac{T(1)}{T(\infty)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} W_i / i} \qquad (1.7)$$

1. The ideal speedup formula given in Eq. 1.7 is based on a fixed workload, regardless of the machine size.
2. Consider the case where DOP = $i > n$.
Assume all n processors are used to execute W_i exclusively.

The execution time of W_i is

$$t_i(n) = \frac{W_i}{i\Delta} \left\lceil \frac{i}{n} \right\rceil \qquad (1.8)$$

Thus the response time is

$$T(n) = \sum_{i=1}^{m} \frac{W_i}{i\Delta} \left\lceil \frac{i}{n} \right\rceil \qquad (1.9)$$

if $i < n$, then $t_i(n) = t_i(\infty) = W_i / i\Delta$.

The fixed-load speedup factor is the ratio of $T(1)$ to $T(n)$:

$$S_n = \frac{T(1)}{T(n)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i} \left\lceil \frac{i}{n} \right\rceil} \qquad (1.10)$$

Let $Q(n)$ be the sum of all system overheads due to

1. communication latencies caused by delayed memory access,
2. interprocessor communications over a bus or a network, or
3. Operating system overhead and delay caused by interrupts.

So Eq. (1.10) becomes

$$S_n = \frac{T(1)}{T(n)+Q(n)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i}\left\lceil \frac{i}{n} \right\rceil + Q(n)} \qquad (1.11)$$

The overhead delay $Q(n)$ is certainly application-dependent as well as machine-dependent.

Amdahl's Law

- In 1967, Gene Amdahl derived a fixed-load speedup for the special case where the computer operates either in sequential mode (with DOP=1) or in perfectly parallel mode (with DOP=n). That is, $W_i = 0$ if $i \neq 1$ or $i \neq n$ in the profile.

Equation 1.10 is then simplified to

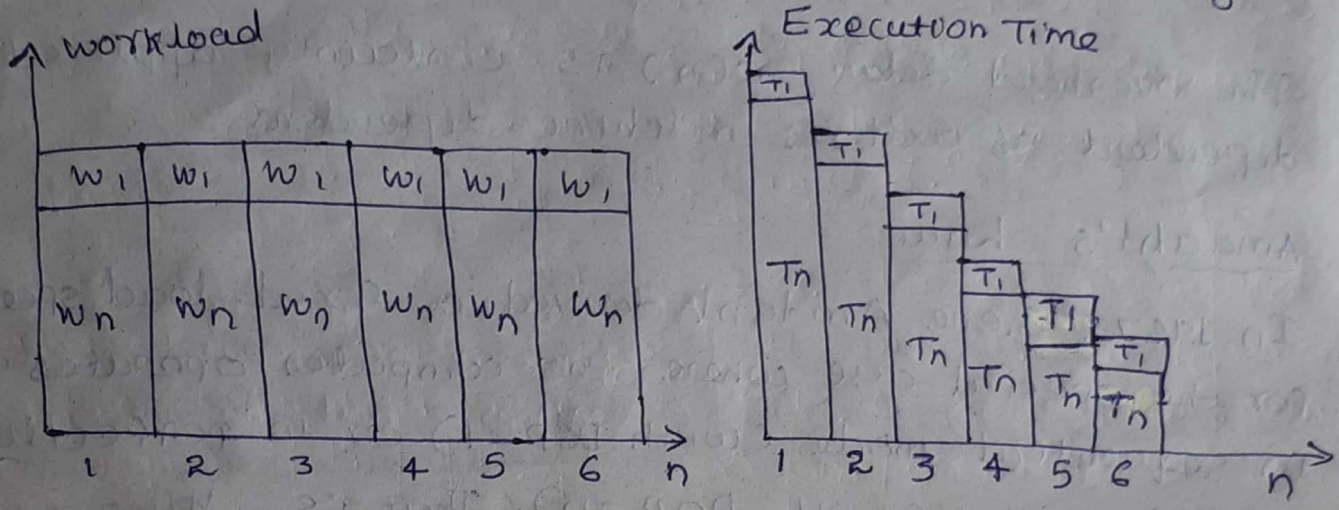
$$S_n = \frac{W_1 + W_n}{W_1 + W_n/n} \qquad (1.12)$$

- Amdahl's law implies that the sequential portion of the program W_1 does not change with respect to the machine size n. However, the parallel portion is evenly executed by n processors, resulting in a reduced time.

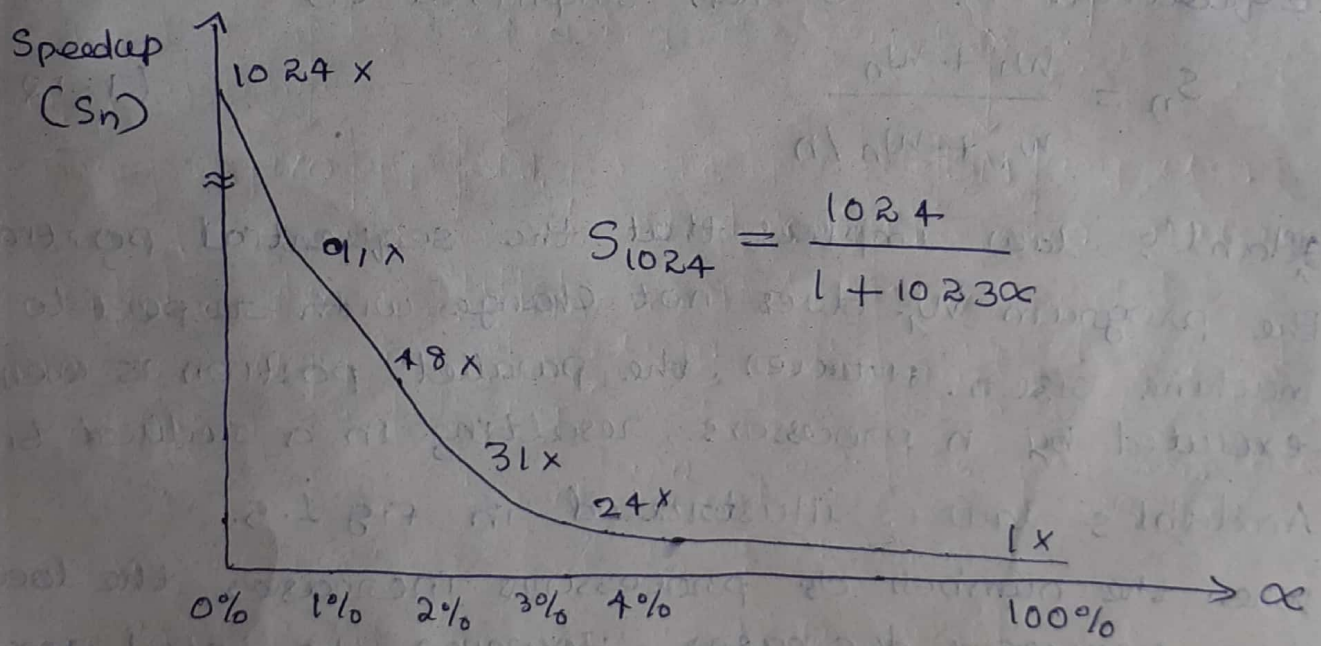
Amdahl's law is illustrated in Fig 1.5..

1. When the number of processors increases, the load on each processor decreases. However, the total amount of work (workload) $W_1 + W_n$ is kept constant as shown in Fig. 1.5a.

2. In Fig. 1.5b, the total execution time decreases because $T_n = W_n / n$. Eventually, the sequential part will dominate the performance because $T_n \to 0$ as n becomes very large and T_1 is kept unchanged.



(a) Fixed workload (b) Decreasing execution time



$$S_{1024} = \frac{1024}{1 + 1023\alpha}$$

Sequential fraction of program

(c) Speed up with a fixed load

Figure 1.5 Fixed-load speedup model and Amdahl's law.

Sequential Bottleneck.

Figure 1.5c plots Amdahl's law over the range $0 \leq \alpha \leq 1$. The maximum speedup $S_n = n$ if $\alpha = 0$. The minimum speedup $S_n = 1$ if $\alpha = 1$. As $n \to \infty$, the limiting value of $S_n \to 1/\alpha$. This implies that the speedup is upper-bounded by $1/\alpha$, as the machine size becomes very large.

The speedup curve in Fig. 1.5c drops very rapidly as α increases. This means that with a small percentage of the sequential code, the entire performance cannot go higher than $1/\alpha$. This α has been called the sequential bottleneck in a program.

The problem of a sequential bottleneck cannot be solved just by increasing the number of processors in a system.

Two major impacts on the parallel computer industry

1. manufacturers were discouraged from making large-scale parallel computers.
2. more research attention was shifted toward developing parallelizing compilers which will reduce the value of α and in turn boost the performance.

(5) MULTIPROCESSORS AND MULTICOMPUTERS

Two architectural organization models are

1. Shared-Memory Multiprocessors
2. Distributed-Memory Multicomputers.

Shared-Memory Multiprocessors

Three models

1. The uniform-memory-access (UMA) model,
2. The nonuniform-memory-access (NUMA) model, and
3. The cache-only memory architecture (COMA) model.

These models differ in how the memory and peripheral resources are shared or distributed.

The UMA model

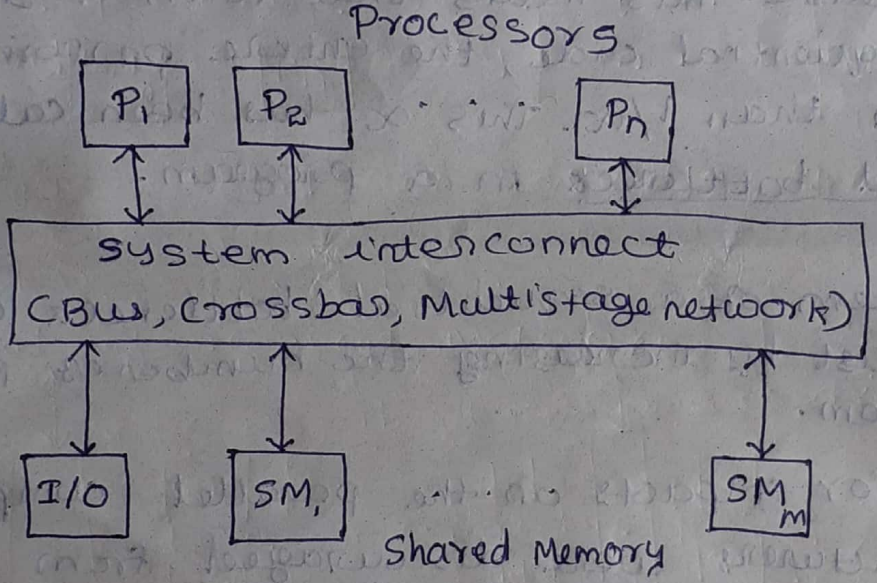


Figure 1.6 The UMA multiprocessor model

1. The physical memory is uniformly shared by all the processors.
2. All processors have equal access time to all memory words, which is why it is called uniform memory access.
3. Each processor use a private cache.
4. Peripherals are also shared in same fashion.
5. Multiprocessors are called tightly coupled systems due to the high degree of resource

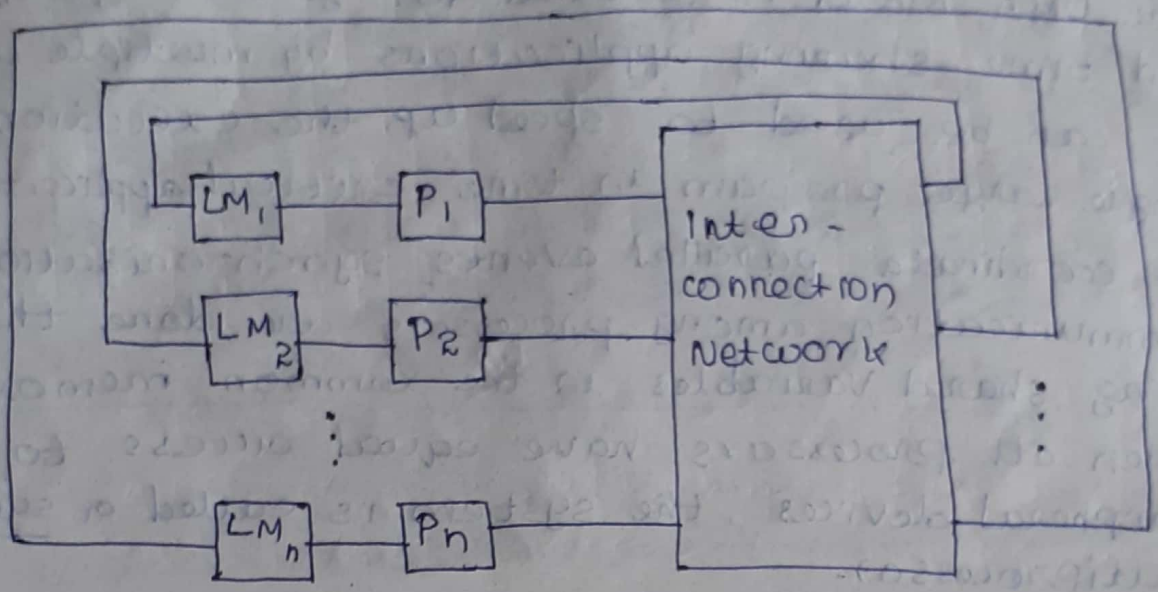
sharing.

6. Most computer manufacturers have multiprocessor (MP) extensions of their uniprocessor (UP) product line.
7. The UMA model is suitable for general-purpose and time-sharing applications by multiple users.
8. It can be used to speed up the execution of a single large program in time-critical applications.
9. To coordinate parallel events, synchronization and communication among processors are done through using shared variables in the common memory.
10. When all processors have equal access to all peripheral devices, the system is called a symmetric multiprocessor.
11. All the processors are equally capable of running the executive programs, such as the OS kernel and I/O service routines.
12. Asymmetric multiprocessor
    i. Only one or a subset of processors are executive-capable.
    ii. An executive or a master processor can execute the operating system and handle I/O. The remaining processors have no I/O capability and thus are called attached processors (APs).

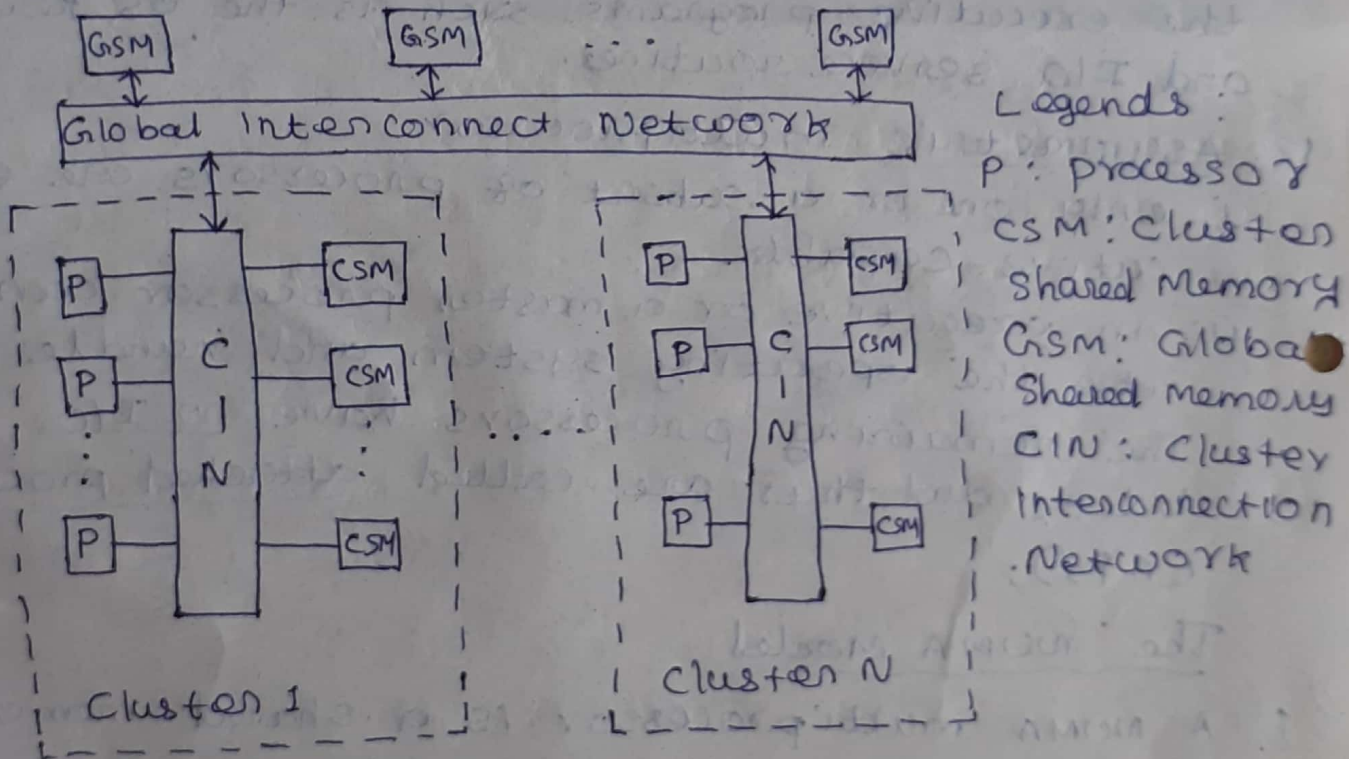
The NUMA Model

1. A NUMA multiprocessor is a shared-memory system in which the access time varies with the location of the memory word.
2. Local memories:
   The shared memory is physically distributed

to all processors, called local memories. The collection of all local memories forms a global address space accessible by all processors.



(a) Shared local memories (e.g., the BBN Butterfly)



(b) A hierarchical cluster model (e.g., the Cedar system at the University of Illinois)

Figure 1.7 Two NUMA models for multiprocessor systems.

3. Local
   i. It is faster to access a local memory with a local processor.
   ii. The access of remote memory attached to other processors takes longer due to the added delay through the interconnection network.
4. Globally shared memory can be added to a multiprocessor system.
   There are 3 memory-access patterns:
   i. The fastest is local memory access.
   ii. The next is global memory access.
   iii. The slowest is access of remote memory.
5. A hierarchically structured multiprocessor is modelled in Fig. 1.7b.
6. The processors are divided into several clusters.
7. Each cluster is itself an UMA or a NUMA multiprocessor. The clusters are connected to global shared-memory modules. All clusters have equal access to the global memory.

The COMA Model

1. A multiprocessor using cache-only memory assumes the COMA model.
   Examples:
   i. Swedish Institute of Computer Science's Data Diffusion Machine (DDM) and
   ii. Kendall Square Research's KSR-1 machine.
   The COMA model is depicted in Fig. 1.8. Details of it:

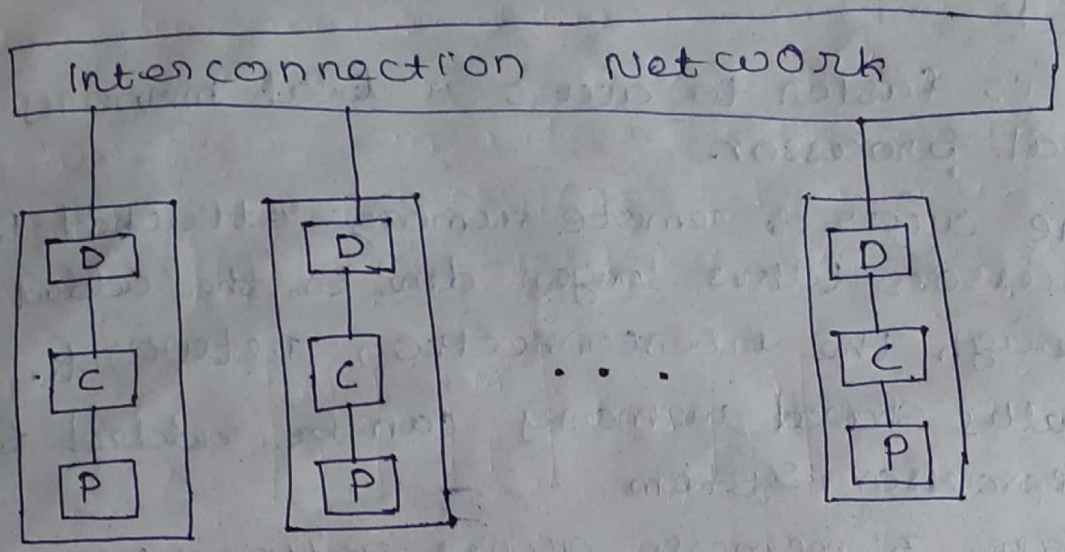


Figure 1.8 The COMA model of a multiprocessor.
(P: Processor, C: Cache, D: Directory;
eg., the KSR-1)

2. The COMA model is a special case of a NUMA machine, in which the distributed main memories are converted to caches.
3. There is no memory hierarchy at each processor node. All the caches form a global address space.
4. Remote cache access is assisted by the distributed cache directories.
5. Other variations for multiprocessors
   1. Cache-coherent non-uniform memory access (CC-NUMA)
      eg: Stanford Dash and MIT Alewife

Distributed-Memory Multicomputers

A distributed-memory multicomputer system is modeled in Fig. 1.9.

1. The system consists of multiple computers,

often called nodes, interconnected by a message-passing network.

2. Each node is an autonomous computer consisting of a processor, local memory, and sometimes attached disks or I/O peripherals.

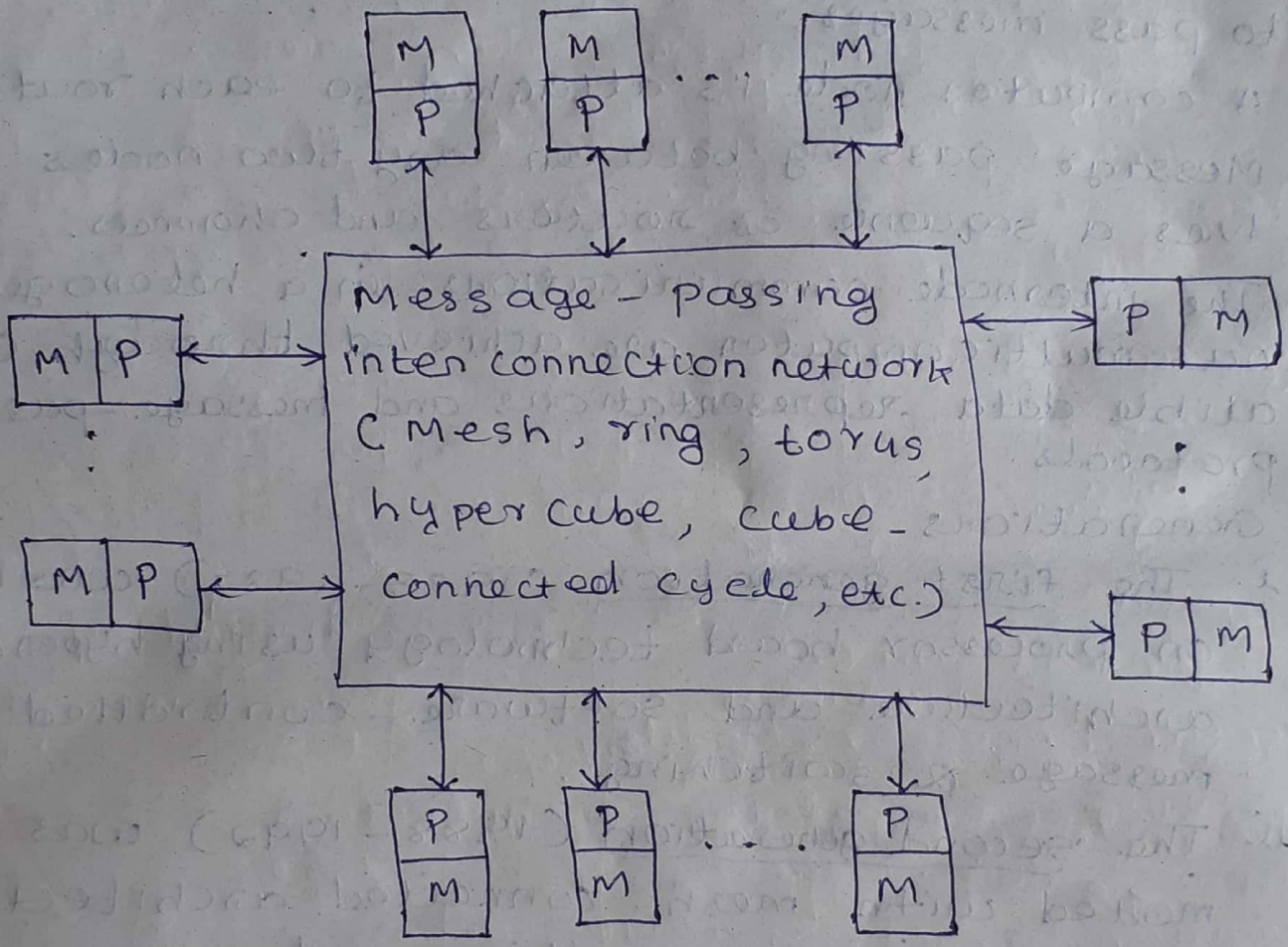


Figure 1.9 Generic model of a message-passing multicomputer.

3. The message-passing network provides point-to-point static connections among the nodes.
4. All local memories are private and are accessible only by local processors. (no-remote-memory-access (NORMA)).

5. Internode communication is carried out by passing messages through the static connection network.

Multicomputer Generations.

1. Modern multicomputers use hardware routers to pass messages.
2. A computer node is attached to each router.
3. Message passing between any two nodes involves a sequence of routers and channels.
4. The internode communications in a heterogeneous multicomputer are achieved through compatible data representations and message-passing protocols.
5. Generations
   i. The first generation (1983-1987) was based on processor board technology using hypercube architecture and software-controlled message switching.
   ii. The second generation (1988-1992) was implemented with mesh-connected architecture, hardware message routing, and a software environment for medium-grain distributed computing.
   iii. The emerging third generation (1993-1997) is expected to be fine-grain multicomputers.

(6) Multivector and SIMD Computers.

Supercomputers and parallel processors for vector processing and data parallelism.

1. Vector supercomputers.
2. SIMD supercomputers

Vector Supercomputers

1. A vector computer is built on top of a scalar processor, as shown in Fig. 1.10.

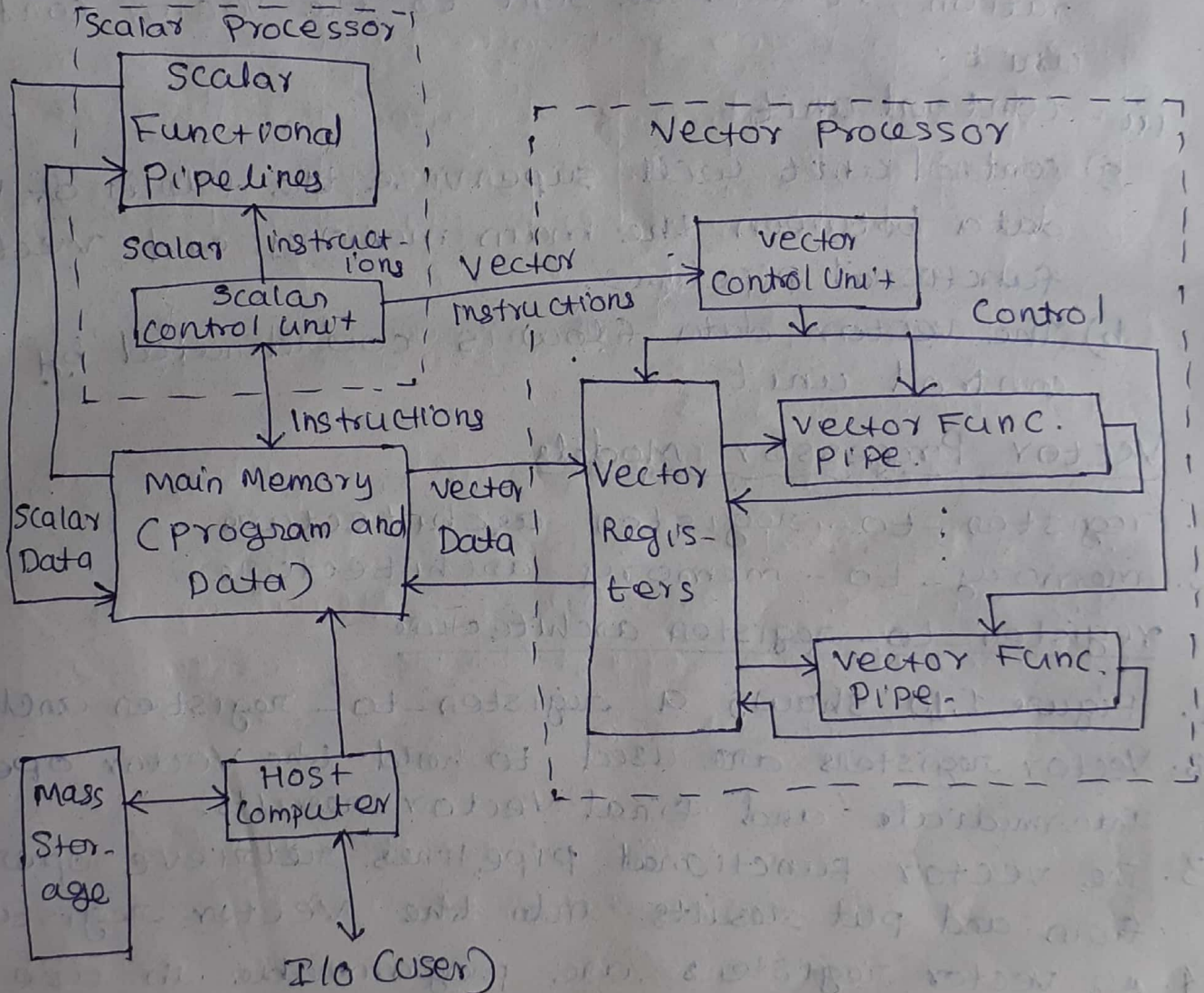


Figure 1.10 The architecture of a vector super-computer.

2. program and data are first loaded into the main memory through a host computer.
3. All instructions are first decoded by the scalar control unit.
   i. If the decoded instruction is a scalar operation or a program control operation, it will be directly executed by the scalar processor using the scalar functional pipelines.
   ii. If the instruction is decoded as a vector operation, it will be sent to the vector control unit.
   iii. control unit
      a) control unit will supervise the flow of vector data between the main memory and vector functional pipelines.
      b) The vector data flow is coordinated by the control unit.

Vector Processor Models

1. register-to-register architecture.
2. memory-to-memory architecture.

register-to-register architecture

1. Figure 1.10 shows a register-to-register architecture.
2. Vector registers are used to hold the vector operands, intermediate and final vector results.
3. The vector functional pipelines retrieve operands from and put results into the vector registers.
4. All vector registers are programmable in user instructions.
5. Each vector register is equipped with a component counter which keeps track of the component registers

used in successive pipeline cycles.

6. The length of each vector register is usually fixed, say, sixty-four 64-bit component registers in a vector register - Cray series supercomputer. Fujitsu VP2000 Series - use reconfigurable vector registers to dynamically match the register length with that of the vector operands.
7. There are fixed numbers of vector registers and functional pipelines in a vector processor.

memory-to-memory architecture

1. A memory-to-memory architecture differs from a register-to-register architecture in the use of a vector stream unit to replace the vector registers.
2. Vector operands and results are directly retrieved from the main memory in superwords.

SIMD Supercomputers

An operational model of SIMD computers is presented in Figure 1.11

SIMD Machine Model : An operational model of an SIMD computer is specified by a 5-tuple:

$$M = < N, C, I, M, R > \quad (1.13)$$

where

(1) N is the number of processing elements (PEs).
(2) C is the set of instructions directly executed by the control unit (CU)
(3) I is the set of instructions broadcast by the CU to all PEs for parallel execution.
eg: arithmetic, logic, data routing, masking

4. M is the set of masking schemes, where each mask partitions the set of PEs into enabled and disabled subsets.
5. R is the set of data-routing functions, specifying various patterns to be set up in the interconnection network for inter-PE communication.

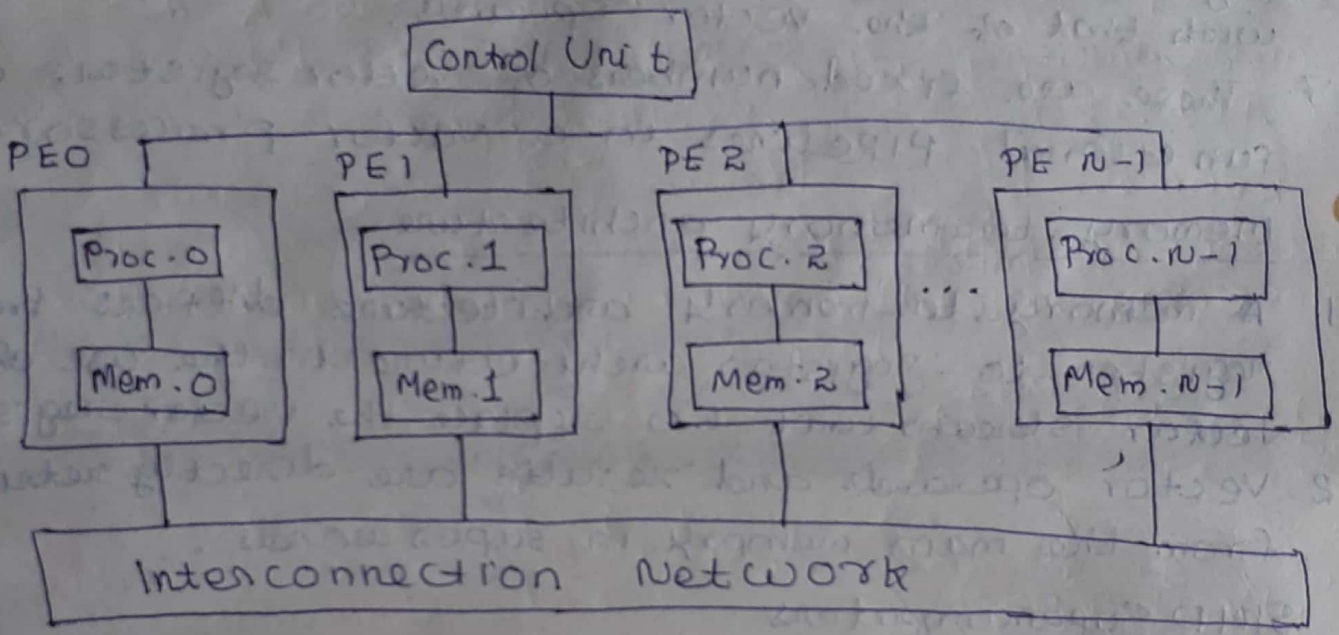


Figure 1.11 Operational model of SIMD computers.

(7) Architectural Development Tracks

The architectures of most existing computers follow certain development tracks.

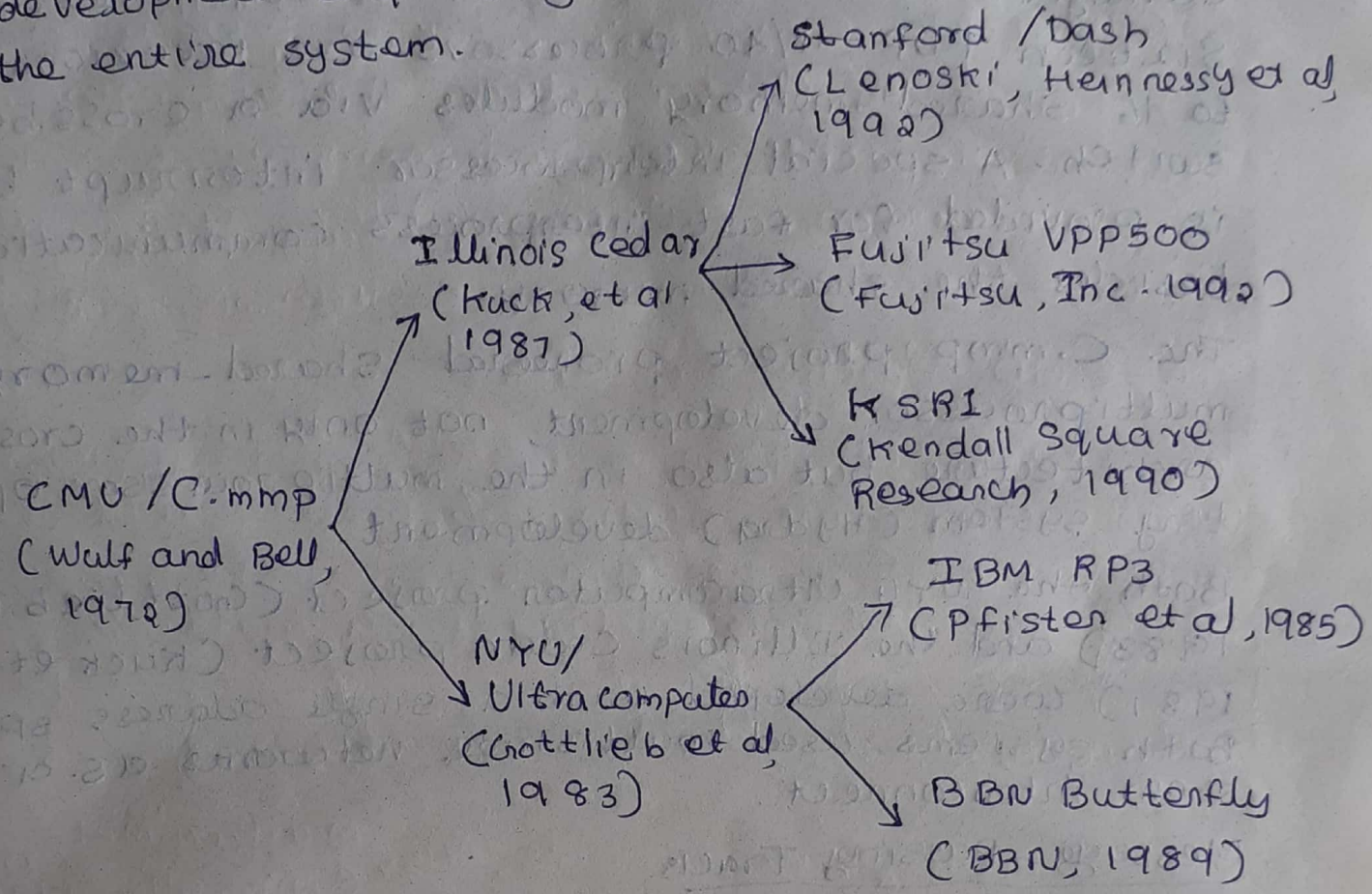
1. Multiple-Processor Tracks
2. Multivector and SIMD Tracks.
3. Multithread and Dataflow Tracks.

Multiple-Processor Tracks.

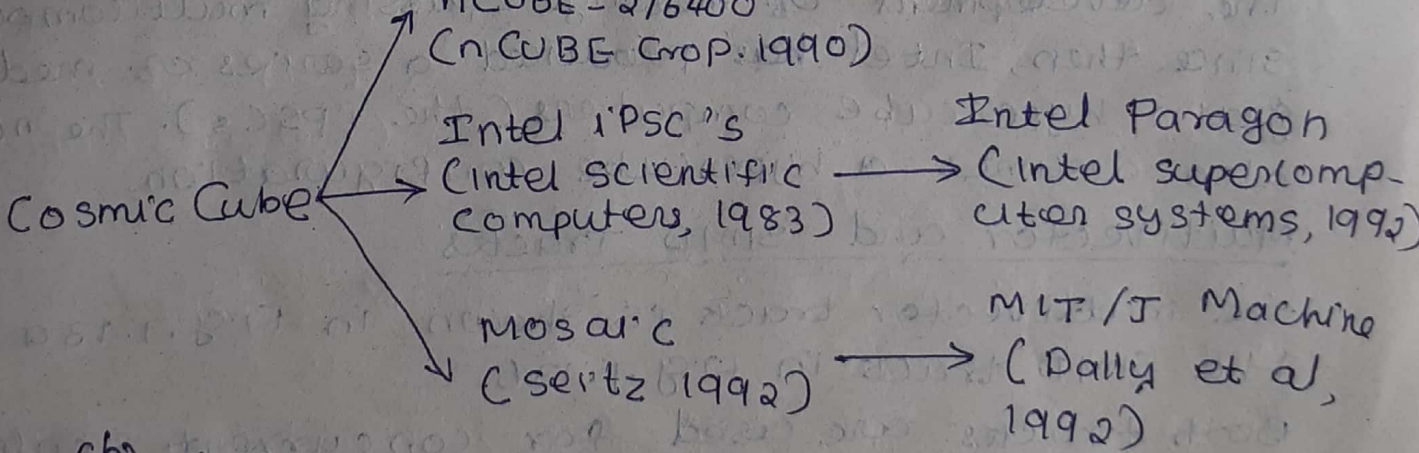
A multiple-processor system can be either a shared-memory multiprocessor or a distributed-memory multicomputer.

Shared-Memory Track

Figure 1.12a shows a track of multiprocessor development employing a single address space in the entire system.



(a) Shared-memory track



(b) Message-passing track

Figure 1.12 Two multiple-processor tracks with and without shared memory.

The track started with the C.mmp system developed at Carnegie-Mellon University (Wulf and Bell, 1972). The C.mmp system developed at Carnegie-Mellon was an UMA multiprocessor. Sixteen PDP 11/40 processors are interconnected to 16 shared-memory modules via a crossbar switch. A special interprocessor interrupt bus is provided for fast interprocess communication, besides the shared memory.

The C.mmp project pioneered shared-memory multiprocessor development, not only in the crossbar architecture but also in the multiprocessor operating system (Hydra) development.

Both the NYU Ultracomputer project (Gottlieb et al., 1983) and the Illinois Cedar project (Kuck et al., 1987) were developed with a single address space. Both systems used multistage networks as a system interconnect.

Message-Passing Tracks

The Cosmic Cube (Seitz et al., 1981) pioneered the development of message-passing multicomputers. Since then, Intel has produced a series of medium-grain hypercube computers (the iPSCs). The nCUBE2 also assumes a hypercube configuration.

Multivector and SIMD Tracks

The multivector track is shown in Fig. 1.13a, and the SIMD track in Fig. 1.13b.

Both tracks are used for concurrent scalar/vector processing.

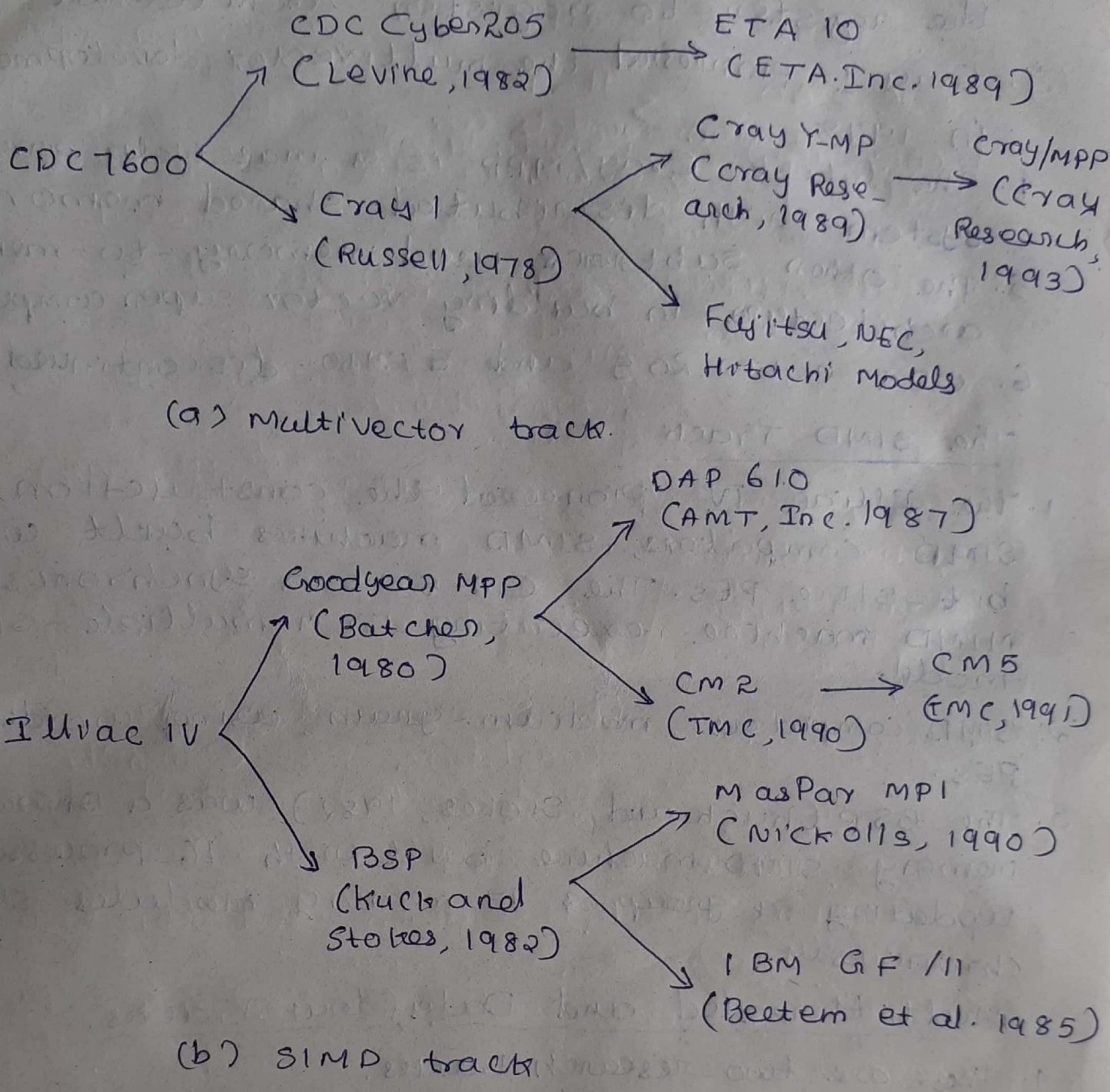


Figure 1.13 Multivector and SIMD tracks.

Multivector Track

These are traditional vector supercomputers.

1. The CDC 7600 was the first vector dual-processor system.
2. The cray and Japanese supercomputers all followed

the register-to-register architecture.

3. Cray 1 pioneered the multivector development in 1978.
4. The latest Cray/MPP is a massively parallel system with distributed shared memory.
5. The other subtrack used memory-to-memory architecture in building vector super computers.
6. CDC Cyber 205 and ETA10 - discontinued now.

The SIMD Tracks

1. The Illiac IV pioneered the construction of SIMD computers, SIMD machines built with bit-slice PEs. The CM-5 is a synchronized MIMD machine executing in a multiple-SIMD mode.
2. SIMD computers, medium-grain, using wordwide PEs.
3. The BSP (Kuck and Stokes, 1982) was a shared-memory SIMD machine built with 16 processors updating a group of 17 memory modules synchronously.

Multithreaded and Dataflow Tracks.

1. These are two research tracks.
2. The term multithreading implies that there are multiple threads of control in each processor.

As shown in Fig. 1.14a, the multithreading idea was pioneered by Burton Smith (1978) in the HEP system which extended the concept of scoreboarding of multiple functional units in the CDC 6400.

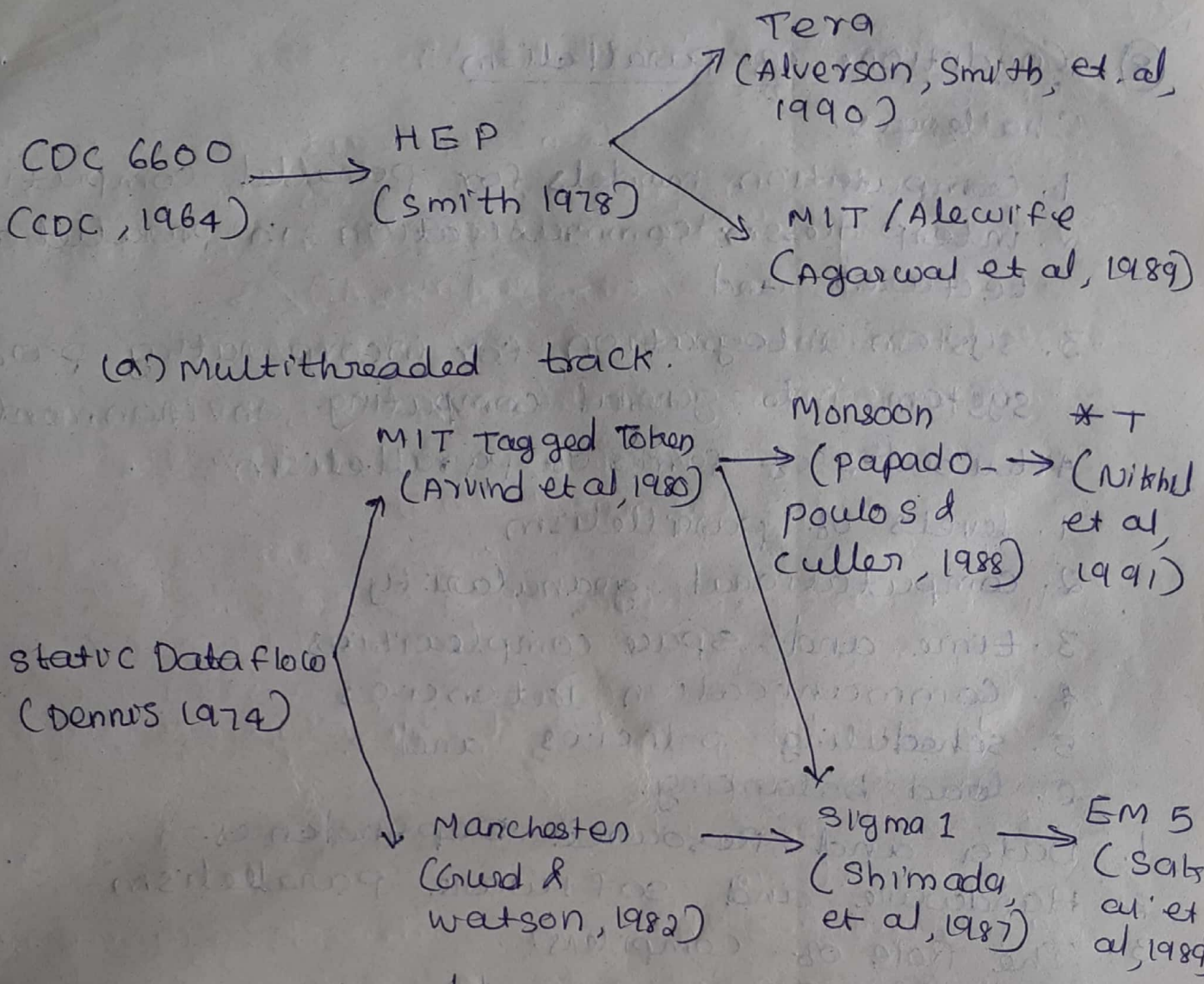


Figure 1.14 Multithreaded and dataflow tracks.

The Dataflow Track

Use dataflow mechanism instead of a control-flow mechanism as in von Neumann machines to direct the program flow. Fine-grain, instruction-level parallelism is exploited in dataflow computers.

The dataflow concept was pioneered by Jack Dennis (1974) with a "static" architecture. Dynamic-tagged-token and manchester,

(8) Conditions of Parallelism

Challenges:

1. computation models for parallel computing
2. interprocessor communication in parallel architectures, and
3. system integration for incorporating parallel systems into general computing environments.

Attributes of forms of parallelism:

1. levels of parallelism,
2. computational granularity,
3. time and space complexities,
4. communication latencies,
5. scheduling policies, and
6. load balancing.

1. Data and Resource Dependences
2. Hardware and software parallelism
3. The Role of compilers.

Data and Resource Dependences

1. Each segment of a program need to be independent of the other segments.
2. Use a dependence graph to describe the relations.

nodes - program statements (instructions)
directed edges - ordered relations among the statements.

Data Dependence

The ordering relationship between statements is indicated by the data dependence.

Five types of data dependence are

1. Flow dependence.

A statement S2 is flow-dependent on statement S1 if an execution path exists from S1 to S2 and if at least one output of S1 feeds in as input to S2. Flow dependence is denoted as S1 → S2.

2. Antidependence

Statement S2 is antidependent on statement S1 if S2 follows S1 in program order and if the output of S2 overlaps the input to S1.

A direct arrow crossed with a bar as in S1 ↛ S2 indicates antidependence from S1 to S2.

3. Output dependence

Two statements are output-dependent if they produce the same output variable. S1 ⊖→ S2 indicates output dependence from S1 to S2.

4. I/O dependence

Read and write are I/O statements. I/O dependence occurs not because the same variable is involved but because the same file is referenced by both I/O statements.

5. Unknown dependence

The dependence relation between two statements cannot be determined in the following situations:

i. The subscript of a variable is itself subscribed (indirect addressing).

ii. The subscript does not contain the loop index variable.

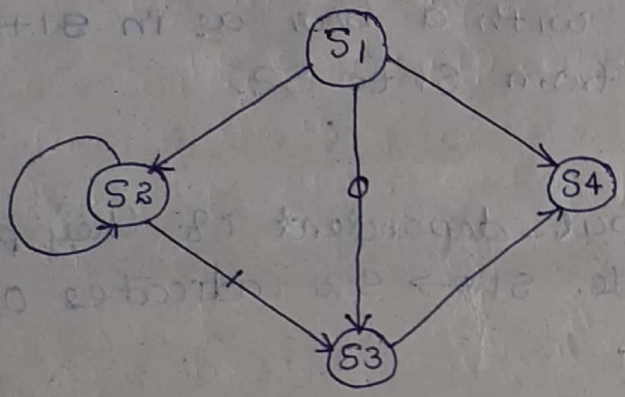
iii. A variable appears more than once with subscripts having different coefficients of the loop variable.

iv. The subscript is nonlinear in the loop index variable.

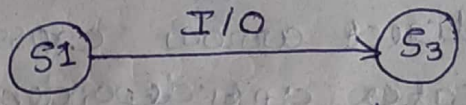
Example:

Consider the following code fragment of four instructions:

S1: Load R1, A /R1 ← Memory(A)/
S2: Add R2, R1 /R2 ← (R1) + (R2)/
S3: Move R1, R3 /R1 ← (R3)/
S4: Store B, R1 /Memory(B) ← (R1)/



(a) Dependence graph



(b) I/O dependence caused by accessing the same file by the read and write statements.

Figure 1.15 Data and I/O dependences in the program

As illustrated in Fig. 1.15(a), S2 is flow-dependent on S1 because the variable A is passed via the register R1. S3 is anti-dependent on S2 because of potential conflicts in register content in R1. S3 is output-dependent on S1 because they both modify the same register R1. other data dependence relationships can be similarly revealed on a pairwise basis. Dependence is a partial ordering relation; that is, the members of not every pair of statements are related.

Statements S2 and S4 in the above program are totally independent.

Consider a code fragment involving I/O operations.

```
S1 :  Read (4), A(I)   /Read array A from tape unit 4/
S2 :  Rewind (4)       /Rewind tape unit 4/
S3 :  Write (4), B(I)  /Write array B into tape unit 4/
S4 :  Rewind (4)       /Rewind tape unit 4/
```

As shown in Fig. 2.1b, the read/write statements S1 and S3, are I/O-dependent on each other because they both access the same file from tape unit 4. The above data dependence relations should not be arbitrarily violated during program execution. Otherwise, erroneous results may be produced with changed program order. The order in which statements are executed in a program is often well defined. Repetitive runs should produce identical results.

On a multiprocessor system, the program order may or may not be preserved, depending on the memory model used.

Control Dependence

This refers to the situation where the order of execution of statements cannot be determined before run time. Conditional statements will not be resolved until run time.

Different paths taken after a conditional branch may introduce or eliminate data dependence among instructions.

Example.
The successive iterations of the following loop are control-independent:

```
Do 20 I = 1, N
    A(I) = C(I)
    IF (A(I).LT.0) A(I) = 1
20 Continue
```

The following loop has control-dependent iterations:

```
Do 10 I = 1, N
    IF (A(I-1).EQ.0) A(I) = 0
10 continue
```

control dependence often prohibits parallelism from being exploited. compiler techniques are needed to get around the control dependence in order to exploit more parallelism.

Resource Dependence

Demands the independence of the work to be done. Resource dependence is concerned with the conflicts in using 1. shared resources such as integer units, floating-point units, registers and memory areas.
When the conflicting resource is an ALU, we call it ALU dependence.
If the conflicts involve workplace storage, we call it storage dependence: each task must work on independent storage locations or use protected access to shared writable data.

Bernstein's Conditions

In 1966, Bernstein revealed a set of conditions based on which two processes can execute in parallel.

A process is a software entity

i. I_i of a process P_i

The set of all input variables needed to execute the process. Fetched from memory or registers.

ii. O_i - the output set

The set of all output variables generated after execution of the process P_i. Results to be stored in working registers or memory locations.

Let P_1 & P_2 two processes

I_1 & I_2 ~~output sets~~ Input sets

O_1 & O_2 Output sets

$P_1 \parallel P_2$: The processes can execute in parallel. If they are independent and do not create confusing results.

$$\left.\begin{aligned} I_1 \cap O_2 &= \phi \\ I_2 \cap O_1 &= \phi \\ O_1 \cap O_2 &= \phi \end{aligned}\right\} \text{Bernstein's conditions.}$$

I_i - read set or the domain of P_i.

O_i - write set or the range of a process P_i.

iii. In terms of data dependences, Bernstein's conditions simply imply that two processes can execute in parallel if they are flow-independent, anti-independent, and output-independent.

Example

Detection of parallelism in a program using Bernstein's conditions.

Consider the simple case in which each process is a single HLL statement. Detect the parallelism embedded in the following five instructions labeled P_1, P_2, P_3, P_4, and P_5 in program order.

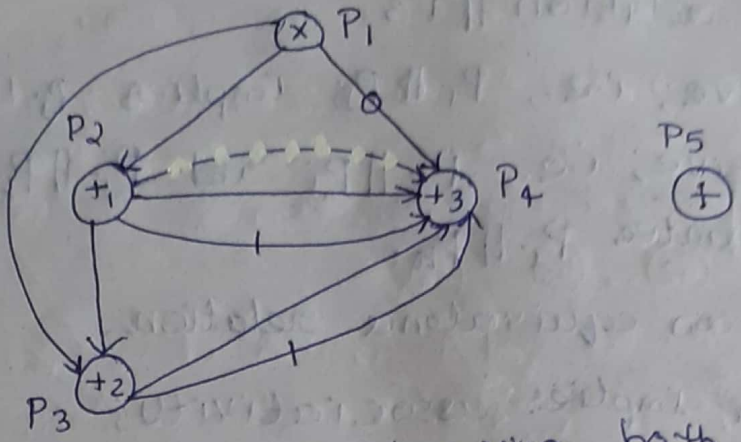
$P_1 : C = D \times E$
$P_2 : M = G + C$
$P_3 : A = B + C$
$P_4 : C = L + M$
$P_5 : F = G \div E$

Ans: Assume that each statement requires one step to execute. No pipelining is considered here. The dependence graph shown in Fig. 1.16a demonstrates flow dependence as well as resource dependence.

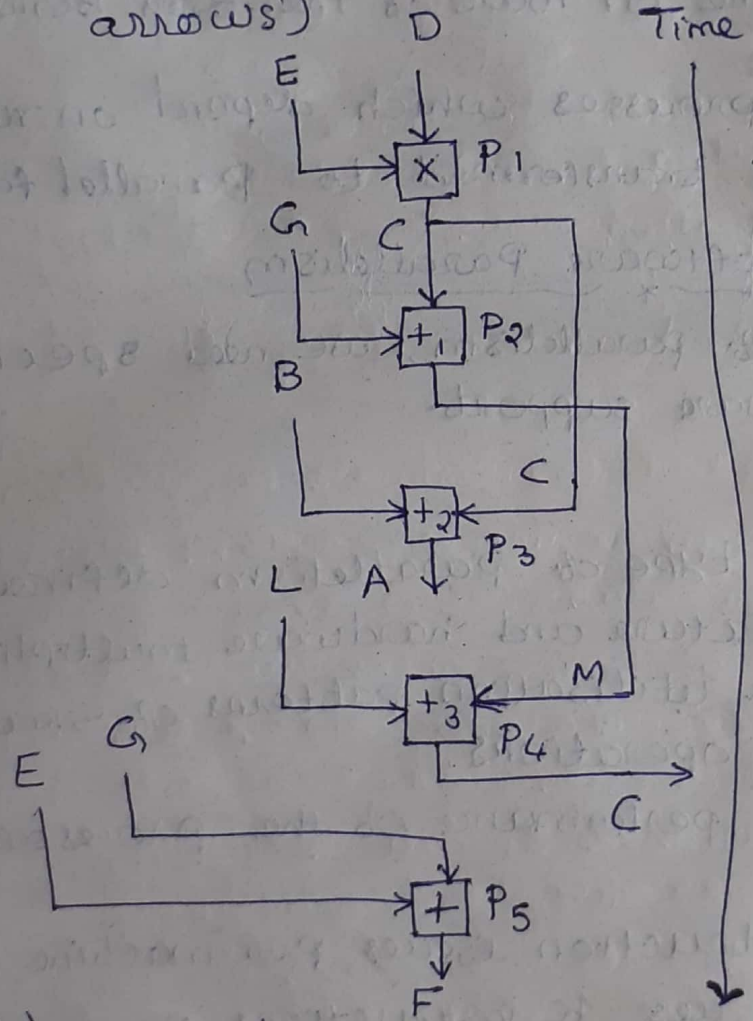
In sequential execution, five steps are needed as shown in Fig. 1.16b.

If two adders are available simultaneously, the parallel execution requires only three steps in Fig. 1.16c. Pairwise, there are 10 pairs of statements to check against Bernstein's conditions. Only 5 pairs, $P_1 \| P_5$, $P_2 \| P_3$, $P_2 \| P_5$, $P_5 \| P_3$, and $P_4 \| P_5$, can execute in parallel as in Fig. 1.16a if there are no resource conflicts.

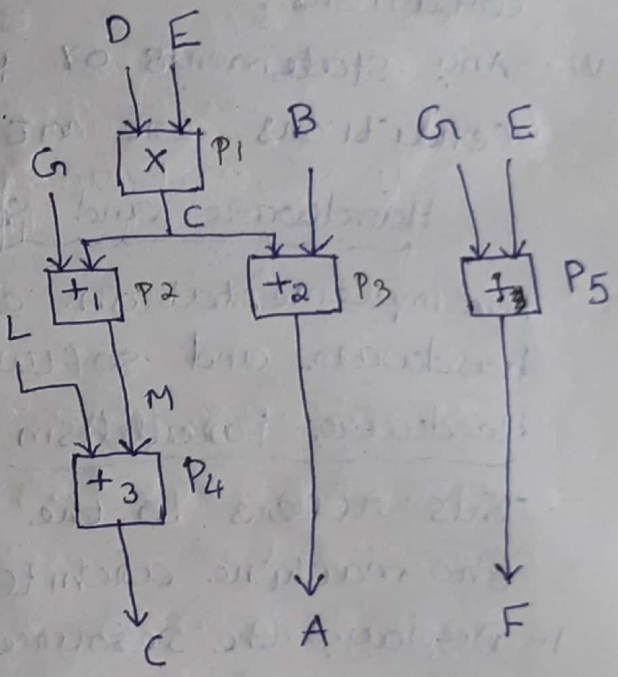
Collectively, only $P_2 \| P_3 \| P_5$ is possible because $P_2 \| P_3$, $P_3 \| P_5$, and $P_5 \| P_2$ are all possible.



(a) A dependence graph showing both data dependence (solid arrows) and resource dependence (dashed arrows)



(b) Sequential execution in five steps, assuming one step per statement (no pipelining)



(c) Parallel execution in three steps, assuming two adders are available per step.

Figure 1.16 Detection of parallelism in the program.

Parallelism relation $\|$ is

i. commutative; i.e., $P_i \| P_j$ implies $P_j \| P_i$.

ii. Not transitive; i.e., $P_i \| P_j$ and $P_j \| P_k$ do not necessarily guarantee $P_i \| P_k$.

iii. $\|$ is not an equivalence relation.

iv. $P_i \| P_j \| P_k$ implies associativity;

i.e., $(P_i \| P_j) \| P_k = P_i \| (P_j \| P_k)$

v. Violations of any one or more of the ~~three~~ Bernstein's conditions.

vi. Any statements or processes which depend on run-time conditions are not transformed to Parallel form.

Hardware and Software Parallelism

For implementation of parallelism, we need special hardware and software support.

Hardware Parallelism

This refers to the type of parallelism defined by the machine architecture and hardware multiplicity.

1. Displays the resource utilization patterns of simultaneously executable operations.
2. Indicate the peak performance of the processor resources.
3. The number of instruction issues per machine cycle. If a processor issues k instructions per machine cycle, then it is called a k-issue processor.

Example: Intel i960CA - three-issue processor with one arithmetic, one memory access, and

- one branch instruction issued per cycle.

IBM RISC/System 6000 is a four-issue processor capable of issuing one one arithmetic, one memory access, one floating-point, and one branch operation per cycle.

4. A multiprocessor system built n k-issue processors should be able to handle a maximum number of nk threads of instructions simultaneously.

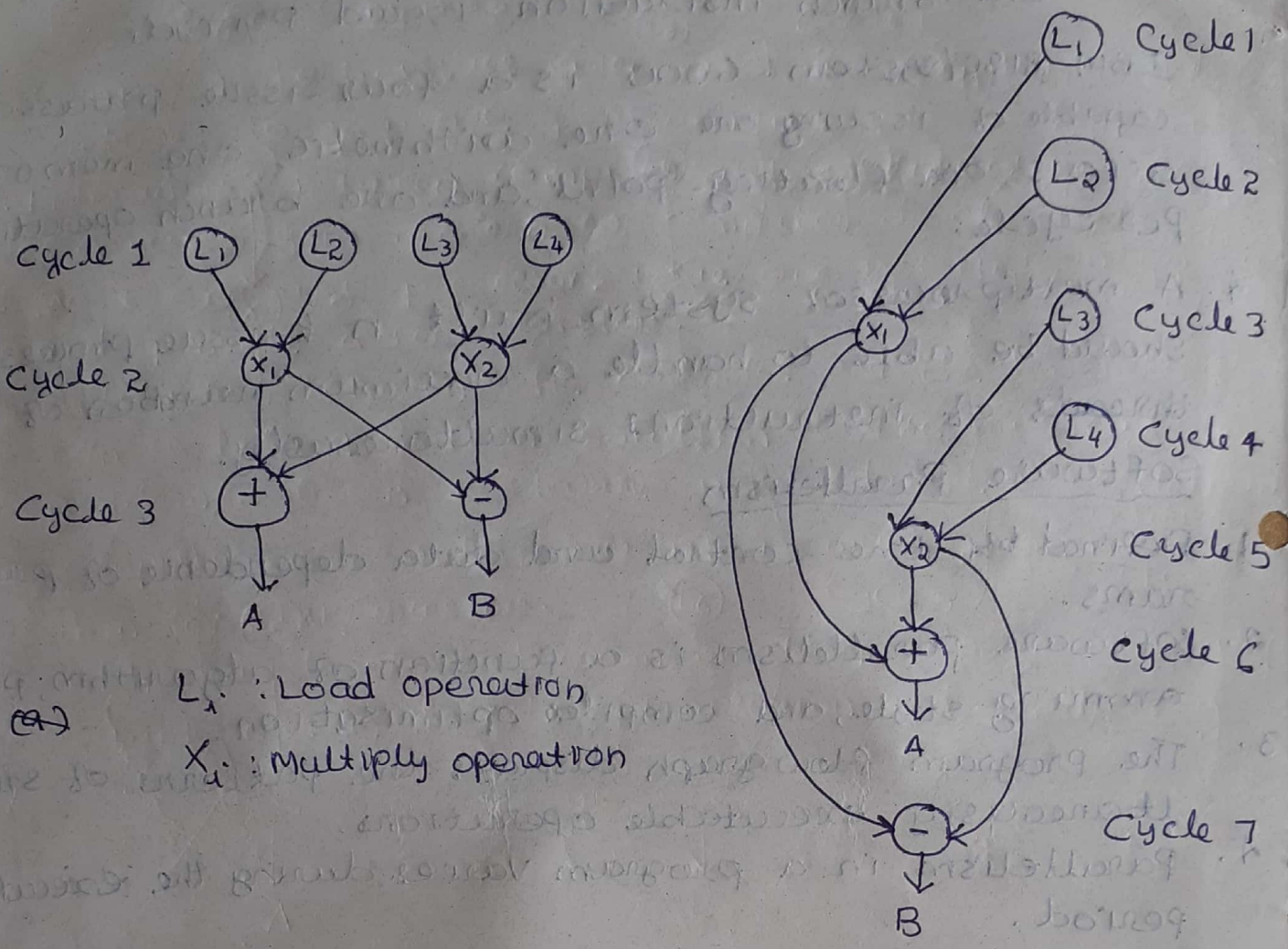
Software Parallelism

1. Defined by the control and data dependence of programs.
2. Software parallelism is a function of algorithm, programming style, and compiler optimisation.
3. The program flow graph displays the patterns of simultaneously executable operations.
4. Parallelism in a program varies during the execution period.

Example

Mismatch between software parallelism and hardware parallelism.

Consider the example program graph in Fig 1.17a. There are eight instructions (four loads and four arithmetic operations) to be executed in three consecutive machine cycles. Four load operations are performed in the first cycle, followed by two multiply operations in the second cycle and two add/subtract operations in the third cycle. Therefore, the parallelism varies from 4 to 2 in three cycles. The average software parallelism is equal to $8/3 = 2.67$ instructions per cycle in this example



(a) Software parallelism (b) Hardware parallelism

Figure 1.17 Executing an example program by a two-issue superscalar processor.

Consider execution of the same program by a two-issue processor which can execute one memory access (load or write) and one arithmetic (add, subtract, multiply, etc) operation simultaneously. With this hardware restriction, the program must execute in seven machine cycles as shown in Fig 1.17b. Therefore, the hardware parallelism displays an average value of $8/7 = 1.14$ instructions executed per cycle. This demonstrates a mismatch between the software parallelism and the hardware parallelism.

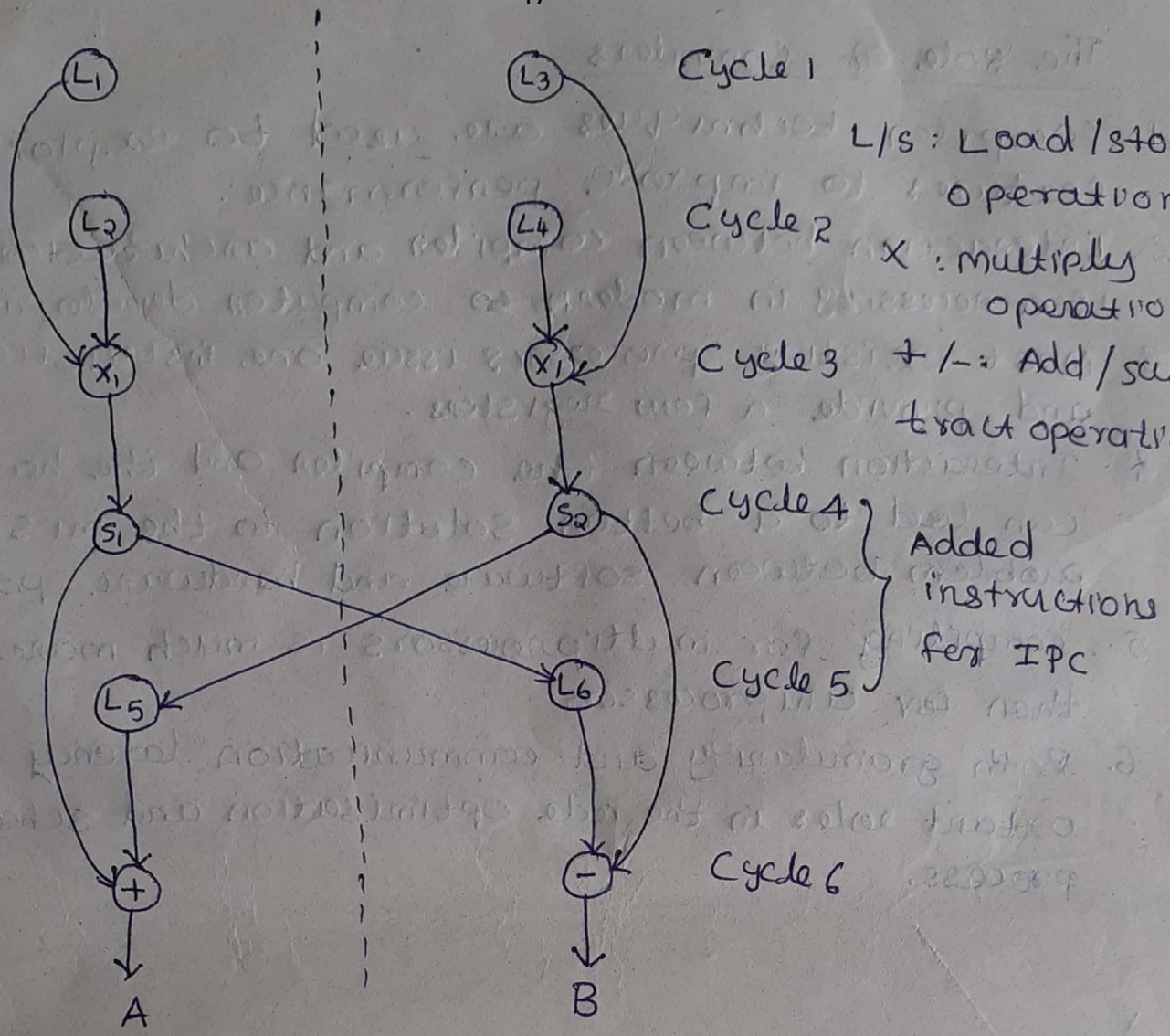


Figure 1.18 Dual-processor execution of the program in Fig. 1.17a.

Two important parallel programming.

1. control parallelism.
2. Data parallelism.

Two or more operations performed simultaneously in control parallelism.

The same operation is performed over many data elements by many processors simultaneously.

Solution for hardware and software parallelism:

1. To develop compilation support, and
2. Through hardware redesign for more efficient exploitation by an intelligent compiler.

The Role of Compilers

1. Compiler techniques are used to exploit hardware features to improve performance.
2. Interaction between compiler and architecture design is a necessity in modern computer development.
3. Most existing processors issue one instruction per cycle and provide a few registers.
4. Interaction between the compiler and the hardware can lead to a better solution to the mismatch problem between software and hardware parallelism
5. Compiling for multiprocessors is much more involved than for uniprocessors.
6. Both granularity and communication latency play important roles in the code optimization and scheduling process.

Module II

Processors and memory hierarchy

Modern processor technology and the supporting memory hierarchy.

Advanced Processor Technology

Architectural families of modern processors with the underlying microelectronics/packaging technologies.

1. Design Space of Processors
2. Instruction-set Architectures
3. CISC Scalar Processors
4. RISC Scalar Processors

Scalar and Vector processors – for numerical computation.
Symbolic processors – for AI applications:

Design Space of Processors

i. The Design Space
ii. Instruction Pipelines
iii. processors and Coprocessors

Various processor families can be mapped onto a coordinated space of Clock rate versus cycles per instruction (CPI) as shown in Fig. 2.1.

Trends:

a) The clock rates of various processors are gradually moving from low to higher speeds toward the right of the design space.
b) Processor manufacturers are trying to lower the CPI rate using hardware and software approaches.

The Design Space

a) Complex-instruction-set computing (CISC): Intel i486, M68040, VAX/8600, IBM 390, etc. Clock rate: 33 to 50 MHz. CPI varies from 1 to 20. So at the upper left of the design space. conventional, microprogrammed control.

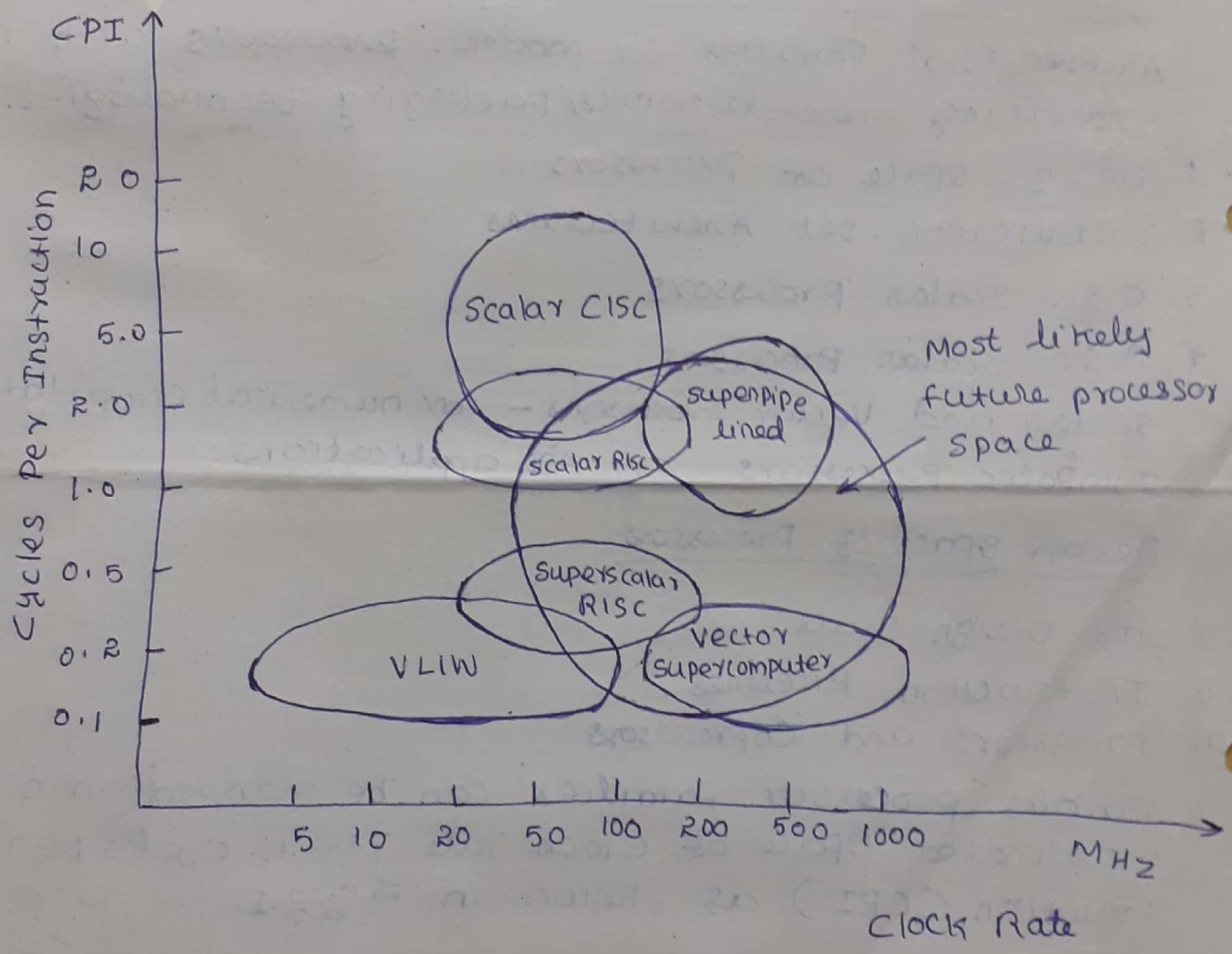


Figure 2.1 Design space of modern processor families.

b) Today's Reduced-instruction-set computing (RISC): eg: Intel i860, SPARC, MIPS R3000, IBM RS/6000 etc.
clock rate: 20 to 120 MHz
CPI: 1 to 2 cycles (hardwired control)

c) Superscalar processors (special subclass of RISC)
   i) multiple instructions to be issued simultaneously during each cycle.
   ii) CPI of a superscalar processor lower than that of a generic scalar RISC processor.
   iii) The clock rate of superscalar processors matches that of scalar RISC processors.

d) The Very long instruction word (VLIW)
   i) Uses more functional units than that of a superscalar processor.
   ii) CPI of VLIW processor further lowered.
   iii) Use very long instructions (256 to 1024 bits per instruction)
   iv) Implemented with microprogrammed control.
   v) The clock rate is slow with the use of read-only memory (ROM).

e) Superpipelined processors
   i) Use multiphase clocks.
   ii) Increased clock rate. (100 to 500 MHz)

f) vector supercomputers
   i) super pipelined
   ii) Use multiple functional units for concurrent scalar and vector operations.

Instruction Pipelines

1. Four phases of the typical execution cycle
   1. fetch, 2. decode, 3. execute, 4. write-back.
2. Instruction phases are executed by an instruction pipeline as demonstrated in Fig. 2.2a.
3. pipeline: Receives successive instructions from its input end and executes them in a streamlined, overlapped fashion as they flow through.

4. pipeline cycle

The time required for each phase to complete its operation, assuming equal delay in all phases (pipeline stages).

5. Definitions associated with instruction pipeline operations:

i. Instruction pipeline cycle

The clock period of the instruction pipeline

ii. Instruction issue latency

The time required between the issuing of two adjacent instructions.

iii. Instruction issue rate

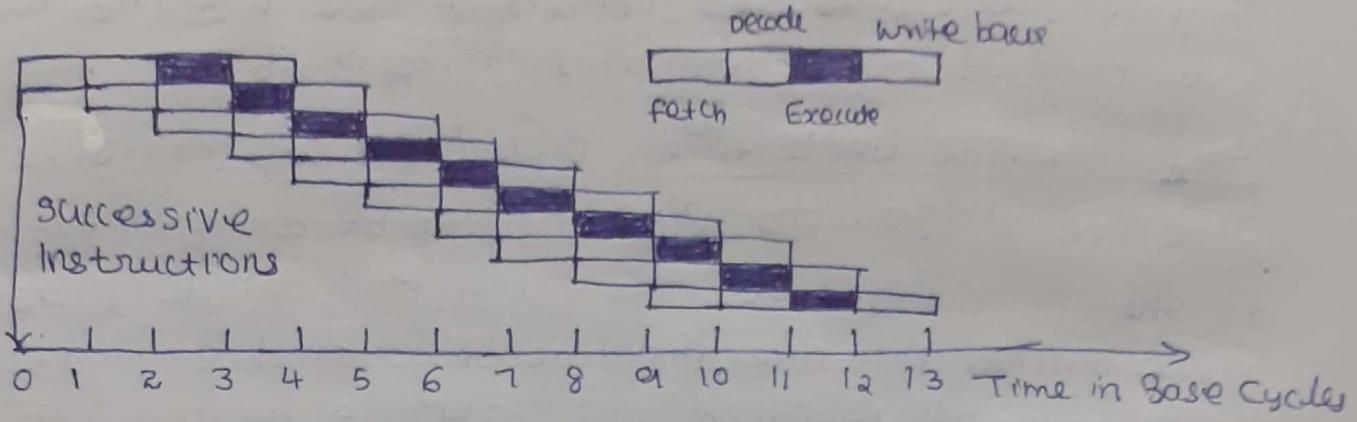
The number of instructions issued per cycle. Degree of a superscalar processor.

iv. Simple operation latency

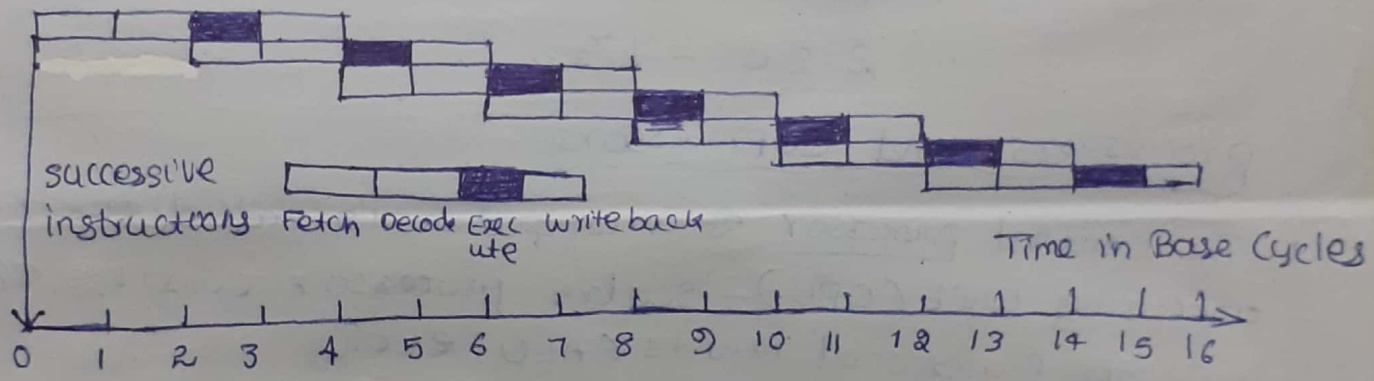
Simple operations are major instructions such as integer adds, loads, stores, branches, moves, etc.

v. Resource conflicts - This refers to the situation where two or more instructions demand use of the same functional unit at the same time.

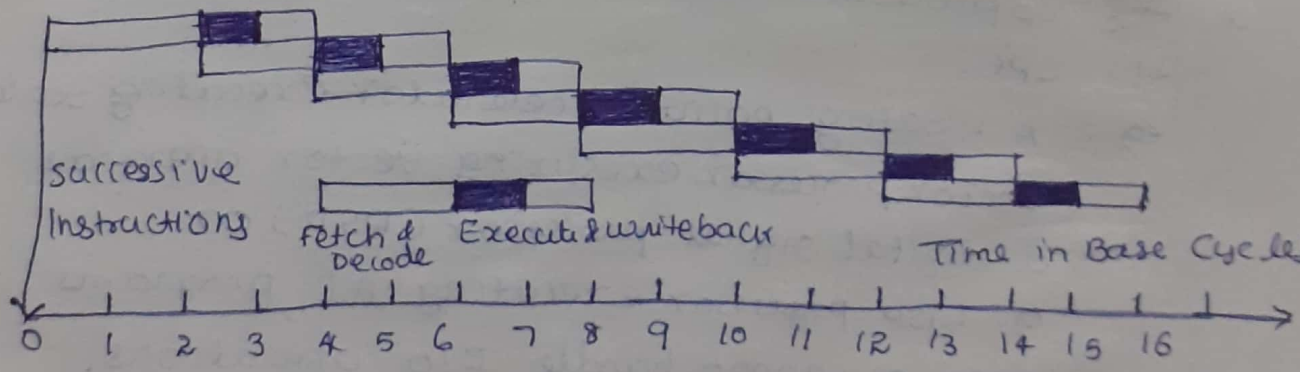
A base scalar processor is defined as a machine with one instruction issued per cycle. A one-cycle latency for a simple operation, and a one-cycle latency between instruction issues. The instruction pipeline can be fully utilized if successive instructions can enter it continuously at the rate of one per cycle, as shown in Fig. 2.29.



(a) Execution in a base scalar processor.



(b) Underpipelined with two cycles per instruction issue



(c) Underpipelined with twice the base cycle

Figure 2.2 Pipelined execution of successive instructions in a base scalar processor and in two underpipelined cases.

The instruction issue latency can be more than one cycle for various reasons. If the instruction issue latency is two cycles per instruction, the pipeline can be underutilized, as demonstrated in Fig. 2.2b.

The pipeline cycle time is doubled by combining pipeline stages, the pipeline can be underpipelined as shown in Fig. 2.2c. The fetch and decode phases are combined into one pipeline stage, and execute and write-back are combined into another stage.

CPI for Fig. 2.2a - 1

" " " 2.2b - 2

" " " 2.2c - 1½.

processors and coprocessors

1. The central processor of a computer is called the central processing unit (CPU) - scalar processor consists of multiple functional units (ALU, etc.)
2. Coprocessor is the unit attached to the CPU.
   i. The coprocessor executes instructions dispatched from the CPU.
      eg: A floating-point accelerator executing scalar data,
      Vector processor executing vector operands,
      a digital signal processor (DSP) or
      a Lisp processor executing AI programs.
   ii. Coprocessors cannot handle I/O operations.
   iii. Coprocessors cannot be used alone.
   iv. Coprocessors also called as attached processors or slave processors.

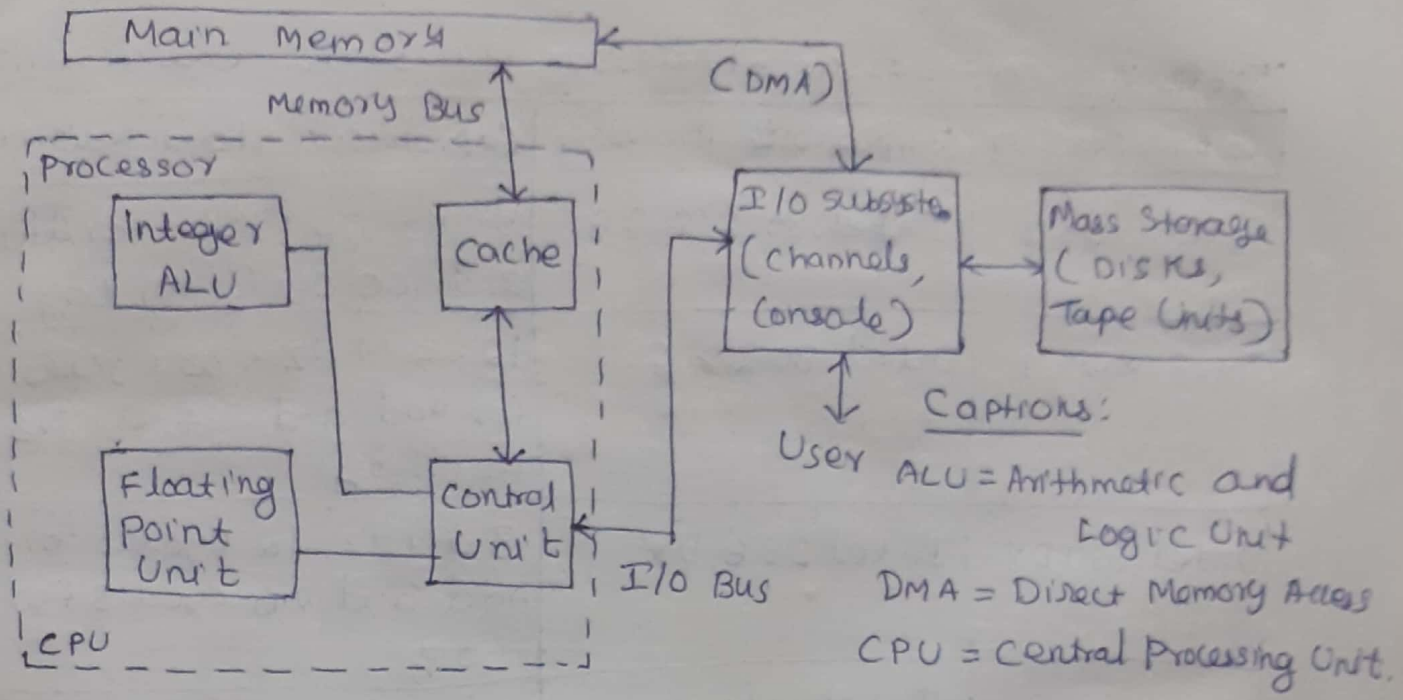
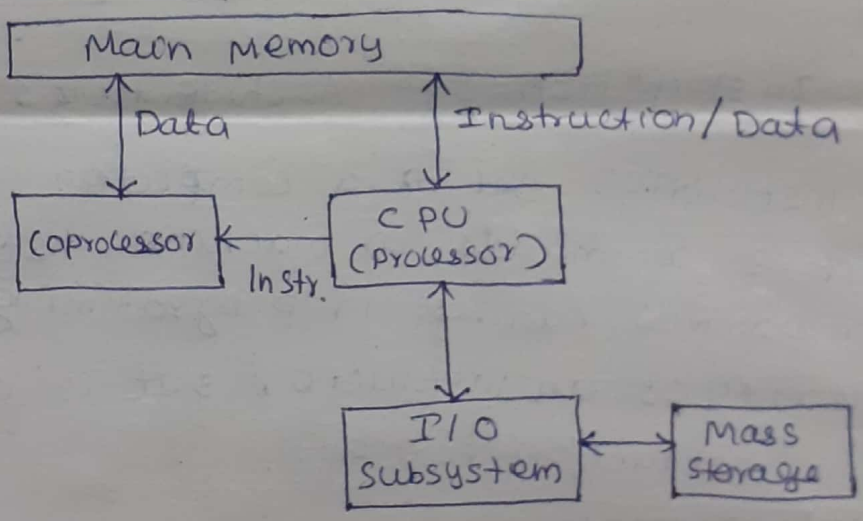


Fig (a) CPU with built-in floating point unit.



(b) CPU with an attached coprocessor

Figure 2.3 Architectural models of a basic scalar computer system.

Table 2.1 lists some processor-coprocessor pairs developed in recent years to speed up numerical computations.

Table 2.1 coprocessor - Processor pairs and characteristics.

coprocessor	Compatible Processor	Coprocessor characteristics
Intel 8087	Intel 8086/8088	5 MHz, 70 cycles for Add and 700 cycles for Log
Intel 80287	Intel 80286	12.5 MHz, 30 cycles for Add and 264 cycles for Log
Intel 387DX	Intel 386 DX	33 MHz, 12 cycles for Add and 210 cycles for Log
Intel i486	Intel i486 (the same chip	33 MHz, 8 cycles for Add and 571 cycles for Log

Instruction-Set Architectures

1. The instruction set of a computer specifies the primitive commands or machine instructions that a programmer can use in programming the machine.
2. complexity of an instruction set is due to
   i. the instruction formats,
   ii. data formats
   iii. addressing modes,
   iv. general-purpose registers,
   v. Opcode specifications and
   vi. flow control mechanisms used.

Two types of instruction sets:

1. complex Instruction sets (CISC)
2. Reduced instruction sets (RISC)

Complex Instruction sets (CISC)

1. Early instruction set was simple because the high cost of hardware.
2. The hardware cost has dropped and software cost has gone up.
3. More functions have been built into the hardware which made the instruction set very large and very complex.

CISC - 120 to 350 instructions
general-purpose registers - 8 to 24
addressing modes - more than a dozen.

Reduced Instruction sets (RISC)

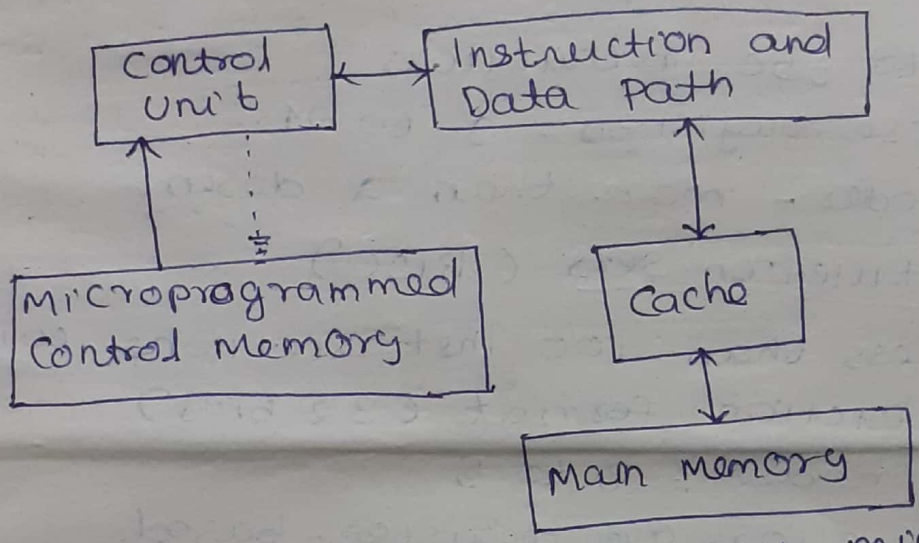
1. Contains less than 100 instructions with a fixed instruction format (32 bits).
2. Addressing modes - 3 to 5
3. Most instructions are register-based.
4. Memory access is done by load / store instructions only.
5. Most instructions execute in one cycle with hardwired control.
6. The entire processor is implementable on a single VLSI chip.
7. Higher clock rate and a lower CPI

Architectural Distinctions

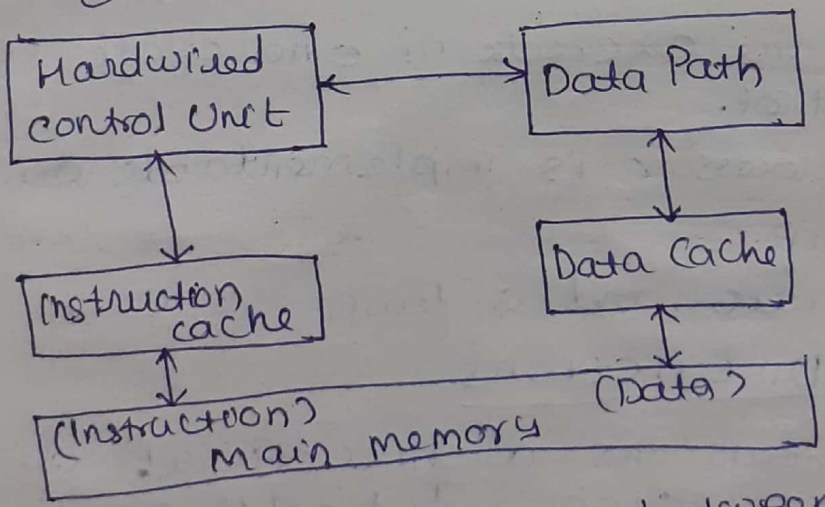
Figure 2.4 shows the architectural distinctions between modern CISC and traditional RISC.

1. conventional CISC architecture uses a unified cache for holding both instructions and data
2. share the same data/instruction path.

3. In a RISC processor, seperate instruction and data caches are used with different access paths.
4. CISC processors may also use split codes.
5. CISC - microprogrammed control.
6. RISC - hardwired control.
7. ROM is needed in CISC
8. CISC may also use hardwired control.



(a) The CISC architecture with micro-programmed control and unified cache.



(b) The RISC architecture with hard-wired control and split instruction cache and data cache.

Figure 2.4 Distinctions between typical RISC and typical CISC processor architectures.

In Table 2.2 the following features are of RISC and CISC processors are compared.
1. instruction sets,
2. addressing modes,
3. register file and cache design,
4. clock rate and expected CPI, and
5. control mechanisms.

Table 2.2 Characteristics of CISC and RISC Architectures

Architectural characteristic	Complex Instruction set computer (CISC)	Reduced Instruction set Computer (RISC)
Instruction-set size and instruction formats	Large set of instructions with variable formats (16-64 bits per instruction).	Small set of instructions with fixed (32-bit) format and most register-based instructions.
Addressing modes	12-24.	Limited to 3-5
General-purpose registers and cache design	8-24 GPRs Unified cache	32-192 GPRs Split data and instruction cache.
Clock rate and CPI	33-50 MHz 2-15 CPI	50-150 MHz <1.5 CPI
CPU control	Most microcoded using control memory (ROM), but modern CISC also uses hardwired control.	Most hardwired without control memory.

CISC Scalar Processors

1. A scalar processor executes with scalar data.
2. A CISC scalar processor can be built either with a single chip or with multiple chips mounted on a processor board.

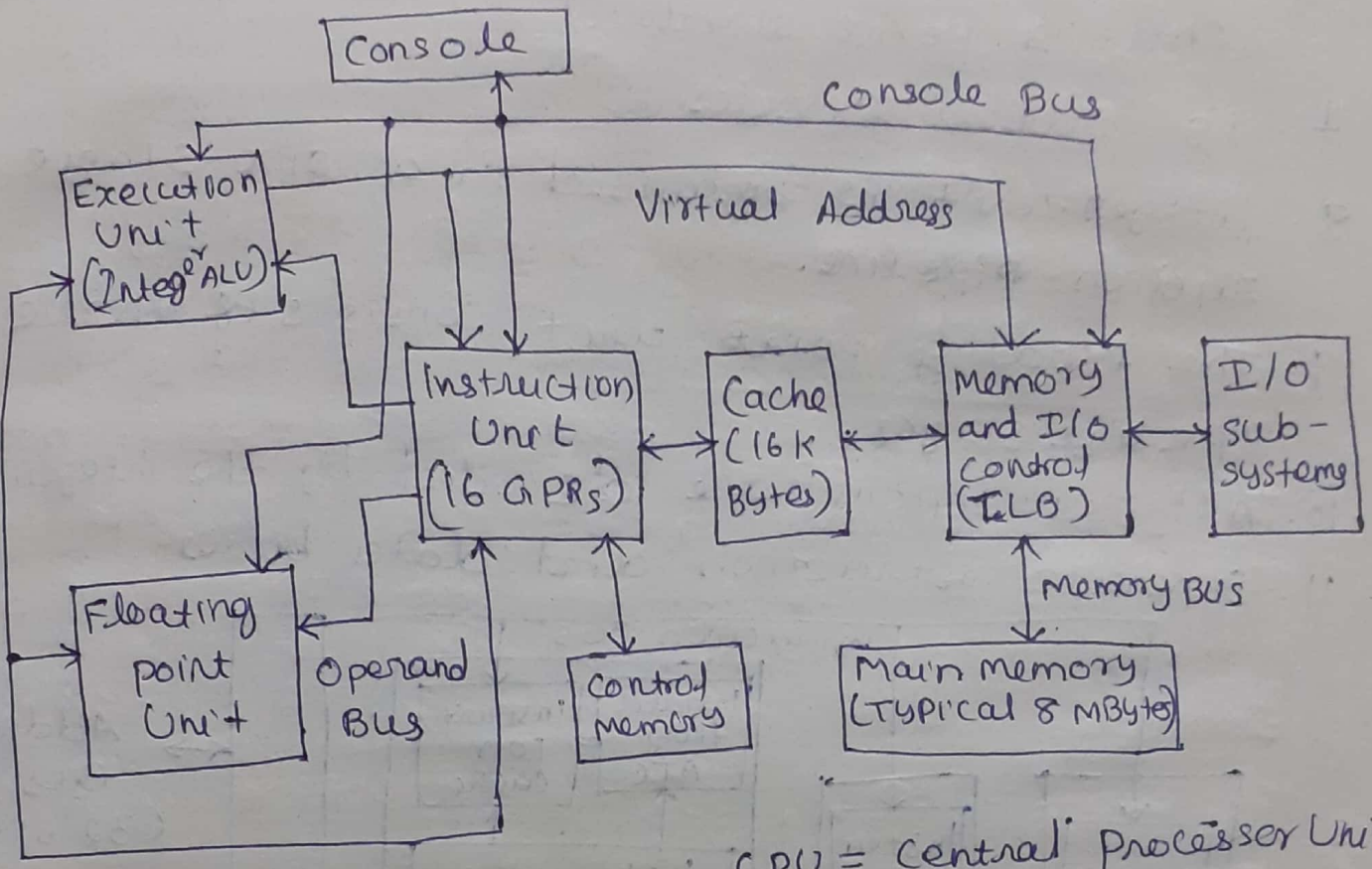
Representative CISC Processors

1. The VAX 8600 processors is built on a PC board.
2. The i486 and M68040 are single-chip microprocessors.

Example 2.1 The digital Equipment VAX 8600 processor architecture.

1. Introduced by Digital Equipment Corporation in 1985.
2. Microprogramined control.
3. 300 instructions
4. 20 different addressing modes.
5. VMS operating system.
6. SBI and Unibus (I/O)
7. The CPU
   i. Two functional units for concurrent execution of integer and floating-point instructions.
   ii. Unified cache - instruction and data.
   iii. 16 General purpose Registers
   iv. Instruction pipelining has been built with six stages in the VAX 8600
8. Translation lookaside buffer (TLB) - Physical address generation from a virtual address.
9. Both integer and floating-point units are pipelined.

10. CPI varies from 2 to 20 cycles.



CPU = Central Processor Unit
TLB = Translation Lookaside Buffer
GPR = General Purpose Register

Figure 2.5 The VAX 8600 CPU, a typical CISC processor architecture.

Example 2.2 The Motorola MC68040 microprocessor architecture.

1. Figure 2.6 shows the MC68040 architecture.
2. 100 instructions
3. 16 General-purpose registers
4. 4-KB data cache, and 4-KB instruction cache
5. Seperate memory management units (MMUs) supported by an address translation cache (ATC) like TLB.

6. 8 to 80 bits data format based on IEEE floating point standard
7. 18 addressing modes
8. Integer unit is organized in a six-stage instruction pipeline.
9. The floating-point unit consists of three pipeline stages.
10. All instructions are decoded by the integer unit.
11. Seperate instruction and data buses

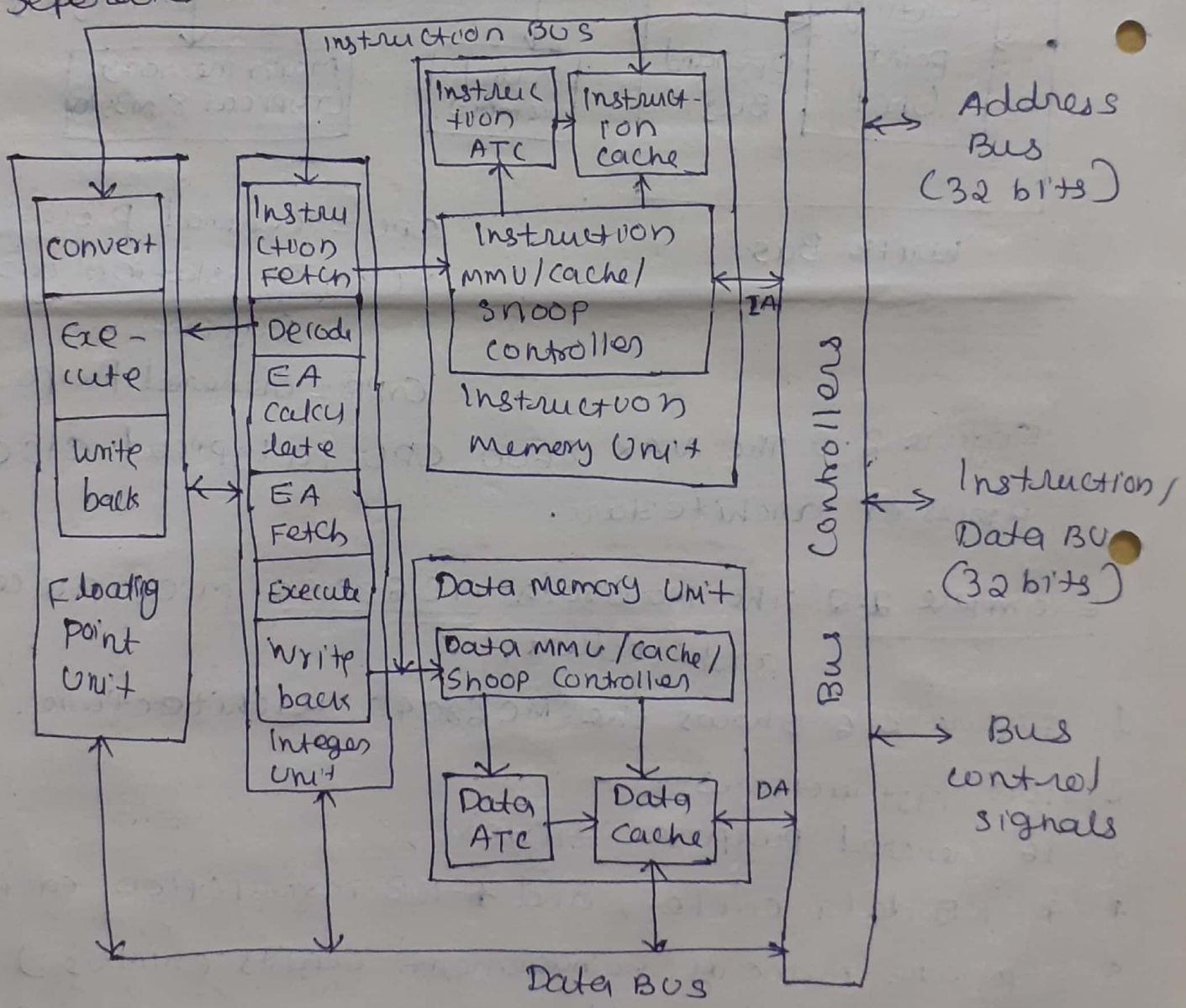


Figure 2.6 Architecture of the MC 68040 Processor

IA = Instruction Address
DA = Data Address
EA = Effective Address
ATC = Address Translation Cache
MMU = Memory Management Unit.

CISC Microprocesser Families.

1. 1971, Intel 4004 - first microprocesser based on a 4-bit ALU.
2. 8-bit 8008, 8080, & 8085.
3. 16-bit - 1978 - 8086, 8088, 80186 & 80286.
4. In 1985 - 80386 - 32 bit, 80486, 80586.
5. Motorola
   i. first 8-bit - MC 6800 - 1974
   ii. 16-bit - 68000 - 1979
   iii. 32-bit - 68020 - 1984.
   iv. MC 68030 & MC68040.

RISC scalar Processers.

1. Generic RISC processers are called scalar RISC
2. One instruction per cycle.
3. The RISC design gains its power by pushing some of the less frequently used operations into software.
4. The simplicity introduced with a RISC processer may lead to the ideal performance of the base scalar machine.

Representative RISC processors.

1. 4 RISC-base processors
   a) Sun SPARC
   b) Intel i860,
   c) Motorola M88100 and
   d) AMD 29000.

   } 32-bit instructions. 51 to 124 instructions.

SPARC - Scalable processor architecture.

The SPARC processor architecture contains 140 - 196 essentially a RISC integer unit (IU) implemented with 2 to 32 register windows.

Example

The Intel i860 processor architecture

1. In 1989, Intel Corporation introduced the i860 microprocessor.
2. 64-bit RISC processor
3. Single chip containing more than 1 million transistors.
4. Peak performance
   i. 80 Mflops single precision or
   ii. 60 Mflops double precision, or
   iii. 40 MIPS in 32-bit integer operations at a 40-MHz clock rate.
5. A schematic block diagram of major components is shown in Fig 2-7.
   i. 9 functional units interconnected by multiple data paths (from 32 to 128 bits).
   ii. address buses are 32-bit wide.

iii. data bus is 64 bits wide.
iv. Internal RISC integer ALU is only 32 bits wide.
v. Instruction cache has 4 KB organized as a two-way set-associative memory with 32B per cache block.
vi. Data cache is a two-way set-associative memory of 8KB. Transfers 128 bits per clock cycle at 40 MHz. Write-back policy is used. Bus control unit coordinates the 64-bit data transfer between the chip and the outside world.
vii. MMU
a) Implements protected 4 KB paged virtual memory of 2^{32} bytes via a TLB.
viii. RISC integer unit executes load, store, integer, bit and control instructions.
ix. Two floating-point units which can be used seperately or simultaneously under the coordination of the floating-point control unit.
(a) multiplier unit (b) adder unit.
x. Both the integer unit and the floating-point control unit can execute concurrently.
xi. The graphics unit executes integer operations corresponding to 8-, 16-, or 32-bit pixel data types.
a) Three-dimensional drawing in a graphics frame buffer with color intensity, shading, and hidden surface elimination.
b) The merge register is used only by vector integer instructions.

6. Executes 82 instructions
   i. 42 RISC integer,
   ii. 24 floating-point
   iii. 10 graphics, and
   iv. 6 assembler pseudo operations.

} execute in one cycle.

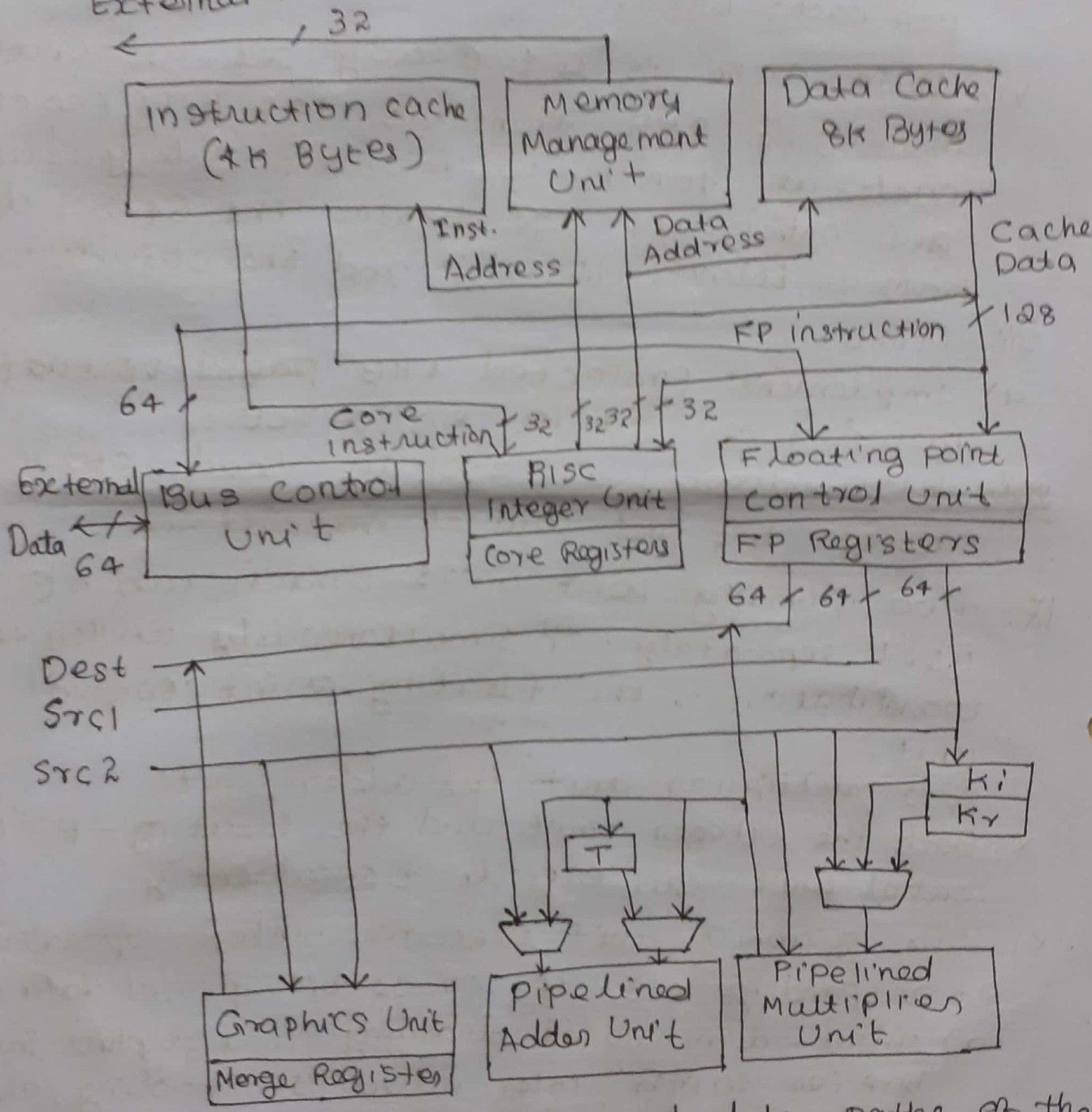


Fig. 2.7 Functional units and data paths of the Intel i860 RISC microprocessor.

SUPERSCALAR AND VECTOR PROCESSORS

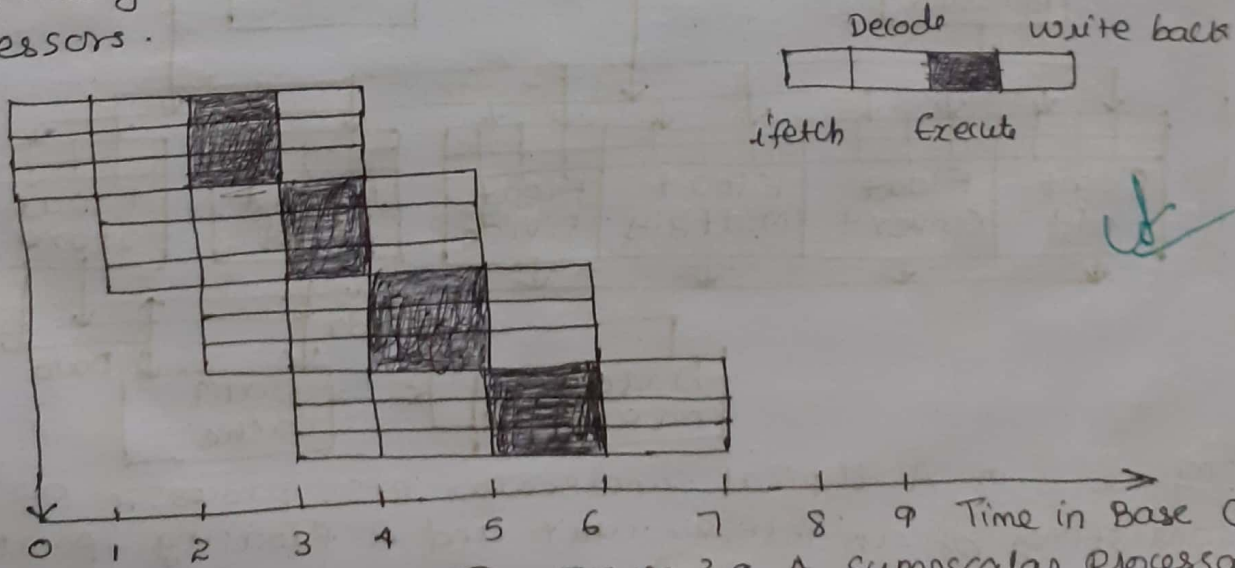
1. Multiple instruction pipelines are used.
2. Multiple instructions are issued per cycle and multiple results are generated per cycle.
3. A vector processor executes vector instructions on arrays of data.
4. Each instruction involves a string of repeated operations, which are ideal for pipelining with one result per cycle.

Superscalar Processors

1. More instruction-level parallelism in user programs.
2. Only independent instructions can be executed in parallel without causing a wait state.

pipelining in Superscalar Processors

The fundamental structure of a superscalar pipeline is illustrated in Fig. 2.8. This diagram shows the use of three instruction pipelines in parallel for a triple-issue processor. Superscalar processors were originally developed as an alternative to vector processors.



(19) Figure 2.8 A Superscalar Processor of degree m = 3.

The base scalar processor, implemented either in RISC or CISC, has degree m = 1.

In order to fully utilize a superscalar processor of degree m, m instructions must be executable in parallel. A typical superscalar architecture for a RISC processor is shown in Fig. 2.9.

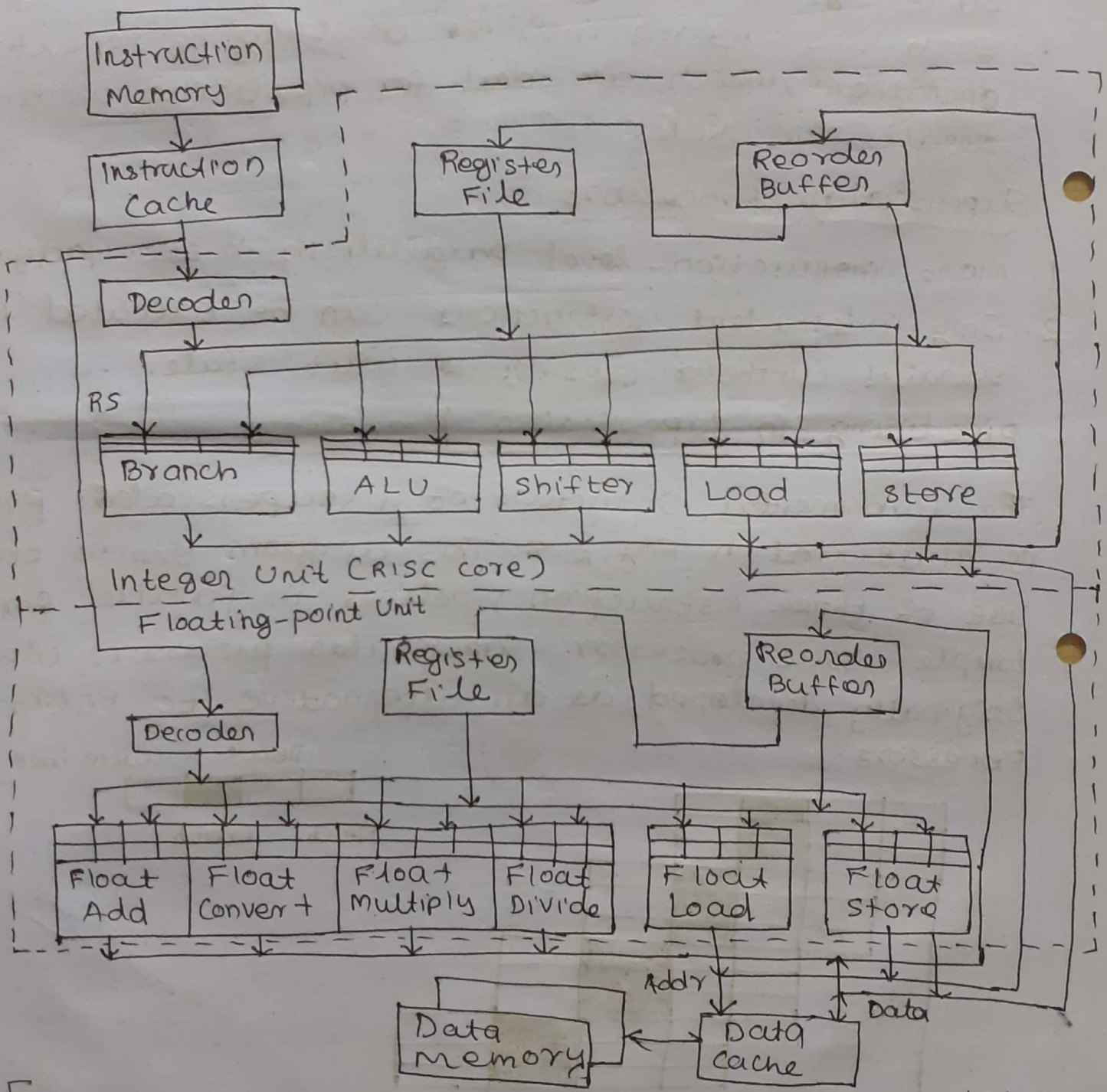


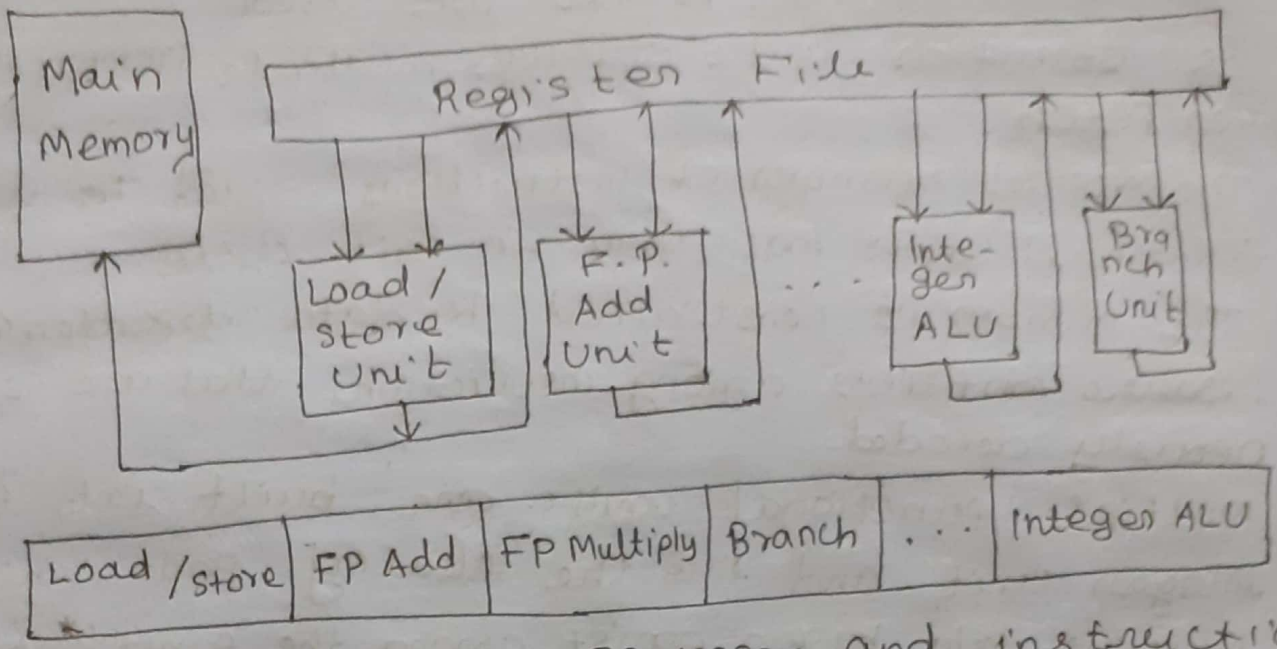
Figure 2.9 A typical superscalar RISC processor architecture consisting of an integer unit and a floating-point unit.

1. Multiple instruction pipelines are used.
2. The instruction cache supplies multiple instructions per fetch.
3. The actual number of instructions issued to various functional units may vary in each cycle.
4. The number is constrained by data dependences and resource conflicts among instructions that are simultaneously decoded.
5. Multiple functional units are built into the integer unit and into the floating-point unit.
6. Multiple data buses exist among the functional units

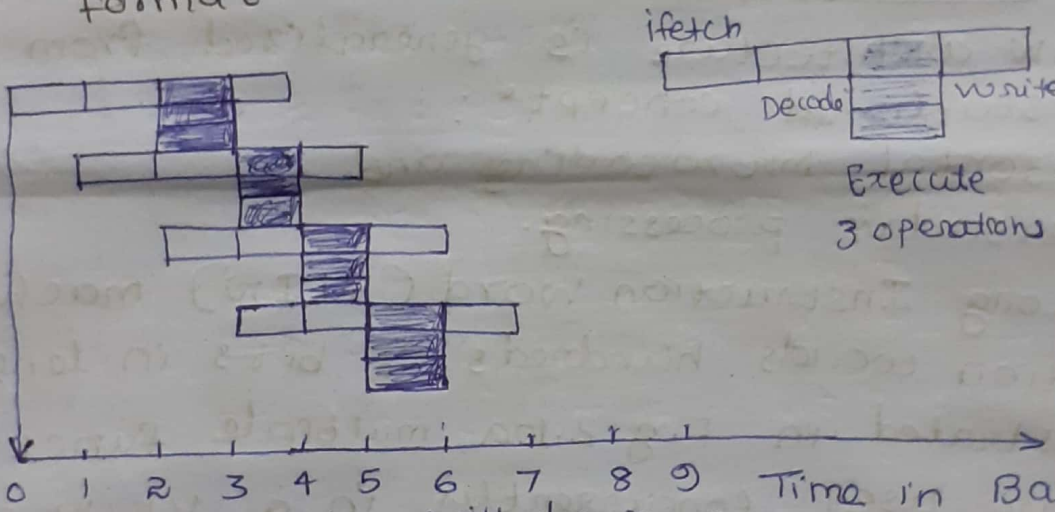
The VLIW Architecture

The VLIW architecture is generalized from two well-established concepts:
(a) horizontal microcoding and
(b) superscalar processing.

1. Very Long Instruction Word (VLIW) machine has instruction words hundreds of bits in length.
2. As illustrated in Fig. 2.10a, multiple functional units are used concurrently in a VLIW processor.
3. The operations to be simultaneously executed by the functional units are synchronized in a VLIW instruction.
4. The VLIW concept is borrowed from horizontal microcoding.
5. Different fields of the long instruction word carry the opcodes to be dispatched to different functional units.



(a) A typical VLIW processor and instruction format



(b) VLIW execution with degree m = 3

Figure 2.10 The architecture of a very long instruction word (VLIW) processor and its pipeline operations.

Pipelining in VLIW Processors

The execution of instructions by an ideal VLIW processor is shown in Fig. 2.10

Difference between VLIW machines and superscalar machines are:

1. The decoding of VLIW instructions is easier than that of superscalar instructions.
2. The code density of the superscalar machine is better when the available instruction-level parallelism is less than that exploitable by the VLIW machine.
3. A superscalar machine can be object-code-compatible with a large family of nonparallel machines.

i. Instruction parallelism and data movement in a VLIW architecture are completely specified at compile time.
ii. The CPI of a VLIW processor can be even lower than that of a superscalar processor.

VLIW Opportunities

1. The architecture is totally incompatible with that of any conventional general-purpose processor.
2. Different implementations of the same VLIW architecture may not be binary-compatible with each other.
3. A VLIW processor can eliminate the hardware or software needed to detect parallelism.
4. The advantage of VLIW architecture is its simplicity in hardware structure and instruction set.
5. The VLIW architecture has not entered the mainstream of computers.
6. In general-purpose applications, the architecture may not be able to perform well.

Vector and Symbolic Processors

1. A vector processor is a coprocessor specially designed to perform vector computations.
2. A vector instruction involves a large array of operands.

Vector Instructions

V_i : a vector register of length n
S_i : a scalar register
$M(1:n)$: a memory array of length n.
'o' : a vector operator.

Typical register-based vector operations are

$V_1 \circ V_2 \rightarrow V_3$ (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)

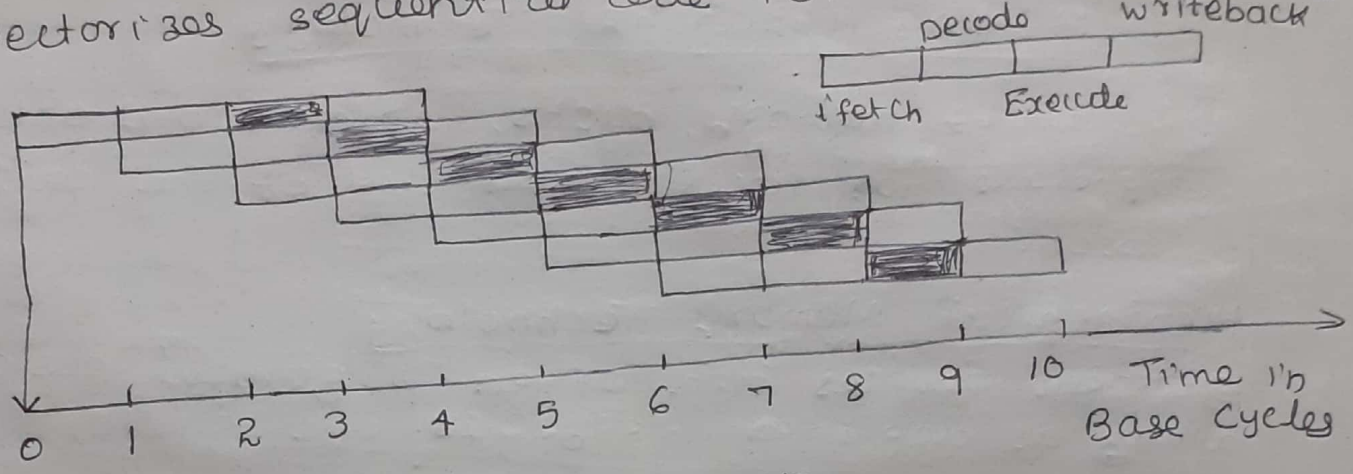
Memory-based vector operations are found in memory-to-memory vector processors such as

$M_1(1:n) \circ M_2(1:n) \rightarrow M(1:n)$
$S_1 \circ M_1(1:n) \rightarrow M_2(1:n)$
$\circ M_1(1:n) \rightarrow M_2(1:n)$
$\circ M_2(1:n) \rightarrow M(k)$

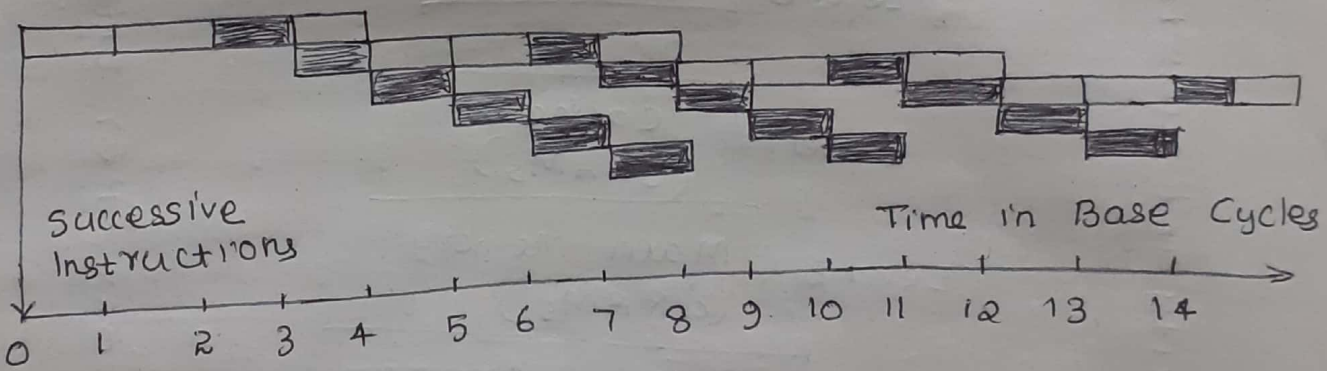
where $M_1(1:n)$ and $M_2(1:n)$ are two vectors of length n and $M(k)$ denotes a scalar quantity stored in memory location k.

Vector Pipelines

1. Vector processors take advantage of unrolled-loop-level parallelism.
2. Dedicated vector pipelines will eliminate some software overhead in looping control.
3. The effectiveness of a vector processor relies on the capability of an optimizing compiler that vectorizes sequential code for vector pipelining.



(a) Scalar pipeline execution



(b) Vector pipeline execution.

Figure 2.11 Pipelined execution in a base scalar processor and in a vector processor.

Symbolic Processors

Symbolic processing has been applied in
i) theorem proving,
ii) pattern recognition,
iii) expert systems
iv) knowledge engineering,
v) text Retrieval,
vi) cognitive science, and
vii) machine intelligence.

symbolic processors have also been called prolog processors, Lisp processors, or symbolic manipulators. Primitive operations for artificial intelligence include search, compare, logic inference, pattern matching, unification, filtering, context, retrieval, set operations, transitive closure, and reasoning operations.

Eg: The symbolics 3600 Lisp processor.

Memory Hierarchy Technology

The cost of memory, disks, printers, and other peripherals has far exceeded that of the central processor.

Hierarchical Memory Technology

storage devices are often organised as a hierarchy as depicted in Fig. 2.12.

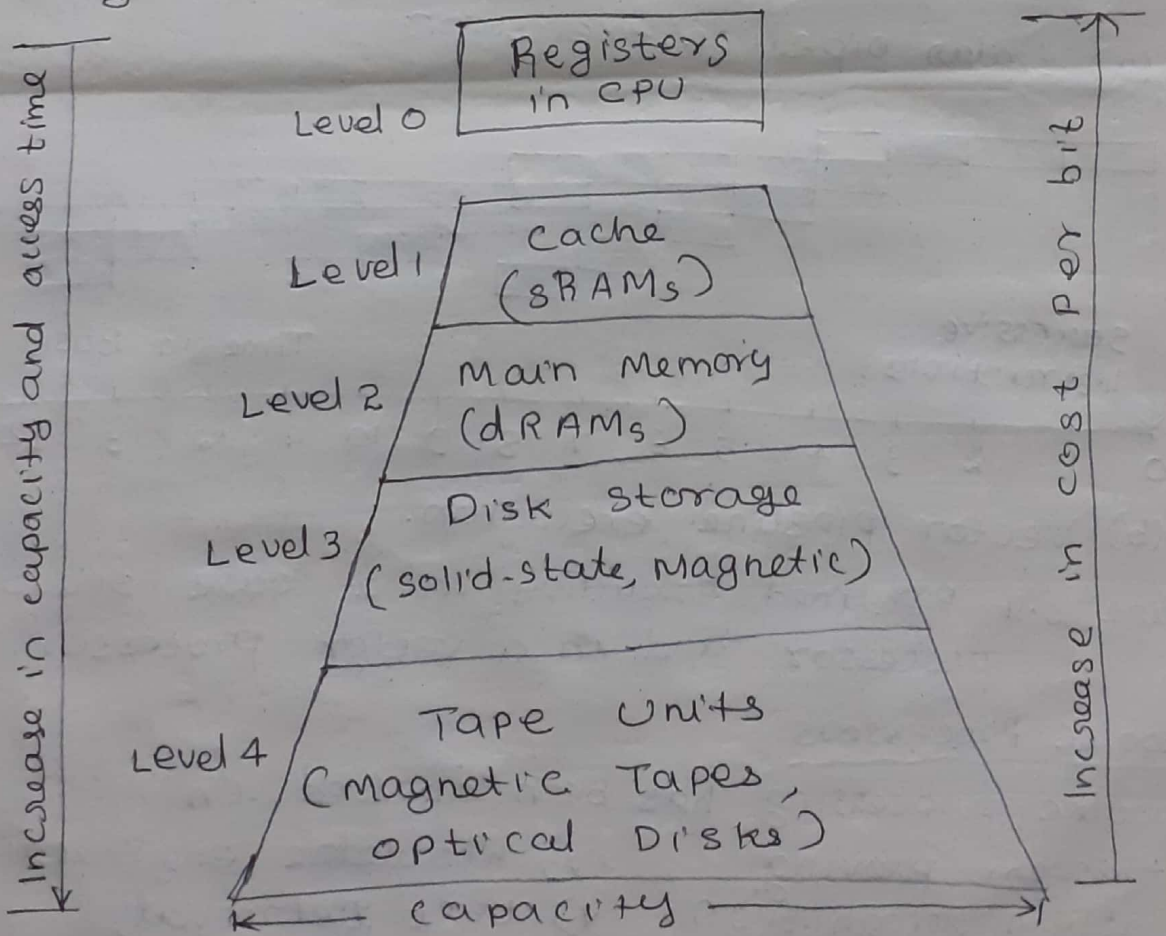


Figure 2.12
A four-level memory hierarchy with increasing capacity and decreasing speed and cost from low to high levels.

The memory technology and storage organisation at each level are characterized by five parameters:

1. the access time (t_i),
2. memory size (s_i),
3. cost per byte (c_i)
4. transfer bandwidth (b_i)
5. unit of transfer (x_i)

1. The access time t_i refers to the round-trip time from the CPU to the ith-level memory.
2. The memory size s_i is the number of bytes or words in level i.
3. The cost of ith-level memory is estimated by the product $c_i s_i$.
4. The bandwidth b_i refers to the rate at which information is transferred between adjacent levels.
5. The unit of transfer x_i refers to the grain size for data transfer between levels i and $i+1$.

Lower level

Memory devices at a lower level are
i) faster to access,
ii) smaller in size, and
iii) more expensive per byte,
iv) having a higher bandwidth.
v) using a smaller unit of transfer.

$t_{i-1} < t_i$, $s_{i-1} < s_i$, $c_{i-1} > c_i$, $b_{i-1} > b_i$ and $x_{i-1} < x_i$ for $i = 1, 2, 3$, and 4, $i = 0$ register level.

level 0 - register
level 1 - cache
level 2 - main memory
level 3 - the disk
level 4 - the tape unit.

Registers and Caches

1. parts of the processor
2. Built either on the processor chip or on the processor board.
3. Register transfer is conducted at processor speed (one clock cycle).
4. Many designers would not consider registers a level of memory.

Main memory (primary memory)

1. Much larger than cache.
2. Implemented by the most cost-effective RAM chips.
3. The main memory is managed by a MMU in cooperation with the operating system.

Disk Drives and Tape Units.

1. Handled by the OS with limited user intervention.
2. Disk-storage is the highest level of on-line memory.
3. Disk holds the system programs such as the OS and compilers and some user programs and their data sets.
4. The magnetic tape units are off-line memory for use as backup storage.

Peripheral Technology

The technology of peripheral devices has improved rapidly in recent years.
Peripheral devices include disk drives, tape units plotters, terminals, monitors, graphics displays, optical scanners, image digitizers, output microfilm devices etc.

Inclusion, Coherence and Locality.

properties of information stored in memory hierarchy ($M_1, M_2, \ldots, M_n$)

1. Inclusion,
2. coherence and
3. Locality as illustrated in Fig. 2.13.

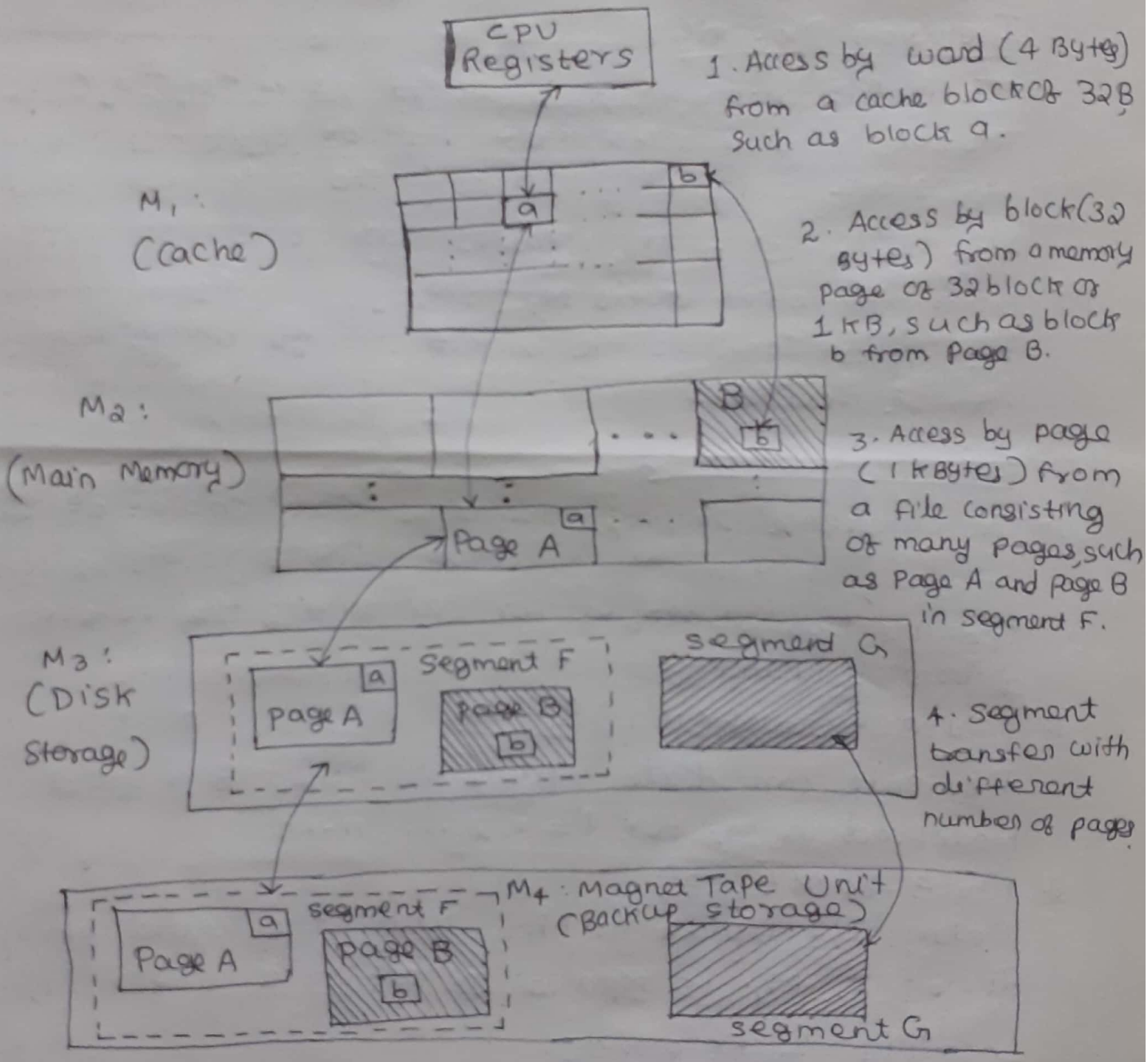


Figure 2.13 The inclusion property and data transfers between adjacent levels of a memory hierarchy.

1. Cache memory the innermost level M_1, which directly communicates with the CPU registers.
2. The outermost level M_n contains all the information words stored.
3. The collection of all addressable words in M_n forms the virtual address space of a computer.

Inclusion Property

The inclusion property is stated as $M_1 \subset M_2 \subset M_3 \subset \cdots \subset M_n$.
The set inclusion relationship implies that all information items are originally stored in the outermost level M_n.
A word miss in M_i implies that it is also missing from all lowel levels $M_{i-1}, M_{i-2}, \ldots, M_1$.
The highest level is the backup storage, where everything can be found.

Coherence Property

1. The coherence property requires that copies of the same information item at successive memory levels be consistent.
2. There are two strategies for maintaining the coherence in a memory hierarchy.

1. Write-through (WT)
   Demands immediate update in M_{i+1} if a word is modified in M_i, for $i = 1, 2, \ldots, n-1$.
2. Write-back (WB)
   Delays the update in M_{i+1} until the word being modified in M_i is replaced or removed from M_i.

Locality of References.

Memory references are generated by the CPU for either instruction or data access.

There are three dimensions of the locality property:
1. temporal,
2. spatial, and
3. sequential.

(1) Temporal locality
Recently referenced items (instructions or data) are likely to be referenced again in the near future.
caused by:
a) iterative loops,
b) process stacks,
c) temporary variables, or
d) subroutines.

(2) spatial locality
This refers to the tendency for a process to access items whose addresses are near one another.
Eg: operations on tables or arrays involve accesses of a certain clustered area in the address space.

(3) sequential locality
The execution of instructions follows a sequential order (or the program order) unless branch instructions create out-of-order executions.
The access of a large data array also follows a sequential order.

Memory Design Implications

1. The sequentiality in program behavior also contributes to the spacial locality because sequentially coded instructions and array elements are often stored in adjacent locations.
2. The temporal locality leads to the popularity of the least recently used (LRU) replacement algorithm.
3. The spatial locality assists us in determining the

size of unit data transfers between adjacent memory levels.

4. The sequential locality affects the determination of grain size for optimal scheduling.

Memory Capacity Planning

1. The performance of a memory hierarchy is determined by the effective access time.
2. Effective access time depends on
   a) hit ratios and
   b) access frequencies
   at successive levels.

Hit Ratios

1. When an information item is found in M_i, we call it a hit, otherwise a miss.
2. The hit ratio h_i at M_i is the probability that an information item will be found in M_i.
3. The miss ratio at M_i is defined as $1-h_i$.
4. The hit ratios at successive levels are a function of
   1) memory capacities
   2) management policies, and
   3) program behavior.
5. Assume $h_0 = 0$ and $h_n = 1$ which means the CPU always accesses M_1 first and the access to the outermost memory M_n is always a hit.

Access frequency

The access frequency to M_i is defined as

$f_i = (1-h_1)(1-h_2)\cdots(1-h_{i-1})h_i$.

when there are $i-1$ misses at the lower levels and

and a hit at M_i.

$$\sum_{i=1}^{n} f_i = 1 \text{ and } f_1 = h_1$$

The inner levels of memory are accessed more often than the outer levels.

$$f_1 >> f_2 >> f_3 >> \cdots >> f_n.$$

Effective access Time

The misses have been called blocks misses in the cache and page faults in the main memory because blocks and pages are the units of transfer between these levels.

Effective access time using the access frequencies f_i for $i = 1, 2, \ldots, n$

$$T_{eff} = \sum_{i=1}^{n} f_i \cdot t_i$$

$$= h_1 t_1 + (1-h_1) h_2 t_2 + (1-h_1)(1-h_2) h_3 t_3 + \cdots + (1-h_1)(1-h_2)\cdots(1-h_{n-1}) t_n$$

Hierarchy Optimization

The total cost of a memory hierarchy is estimated as follows:

$$C_{total} = \sum_{i=1}^{n} c_i \cdot s_i$$

This implies that the cost is distributed over n levels. since $c_1 > c_2 > c_3 > \cdots c_n$, we have to choose $s_1 < s_2 < s_3 < \cdots s_n$.

The optimal design of a memory hierarchy should result in a T_{eff} close to the t_1 of M_1 and a total

cost close to the C_n of M_n. In reality, this is difficult to achieve due to the tradeoffs among n levels.

Example: The design of a memory hierarchy

Consider the design of a three-level memory hierarchy with the following specifications for memory characteristics:

Memory level	Access time	Capacity	Cost / Kbyte
Cache	$t_1 = 25$ ns	$S_1 = 512$ Kbytes	$C_1 = \$1.25$
Main memory	t_2 = unknown	$S_2 = 32$ Mbytes	$C_2 = \$0.2$
Disk array	$t_3 = 4$ ms	S_3 = unknown	$C_3 = \$0.0002$

The design goal is to achieve an effective memory access time $t = 10.04$ μs with a cache hit ratio $h_1 = 0.98$ and a hit ratio $h_2 = 0.9$ in the main memory. Also, the total cost of the memory hierarchy is upper-bounded by \$15,000. The memory hierarchy cost is calculated as

$$C = c_1 S_1 + c_2 S_2 + c_3 S_3 \leq 15,000$$

The maximum capacity of the disk is thus obtained as $S_3 = 39.8$ Gbytes without exceeding the budget.

Next, we want to choose the access time (t_2) of the RAM to build the main memory. The effective memory-access time is calculated as

$$t = h_1 t_1 + (1-h_1) h_2 t_2 + (1-h_1)(1-h_2) h_3 t_3 \leq 10.04$$

Substituting all known parameters, we have

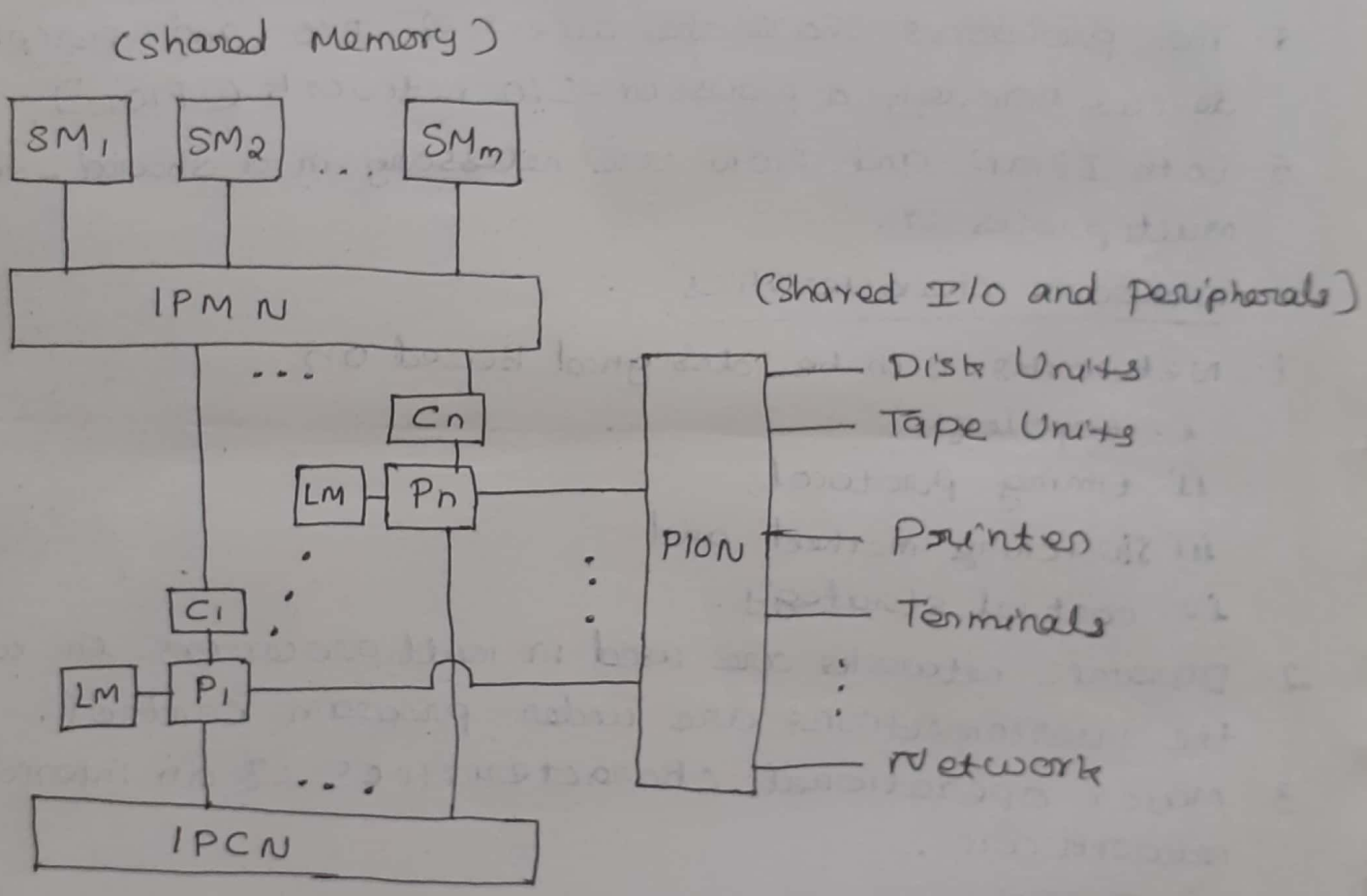
$10.04 \times 10^{-6} = 0.98 \times 25 \times 10^{-9} + 0.02 \times 0.9 \times t_2 + 0.02 \times 0.1 \times 1 \times 4 \times 10^{-3}$. Thus $t_2 = 903$ ns.

Suppose one wants to double the main memory to 64 Mbytes at the expense of reducing the disk capacity under the same budget limit. This change will not affect the cache hit ratio. But it may increase the hit ratio in the main memory if a proper page replacement algorithm is used. Also, the effective memory-access time will be enhanced.

Module III

Multiprocessor System Interconnects

Parallel processing demands the use of efficient system interconnects for fast communication among multiple processors and shared memory, I/O, and peripheral devices. Hierarchical buses, Crossbar switches, and multistage networks are often used for this purpose.



Legends: IPMN (Inter-Processor-Memory Network)
PION (Processor-I/O Network)
IPCN (Inter-Processor Communication Network)
P (processor)
C (cache)
SM (shared memory)
LM (Local Memory)

Figure 3.1 Interconnection structures in a generalised multiprocessor system with local memory, Private caches,

shared memory, and shared peripherals.

1. A generalized multiprocessor system is depicted in Fig 3.1. This architecture combines features from the UMA, NUMA, and COMA models.
2. Each processor P_i is attached to its own local memory and private cache.
3. Multiple processors are connected to shared-memory modules through an interprocessor-memory network (IPMN).
4. The processors share the access of I/O and peripheral devices through a processor-I/O network (PION).
5. Both IPMN and PION are necessary in a shared-resource multiprocessor.

Network characteristics

1. Networks can be designed based on
   i. topology,
   ii. timing protocol,
   iii. switching method, and
   iv. control strategy.
2. Dynamic networks are used in multiprocessors in which the interconnections are under program control.
3. Major operational characteristics of an interconnection network are
   i. Timing,
   ii. switching, and
   iii. control.
4. Timing can be either
   i. Synchronous or - controlled by global clock
   ii. Asynchronous - handshaking or interlocking mechanism.
5. Switching can be either
   i. circuit switching or ii. packet switching.

In circuit switching, once a device is granted a path in the network, it occupies the path for the entire duration of the data transfer.

In packet switching, the information is broken into small packets individually competing for a path in the network.

6. Control is classified as
   i. centralized or
   ii. distributed.

With centralised control, a global controller receives requests from all devices attached to the network and grants the network access to one or more requests.

In a distributed system, requests are handled by local devices independently.

Hierarchical Bus Systems.

The hierarchy of bus systems are packaged at different levels are depicted in Fig. 3.2, including local buses on boards, backplane buses, and I/O buses.

Local Bus

1. Buses implemented on printed-circuit boards are called local buses.
2. A memory board uses a memory bus to connect the memory with the interface logic.
3. An I/O board or network interface board uses a data bus.

Each of these board buses consists of signal and utility lines. With the sharing of the lines by many I/O devices, the layout of these lines may be at different layers of the PC board.

Backplane Bus

1. A Backplane is a printed circuit on which many connectors are used to plug in functional boards.
2. A system bus consisting of shared signal paths and utility lines, is built on the backplane.
3. This system bus provides a common communication path among all plug-in boards.

Eg. backplane bus standards are

i. VME bus (IEEE standard 1014-1987)
ii. Multibus II (IEEE standard 1296-1987), and
iii. Futurebus + (IEEE standard 896.1-1991).

I/O Bus

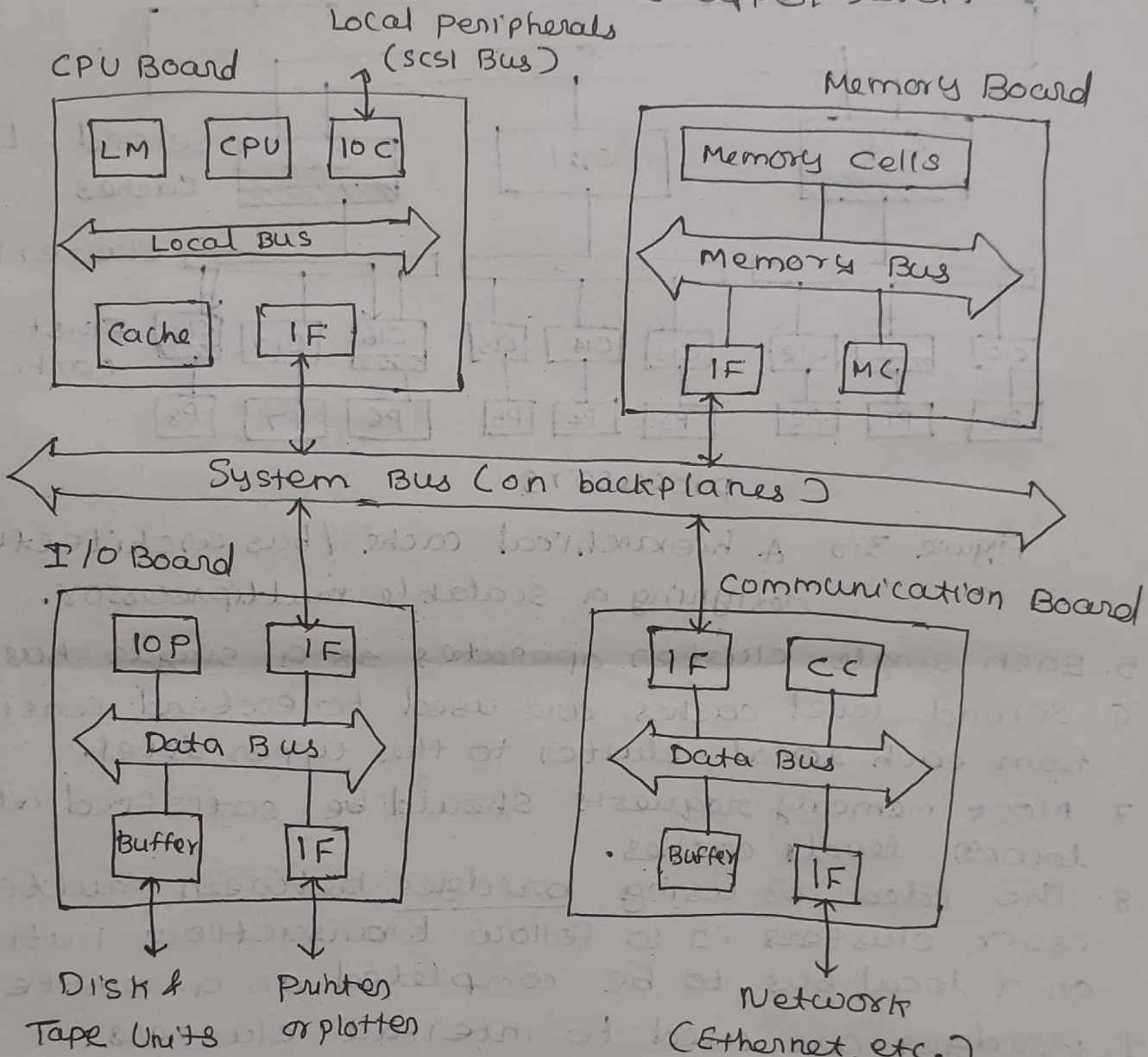
1. Input/Output devices are connected to a computer system through an I/O bus such as the SCSI bus.
2. This bus is made of coaxial cables with taps connecting disks, printer, and tape unit to a processor through an I/O controller.

Hierarchical Buses and Caches:

Wilson has proposed a hierarchical cache/bus architecture as shown in Figure 3.3.

1. It is a multilevel tree structure in which the leaf nodes are processors and their private caches (denoted P_j and C_{1j}).
2. Leaf nodes are divided into several clusters, each of which is connected through a cluster bus.
3. An intercluster bus is used to provide communication among the clusters.
4. Second level caches (denoted as C_{2i}) are used between each cluster bus and the intercluster bus.

5. Each single cluster operates as a single-bus system.
6. Second-level caches are used to extend consistency from each local cluster to the upper level.



Legends: IF (Interface logic), LM (Local Memory)
IOC (I/O controller), MC (Memory Controller)
IOP (I/O processor), CC (Communication Controller)

Figure 3.2 Bus systems at board level, backplane level, and I/O level.

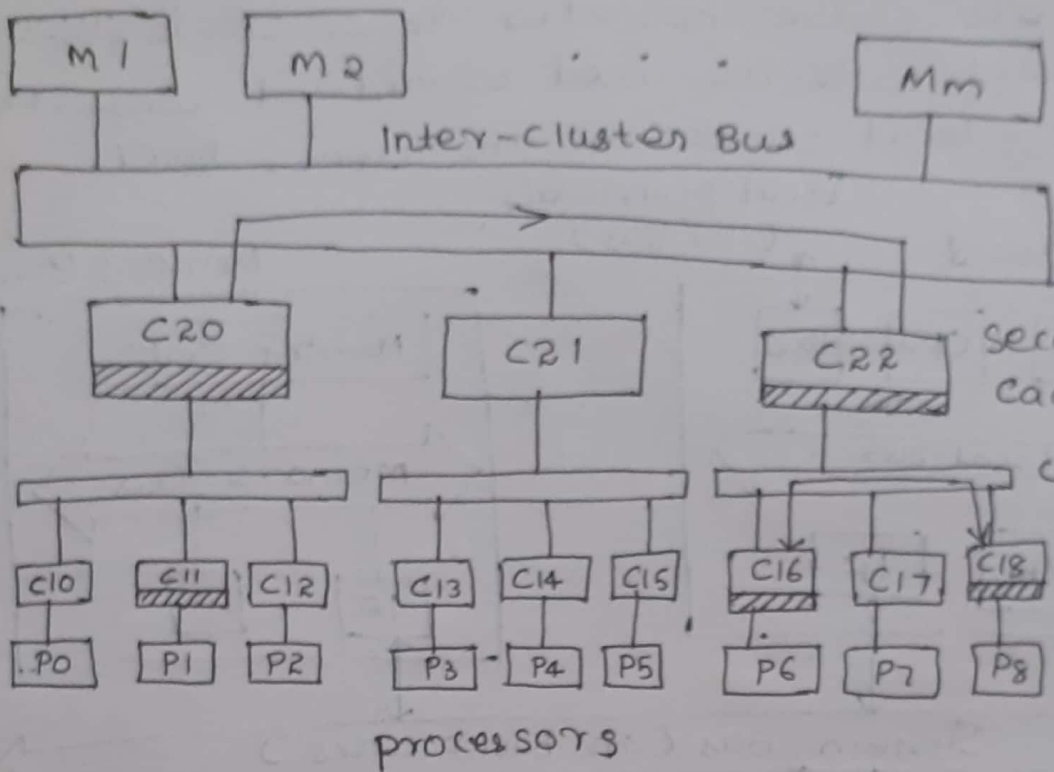


Figure 3.3 A hierarchical cache/bus architecture for designing a scalable multiprocessor.

5. Each single-cluster operates as a single-bus system
6. Second-level caches are used to extend consistency from each local cluster to the upper level.
7. Most memory requests should be satisfied at the lower-level caches.
8. The idea of using bridges between multiprocessor clusters is to follow transactions initiated on a local bus to be completed on a remote bus.
9. Bridges are used to interface clusters.
10. The main functions of a bridge include
    i. communication protocol conversion,
    ii. interrupt handling in split transactions, and
    iii. serving as cache and memory agents.

Example: Encore's ultramax multiprocessor architecture.
It has a two-level hierarchical-bus architecture
It is very similar to that characterised by wilson, except that the global nanobus is used only for interclusters communications.

Crossbar Switch and Multiport Memory

1. Switched networks provide dynamic interconnections between the inputs and outputs.
2. For interconnecting computer subsystems or for constructing multiprocessors or multicomputers static and dynamic networks are used.
3. Static and dynamic networks can be used for internal connections among processors, memory modules, and I/O disk arrays in a centralized system, or for distributed networking of multicomputer nodes.
4. Static networks are formed of point-to-point direct connections which will not change during program execution.
5. Dynamic networks are implemented with switched channels which are dynamically configured to match the communication demand in user programs.

   Dynamic networks include buses, crossbar switches and multistage networks which are used in shared-memory multiprocessors.

Single-stage networks (recirculating network)

Data items have to recirculate through the single stage many times before reaching their destination.

Eg:
1. Crossbar switch and
2. multiport memory

Multistage network

Consists of more than one stage of switch boxes.
Able to connect from any input to any output.

Eg: 1. The Omega network,
2. Flip network, and
3. Baseline networks.

Blocking Networks

A multistage network is called blocking if the simultaneous connections of some multiple input-output pairs result in conflicts in the use of switches or communication links.

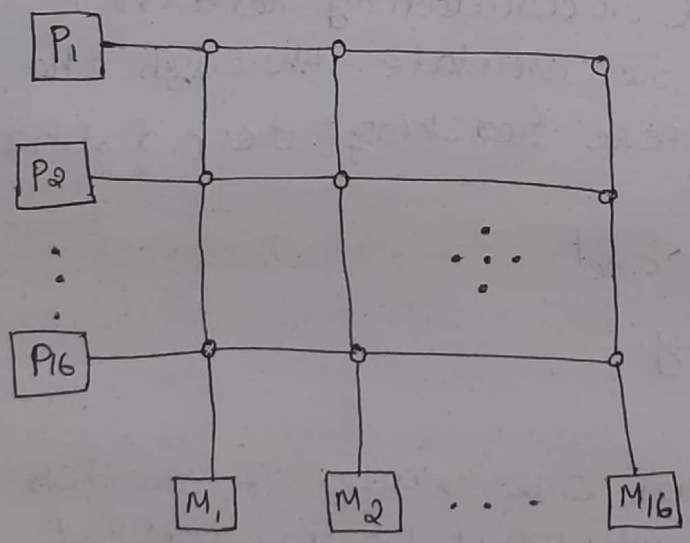
Eg: 1. Omega (Lawrie, 1975)
2. Baseline (Wu and Feng, 1980),
3. Banyan (Goke and Lipovski, 1973), and
4. Delta networks (Patel, 1979).

NonBlocking Network

A multistage network is called nonblocking if it can perform all possible connections between inputs and outputs by rearranging its connections. A connection path can always be established between any input-output pair.

Eg. 1. Benes networks (Benes, 1965)
2. Clos networks (Clos, 1953)

Crossbar Networks



Interprocessor-memory crossbar network.

1. In a crossbar network, every input port is connected to a free output port through a crosspoint switch without blocking.

2. A crossbar network is a single-stage network built with unary switches at the crosspoints, which can be set open or closed, providing a point-to-point connection path between the source (processor) and destination (memory).
3. Once the data is read from the memory, its value is returned to the requesting processor along the same cross-point switch.
4. A crossbar switch network is a single-stage, nonblocking, permutation network.

Crosspoint Switch Design

1. Out of n crosspoint switches in each column of an n x m crossbar mesh.
2. Each crosspoint switch requires the use of a large number of connecting lines accommodating address, data path, and control signals.
3. So far only small crossbar networks with $n \leq 16$ have been built into commercial machines.
4. Figure 3.6 shows the schematic design of a crosspoint switch in a single crossbar network.
5. Multiplexer modules are used to select one of n read or write requests for service.
6. Each processor sends in an independent request, and the arbitration logic makes the selection based on certain fairness or priority rules.
7. Based on the control signal received, only one out of n sets of information lines is selected as the output of the multiplexer tree.
8. The memory address is entered for both read and write access.

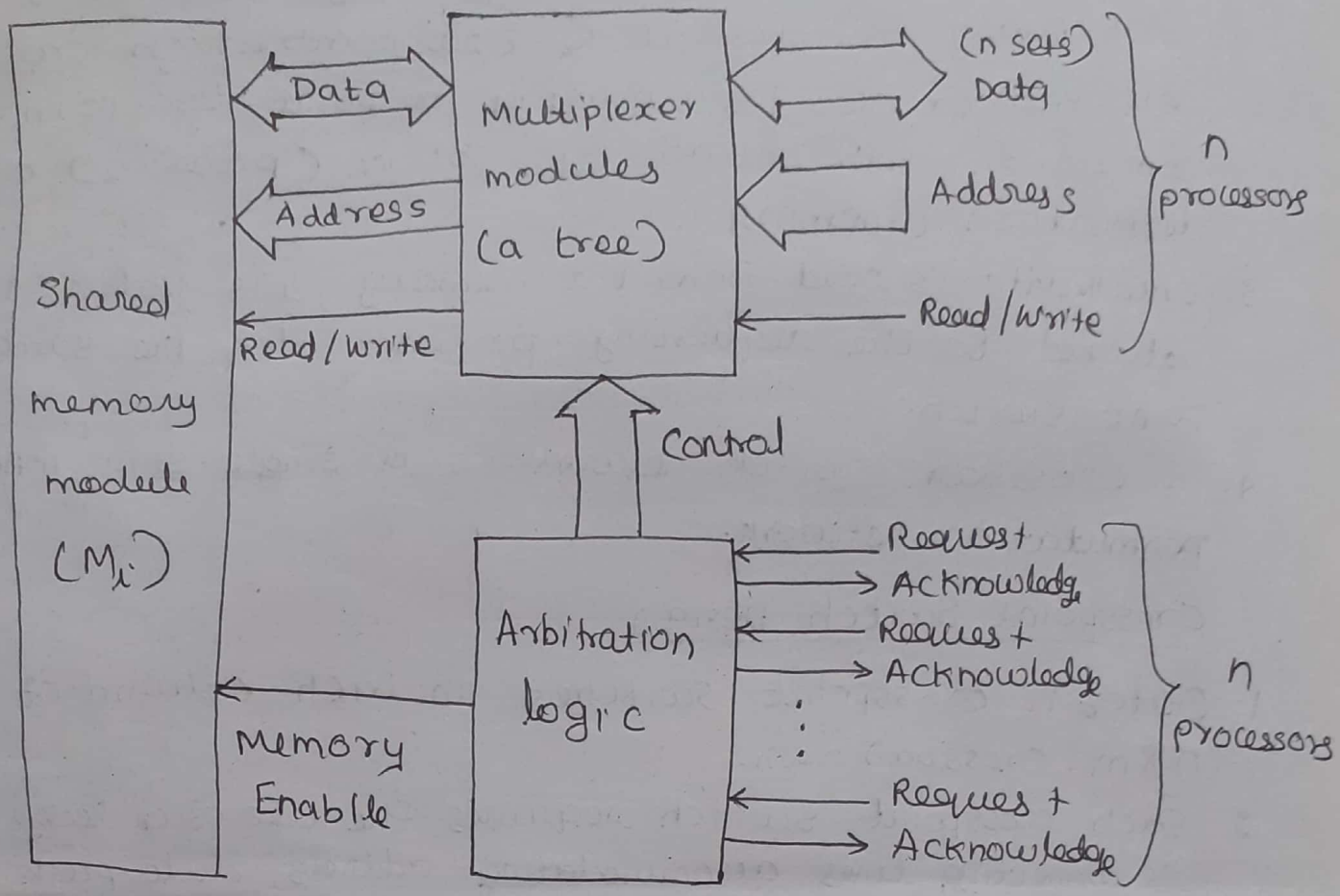


Figure 3.4 Schematic design of a crosspoint switch in a crossbar network.

9. Read : The data fetched from memory are returned to the selected processor in the reverse direction using the data path established at the corresponding crosspoint.
10. Write : The data on the data path are stored in memory.

Crossbar Limitations

1. A single processor can send many requests to multiple memory modules.
2. The crossbar network offers the highest bandwidth of n data transfers per memory cycle, as compared with only one data transfer per bus cycle.

3. A crossbar network is cost-effective only for small multiprocessors with a few processors accessing a few memory modules.
4. A single-stage crossbar network is not expandable once it is built.
5. Redundancy or parity-check lines can be built into each crosspoint switch to enhance the fault tolerance and reliability of the crossbar network.

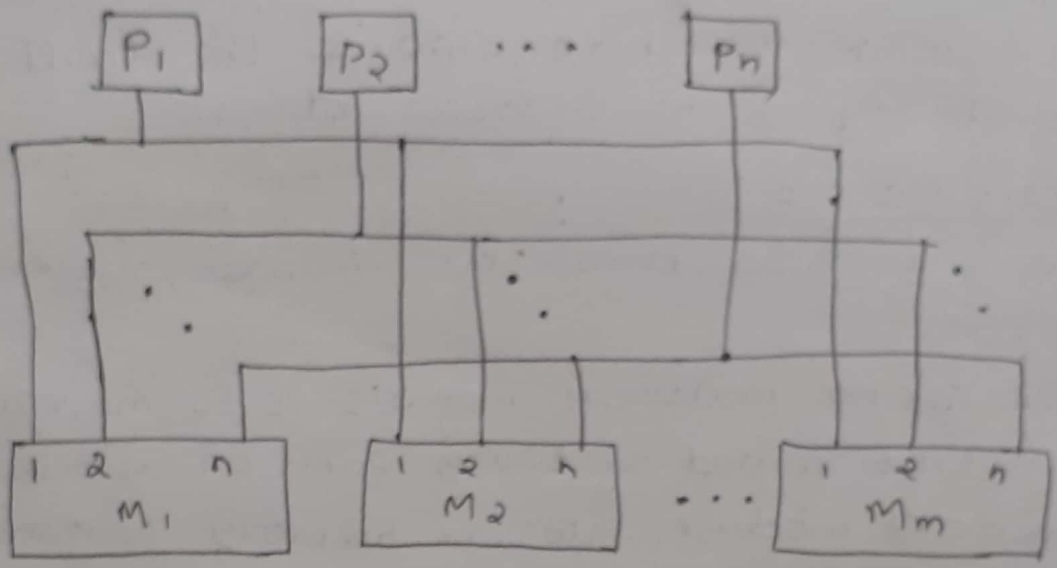
Multiport Memory

1. Building a crossbar network into a large system is cost-prohibitive.
2. The idea for the multiport memory is to move all crosspoint arbitration and switching function associated with each memory module into the memory controller.
3. Many mainframe multiprocessors use a multiport memory organization.
4. The memory module becomes more expensive due to the added access ports and associated logic as demonstrated in Fig. 3.7a.
5. Only one of n processor requests will be honored at a time.
6. The multiport memory organization is a compromise solution between a low-cost, low-performance bus system and a high-cost, high-bandwidth crossbar system.

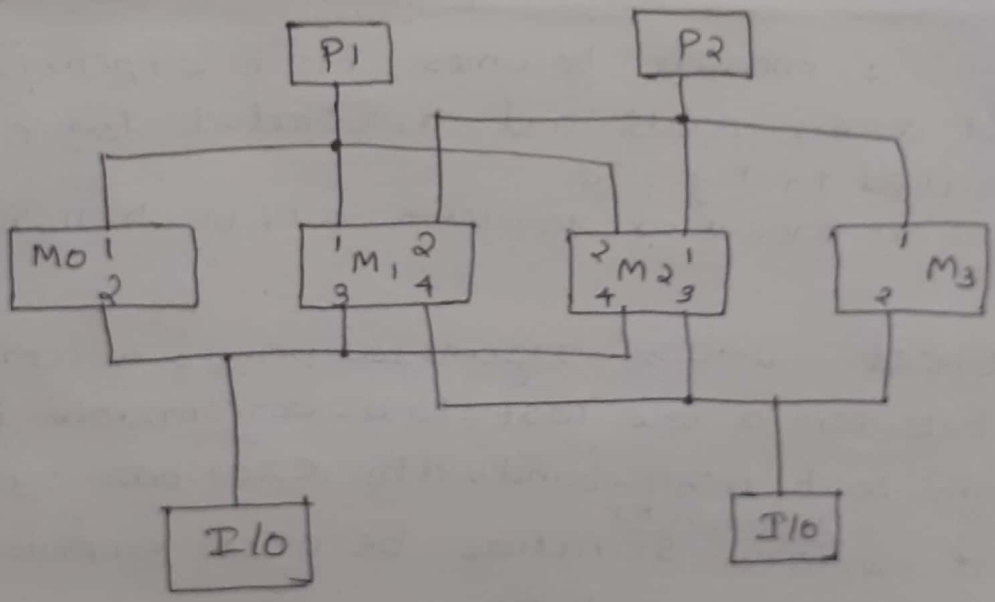
Drawbacks

7. Multiport memory structure becomes expensive when m and n become very large.
8. A multiport memory is not scalable because once the ports are fixed, no more processors can be added without redesigning the memory controller.
9. Another drawback is the need for a large number of interconnection cables and connectors when the configuration becomes large.

10. some of processors are CPUs, some are I/O processors, and some are connected to dedicated processors.
11. The access to the ports is prioritized under operating system control.



(a) m shared memory modules



(b) Memory ports prioritized or privileged in each module by numbers.

Figure 3.5 Multiport memory organizations for multiprocessor systems.

Multistage and Combining Networks

1. Multistage networks are used to build larger multiprocessor systems.
   Two multistage networks are
   i. Omega network and
   ii. the Butterfly network
2. There is a special class of multistage networks, called combining networks, for resolving access conflicts automatically through the network.

Omega Network

Figure 3.6a to 3.6d show four possible connections of 2×2 switches used in constructing the Omega network.

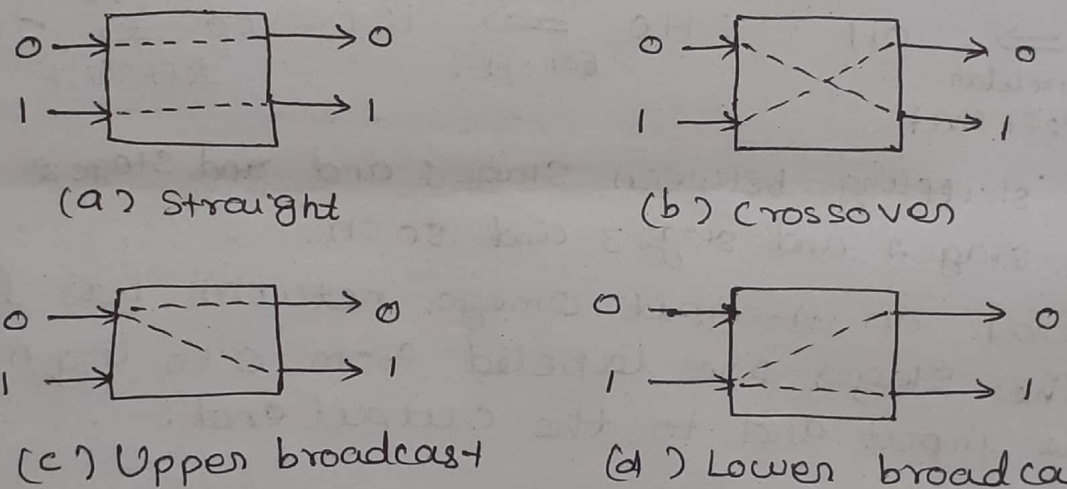


Figure 3.6 2×2 switches.

1. An n-input Omega network requires $\log_2 n$ stages of 2×2 switches.
2. Each stage requires n/2 switch modules.
3. In total, the network uses $n \log_2 n/2$ switches.
4. Each switch module is individually controlled.
5. An Omega network is a network configuration used in parallel computing architectures. It is an indirect topology that relies on the perfect shuffle interconnection algorithm.

n x n omega network.

where n is the number of inputs and outputs of omega network.

k x k switch

where k is the number of inputs and outputs of switch

No: of stages = $\log_k n$

where n is the number of inputs

k is the switch type

No: of switch in a stage = $\frac{n}{k}$

Shuffling

Circular left shift by $\log_2 k$ bits

The number of bits to shift left circularly = $\log_2 k$

eg: 101 $\Rightarrow$ 011 (circular left shift); 110 $\Rightarrow$ 101 (circular left shift); 100 $\Rightarrow$ 001 (circular left shift)

copy the shuffling between stage 1 and stage 2 to between stage 2 and stage 3 and so on.

6. In general, an n-input omega network has $\log_2 n$ stages. The stages are labeled from 0 to $\log_2 n - 1$ from the input end to the output end.

Routing in Omega network.

1. Data routing is controlled by inspecting the destination code in binary.
2. When the ith high-order bit of the destination code is a 0, a 2x2 switch at stage i connects the input to the upper output. Otherwise, the input is directed to the lower output.
3. Two switch settings are shown in Figures 3.7 a and b with respect to permutations $\pi_1 =$ $\pi_4 = (0, 7, 6, 4, 2)(1, 3)(5)$ and

$\Pi_2 = (0,6,4,7,3)(1,5)(2)$, respectively.

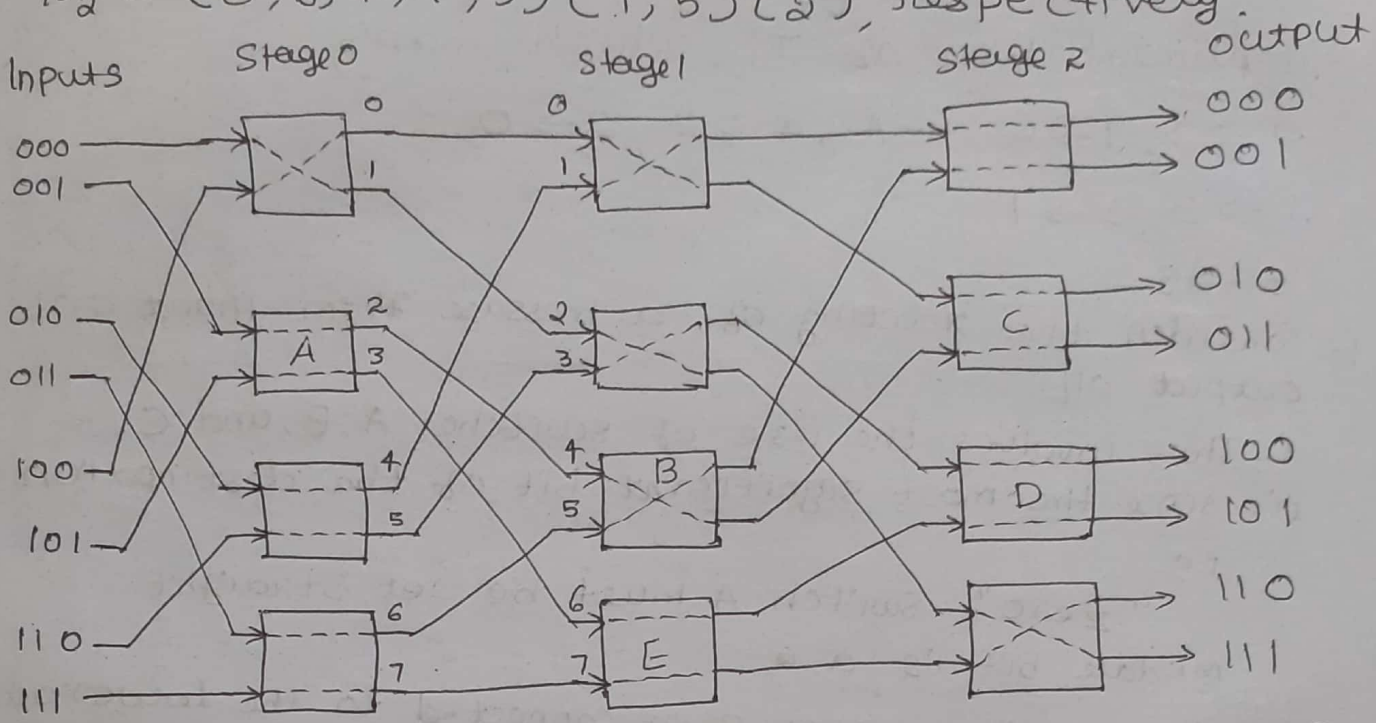


Fig. 3.7(a) Permutation $\Pi_1 = (0,7,6,4,2)(1,3)(5)$ implemented on an Omega network without blocking.

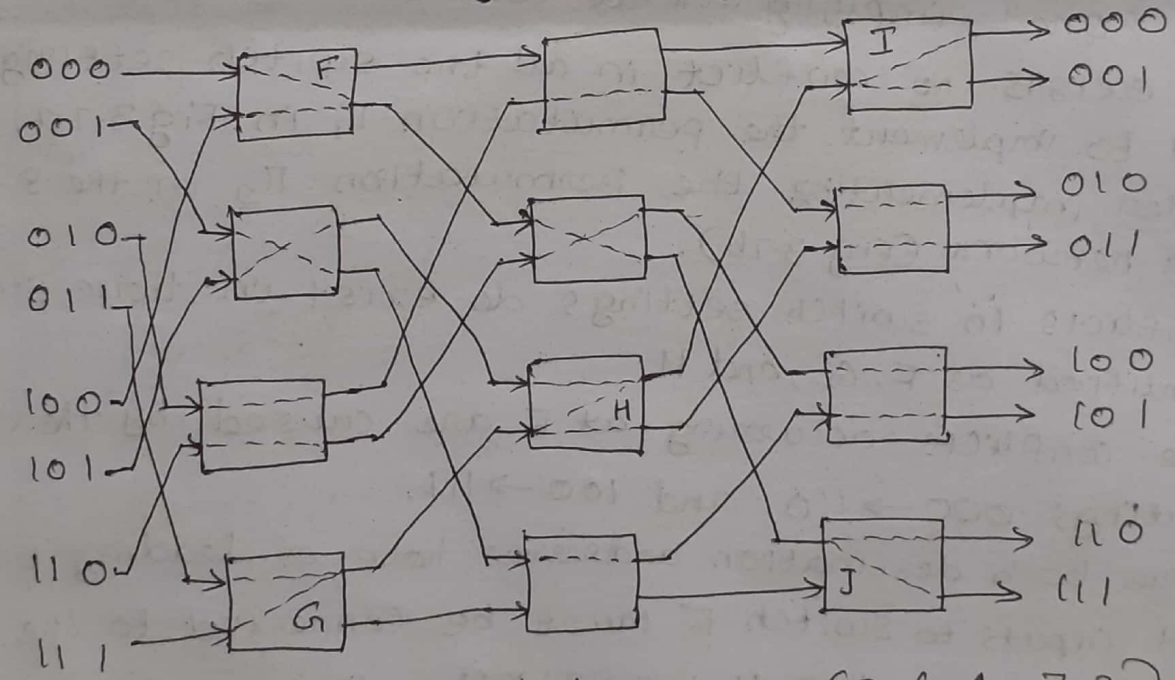


Figure 3.7(b) Permutation $\Pi_2 = (0,6,4,7,3)(1,5)(2)$ blocked at switches marked F, G, and H

Figure 3.7 Two switch settings of an 8×8 Omega network built with 2×2 switches.

The switch settings are in Fig. 3.7a are for the implementation of π_1, which maps

$0 \to 7, 7 \to 6, 6 \to 4, 4 \to 2, 2 \to 0,$

$1 \to 3, 3 \to 1$

$5 \to 5.$

1. Consider the routing of a message from input 001 to output 011.
   i. This involves the use of switches A, B, and C.
   ii. Since the most significant bit of the destination 011 is
   "zero", switch A must be set straight
   Middle bit is a
   "one", switch B is connected to the lower output with a "crossover" connection.
   The least significant bit is
   "one", implying a flat connection in switch C.
2. There exists no conflict in all the switch settings needed to implement the permutation π_1 in Fig. 3.7a.
3. Consider implementing the permutation π_2 in the 8-input Omega network (Fig 3.7b).
   i. Conflicts in switch settings do exist in three switches identified as F, G, and H.
   ii. The conflicts occurring at F are caused by the desired routings $000 \to 110$ and $100 \to 111$.
   iii. Since both destination addresses have a leading bit 1, both inputs to switch F must be connected to the lower output. To resolve the conflicts, one request must be rejected.
4. Not all permutations can be implemented in one pass through the Omega network.

5. The Omega network is a kind of block network.
6. In case of blocking, one can establish the conflicting connections in several passes.
7. Example: for Π_2 we can connect
   000 → 110, 001 → 101, 010 → 010, 101 → 001, 110 → 100 in the first pass and 011 → 000, 100 → 111, 111 → 011 in the second pass.
8. In general, if 2×2 switch boxes are used, an n-input Omega network can implement $n^{n/2}$ permutations in a single pass. There are n! permutations in total.

Problem

Design an 8 input omega network using 2×2 switches as building blocks. Show the switch settings for the permutation $\Pi_1 = (0, 6, 4, 7, 3)(1, 5)(2)$. Show the conflicts in switching setting if any. Explain blocking and non-blocking networks in this context.

Ans: Figure 3.7(b) and its explanation is the answer.

No: of inputs = 8 so 8×8 omega network.

Therefore n = 8

Switch is of 2×2, so k = 2 i.e k×k switch.

Number of switch in a stage = n/k = 8/2 = 4 switches.

No: of stages = $\log_k n = \log_2 8 = 3$

Circular left shift by $\log_2 k$ bits. = $\log_2 2 = 1$.

9. The omega network can also be used to broadcast data from one source to many destinations as exemplified in Fig. 3.8.

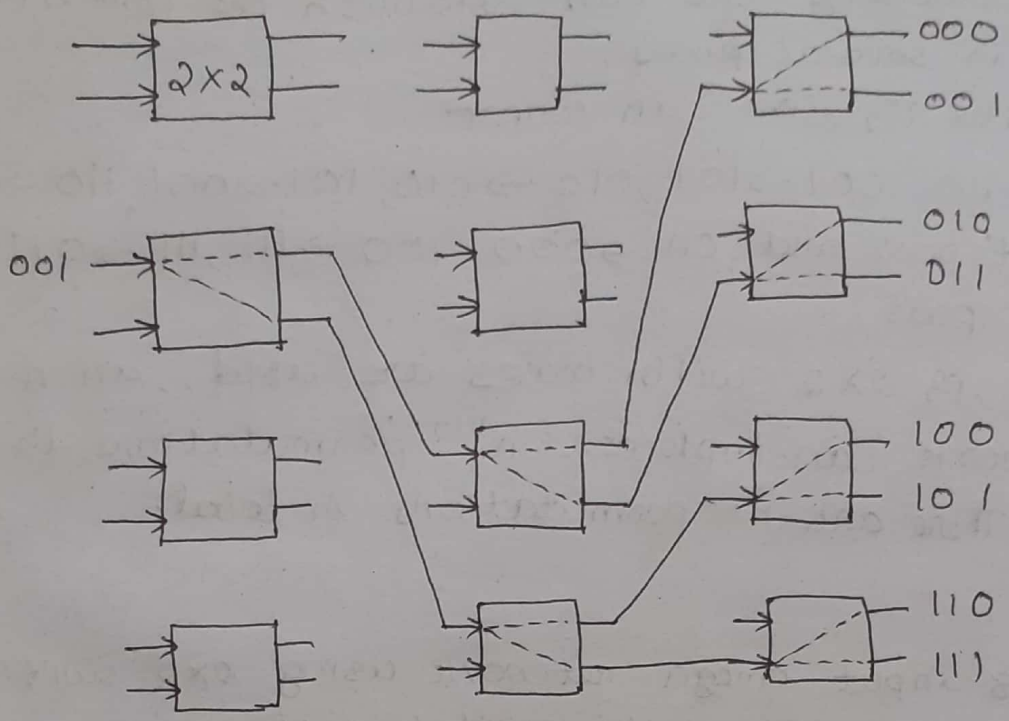


Figure 3.8. Broadcast connections

The omega network can also be used to broadcast data from one source to many destinations using the upper broadcast or lower broadcast switch settings.

In Figure 3.8, the message at input 001 is being broadcast to all eight outputs through a binary tree connection.

Routing in Butterfly Networks

1. Butterfly networks is constructed with crossbar switches as building blocks.
2. The following figure shows a 64-input butterfly network built with two stages ($2 = \log_8 64$) of 8×8 crossbar switches.
3. The eight-way shuffle functions is used to establish the interstage connections between stage 0 and stage 1.
4. No broadcast connections are allowed in a Butterfly network.
5. Butterfly network is a restricted subclass of Omega networks.

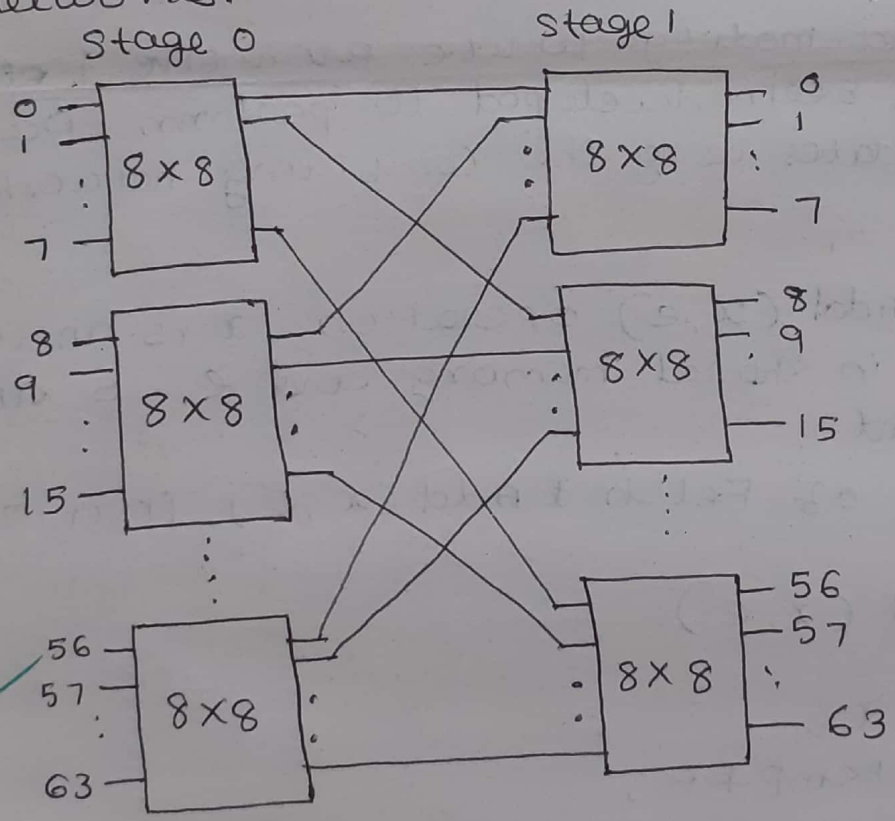


Figure 3.9 A two-stage 64×64 Butterfly switch network built with 16 8×8 Crossbar switches and eight-way shuffle interstage connections.

The Hot-spot problem

1. Hot-spot : certain memory module being excessively accessed by many processors at the same time.
2. When the network traffic is nonuniform, a hot spot may appear.

   Eg: A semaphore variable being used as a synchronization barrier may become a hot spot since it is shared by many processors.
3. Combining mechanism:

   The purpose of combining mechanism was to combine multiple requests heading for the same destination at switch points where conflicts are taking place.

   An atomic read-modify-write primitive Fetch & Add (x,e), has been developed to perform parallel memory updates using the combining network.
4. Fetch & Add.

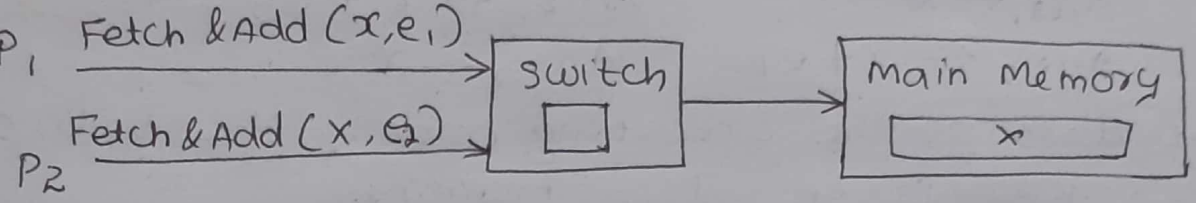
   In a Fetch & Add (x,e) operation, x is an integer variable in shared memory and e is an integer increment.

   The semantics of Fetch & Add (x,e) in a single processor is

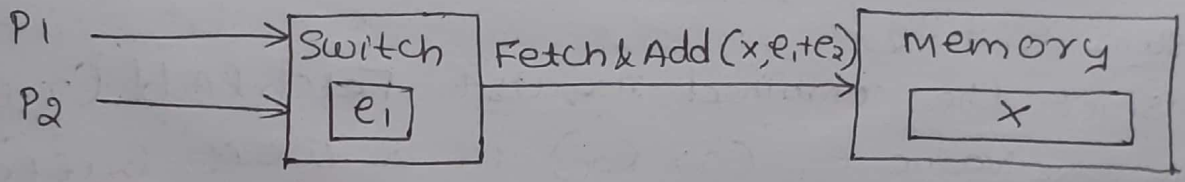
```
Fetch & Add (x, e)
{ temp ← x;
  x ← temp + e;
  return temp }
```

When N processes attempt to Fetch & Add (x, e) the same memory word simultaneously, the memory is updated only once following a serialization principle.

The following Figure illustrates two simultaneous requests are combined in a switch.

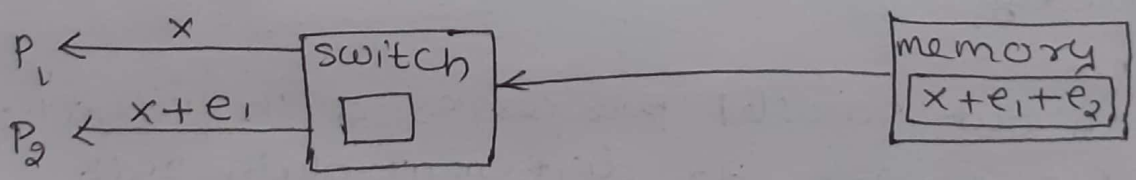


(a) Two requests meet at a switch



(b) The switch forms the sum $e_1 + e_2$, stores e_1 in buffer, and transmits the combined request to memory.

(c) The original value stored in x is returned to switch, and the new value $x + e_1 + e_2$ is stored in memory.



(d) The values x and $x + e_1$ are returned to P_1 and P_2, respectively.

Figure 3.10 Two Fetch & Add operations are combined to access a shared variable simultaneously via a combining network.

Regardless
One of the following operations will be performed if processor P_1 executes $Ans_1 \leftarrow$ Fetch & Add (x, e_1) and P_2 executes $Ans_2 \leftarrow$ Fetch & Add (x, e_2) simultaneously on the shared variable x.

P_1 before P_2	P_2 before P_1
$Ans_1 \leftarrow x$	$Ans_1 \leftarrow x + e_2$
$Ans_2 \leftarrow x + e_1$	$Ans_2 \leftarrow x$.

1. Regardless of the executing order, the value $x + e_1 + e_2$ is stored in memory.
2. It is the responsibility of the switch box to form
   i) the sum $e_1 + e_2$,
   ii) transmit the combined request Fetch & Add $(x, e_1 + e_2)$,
   iii) store the value e_1 (or e_2) in a wait buffer of the switch, and
   iv) return the values x and $x + e_1$ to satisfy the original requests Fetch & Add (x, e_1) and Fetch & Add (x, e_2), respectively.

Applications and Drawbacks

Applications

1. The Fetch & Add is very effective in accessing sequentially allocated queue structures in parallel, or
2. Forking out parallel processes with identical code that operate on different data sets

Drawbacks

1. The advantage of using a combining network to implement the Fetch & Add operation is achieved at a significant increase in network cost.

2. Additional switch cycles are also needed to make the entire operation, an atomic memory operation. This will increase the network latency significantly.

CACHE COHERENCE AND SYNCHRONIZATION MECHANISMS

1. The cache coherence problem.
2. Snoopy Bus protocols
3. Directory-Based protocols.
4. Hardware synchronization Mechanisms

1. The Cache Coherence Problem.

a) Data inconsistency can occur between adjacent levels or within the same level in a memory hierarchy for a multiprocessor system.

b) multiple caches may possess different copies of the same memory block because multiple processors operate asynchronously and independently.

c) The coherence property requires that copies of the same information item at successive memory levels be consistent.

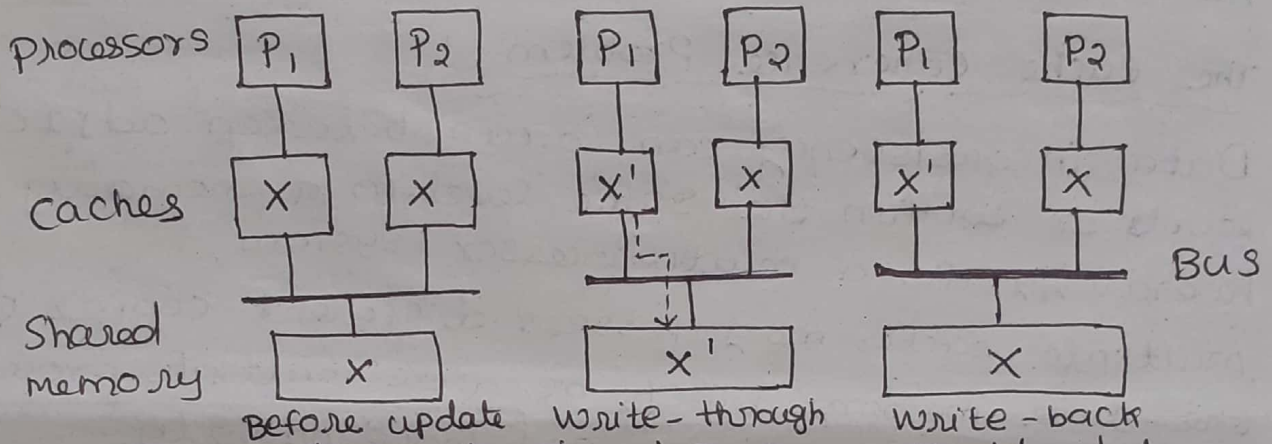
Cache coherence Problem: When multiple processors maintain locally cached copies of a unique shared-memory location, any local modification of the location can result in a globally inconsistent view of memory.

Inconsistencies in cache caused by
i. data sharing,
ii. process migration, or
iii. I/O

Inconsistency in Data Sharing

1. occurs when multiple private caches are used.
2. Three sources of the problem are identified:
   i. sharing of writable data,
   ii. process migration, and
   iii. I/O activity.

The following figure illustrates the problems caused by sharing of writable data.



(a) Inconsistency in sharing of writable data

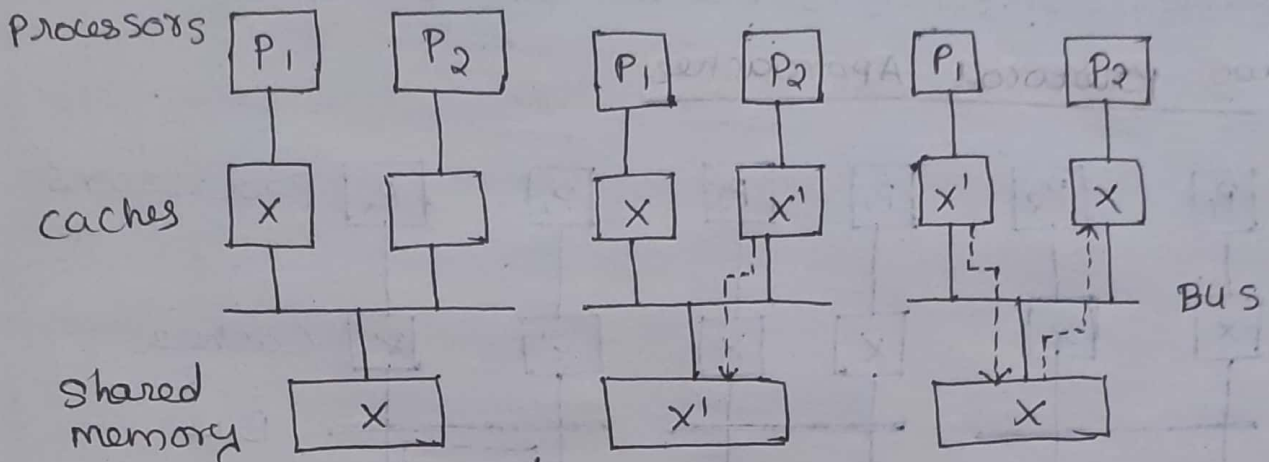
Figure 3.11 Cache coherence problems in data sharing.

a) Let x be a shared data element which has been referenced by both processors
b) Before update, the three copies of X are consistent.
c) If processor P_i writes new data x' into the cache, the same copy will be written immediately into the shared memory using a write-through policy.
d) Inconsistency also occur when a write-back policy is used. The main memory will be eventually updated when the modified data in the cache.

Process Migration

The following figure shows the occurrence of inconsistency after a process containing a shared variable

X migrates from processor 1 to processor 2 using the write-back cache on the right.



Before migration write-through write-back

(b) Inconsistency after process migration.

Figure 3.42 Cache coherence problems in process migration. In the middle, a process migrates from processor 2 to processer 1 when using write-through caches. special precautions must be exercised to avoid such inconsistencies.

I/O

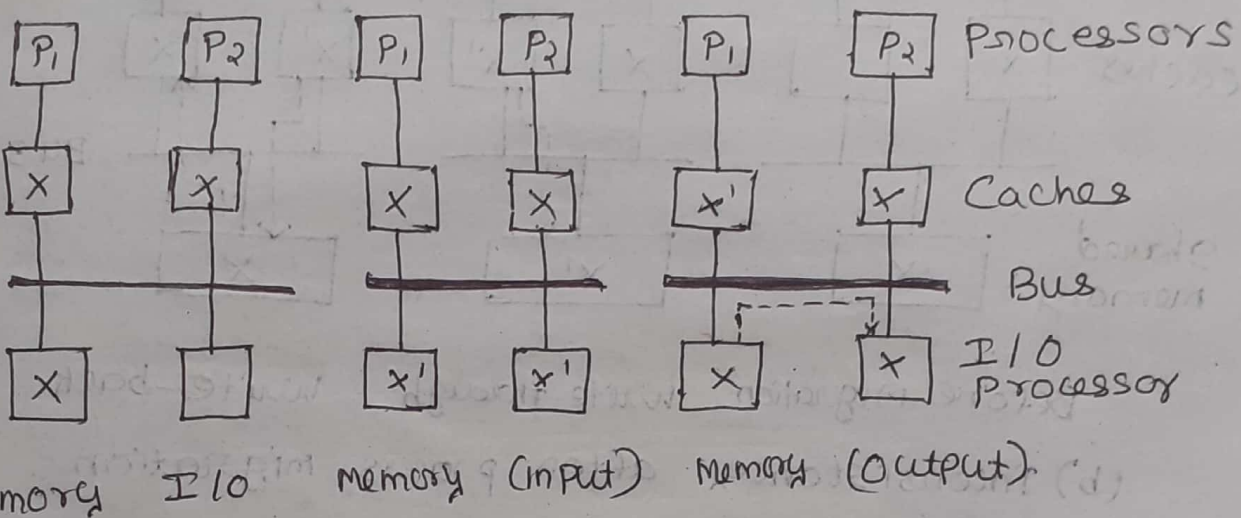
Inconsistency problems may occur during I/O operations that bypass the caches.

i. When the I/O processor loads a new data x' into the main memory, bypassing the write-through caches, inconsistency occurs between cache 1 and the shared memory.

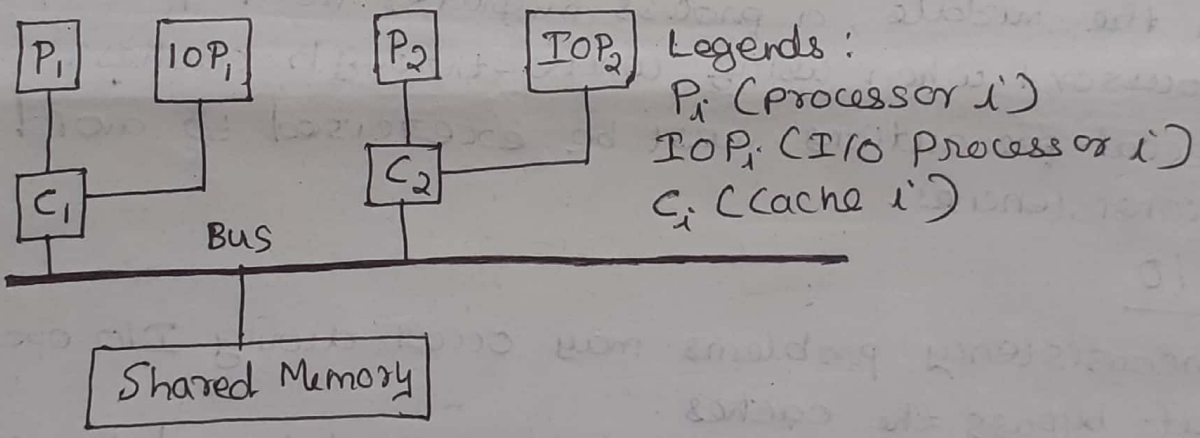
ii. when outputting a data directly from the shared memory (bypassing the caches), the write-back caches also create inconsistency.

iii. One possible solution to the I/O inconsistency problem is to attach the I/O processors to the private caches.

iv. The I/O consistency can be maintained if cache-to-cache consistency is maintained via the bus.



(a) I/O operations bypassing the cache



(b) A possible solution.

Figure 3.13

Two Protocol Approaches

1. Snoopy Protocols

A bus is a convenient device for ensuring cache coherence because it allows all processors in the system to observe ongoing memory transactions.

The consistency of a locally cached object can be controlled by the appropriate actions to invalidate the local copy by the cache controller.

2. Direct or multistage networks
Multiprocessor systems interconnect processors using short point-to-point wires.
The bandwidth of these networks increases as more processors are added to the system.

3. Directory schemes
Networks do not have a convenient snooping mechanism and do not provide an efficient broadcast capability such systems cache coherence problem can be solved using some variant of directory schemes.

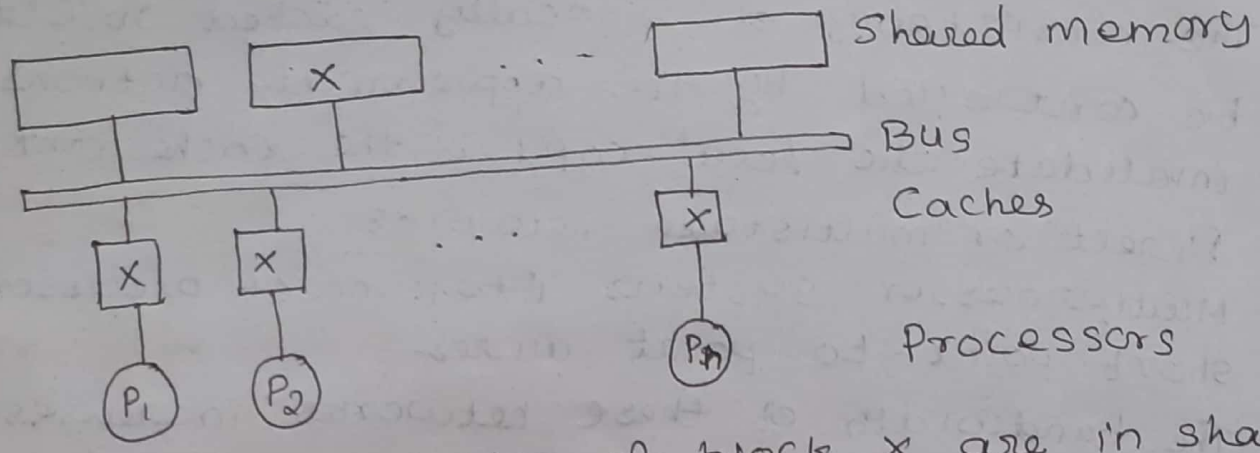
SNOOPY BUS PROTOCOLS

1. Cache consistency of private caches associated with processors tied to a common bus can be maintained by:
   1. Write-invalidate and
   2. write-update policies.

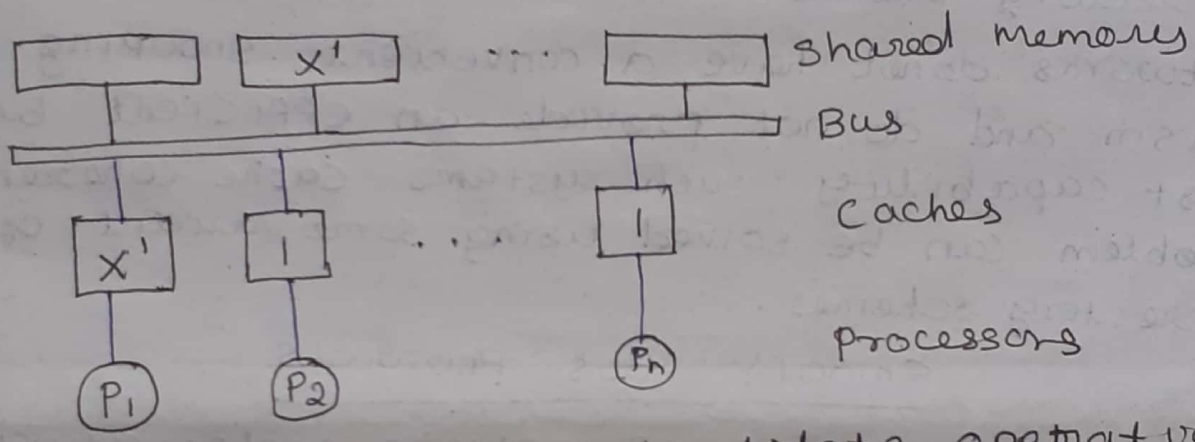
   The write-invalidate policy will invalidate all remote copies when a local cache block is updated.

   The write-update policy will broadcast the new data block to all caches containing a copy of the block.

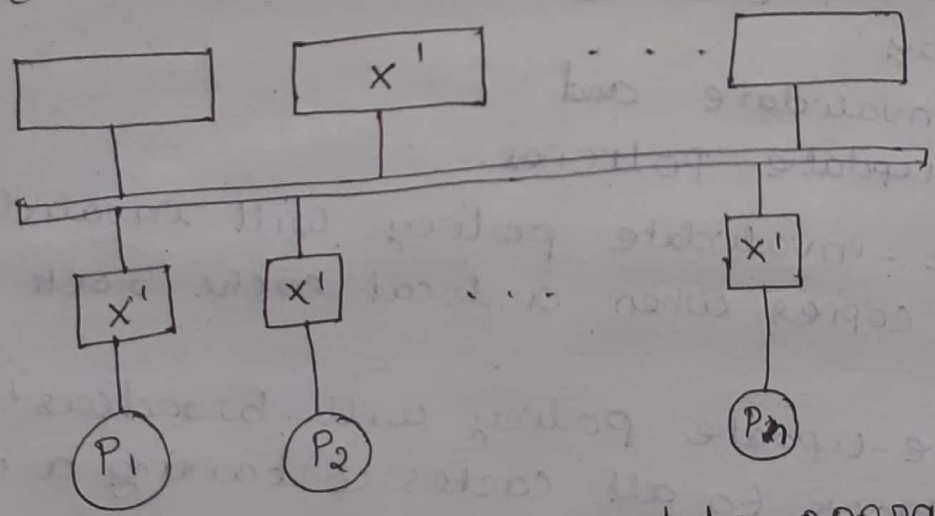
2. Snoopy protocols achieve data consistency among the caches and shared memory through a bus watching mechanism.



(a) Consistent copies of block X are in shared memory and three processor caches.



(b) After a write-invalidate operation by P_i



(c) After a write-update operation by P_i

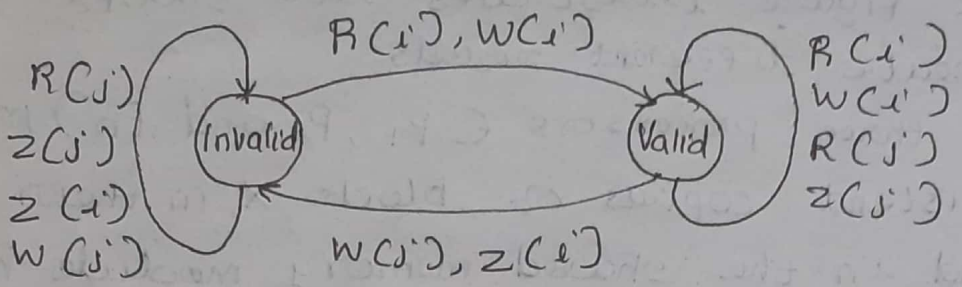
Figure 3.14 Write-invalidate and write-update coherence protocols for write-through caches (I: invalidate).

The above Figure illustrates two snoopy bus protocols create different results.

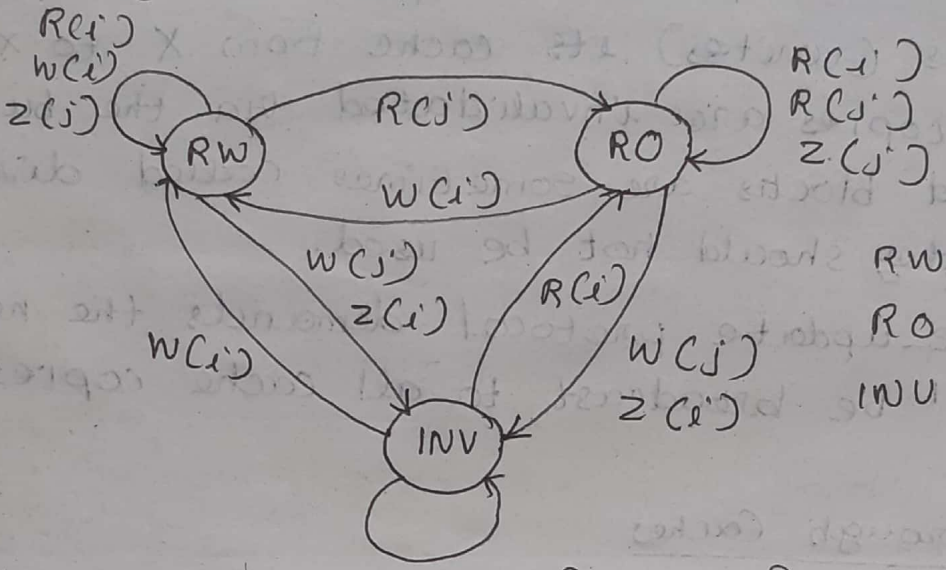
1. consider three processors (P_1, P_2 and P_n) maintaining consistent copies of block X in their local caches and in the shared-memory module marked x.
2. Using a write-invalidate protocol, the processor P_1 modifies (writes) its cache from X to x', and all other copies are invalidated via the bus. Invalidated blocks are sometimes called dirty, meaning they should not be used.
3. The write-update protocol demands the new block content x' be broadcast to all cache copies via the bus.

Write-Through Caches

1. The states of a cache block copy change with respect to read, write, and replacement operations in the cache.
2. The following Figure shows the state transitions for two basic write-invalidate snoopy protocols developed for write-through and write-back caches respectively.
3. A remote processor is denoted j, where $j \neq i$. For each of the two cache states, six possible events may take place.
4. All cache copies of the same block use the same transition graph in making state changes.



(a) Write-through cache



RW: Read-Write
RO: Read Only
INV: Invalidated or not in cache

W(i') = Write to block by processor i'.
R(i') = Read block by processor i'.
Z(i') = Replace block in cache i'.
W(j') = Write to block copy in cache j by processor j' ≠ i'.
R(j') = Read block copy in cache j by processor j' ≠ i'.
Z(j') = Replace block copy in cache j' ≠ i'

(b) Write-back cache

Figure 3.15 Two state-transition graphs for a cache block using write-invalidate snoopy protocols.

5. In a valid state (a), all processors can read (R(i), R(j)) safely. Local processor i can also write (W(i)) safely in a valid state.
6. The invalid state corresponds to the case of the block either being invalidated or being replaced (Z(i) or Z(j)).
7. Whenever a remote processor write (W(j)) into its cache copy, all other cache copies become invalidated.
8. The cache block in cache i becomes valid whenever a successful read (R(i)) or write (W(i)) is carried out by a local processor i.

Write-Back Caches

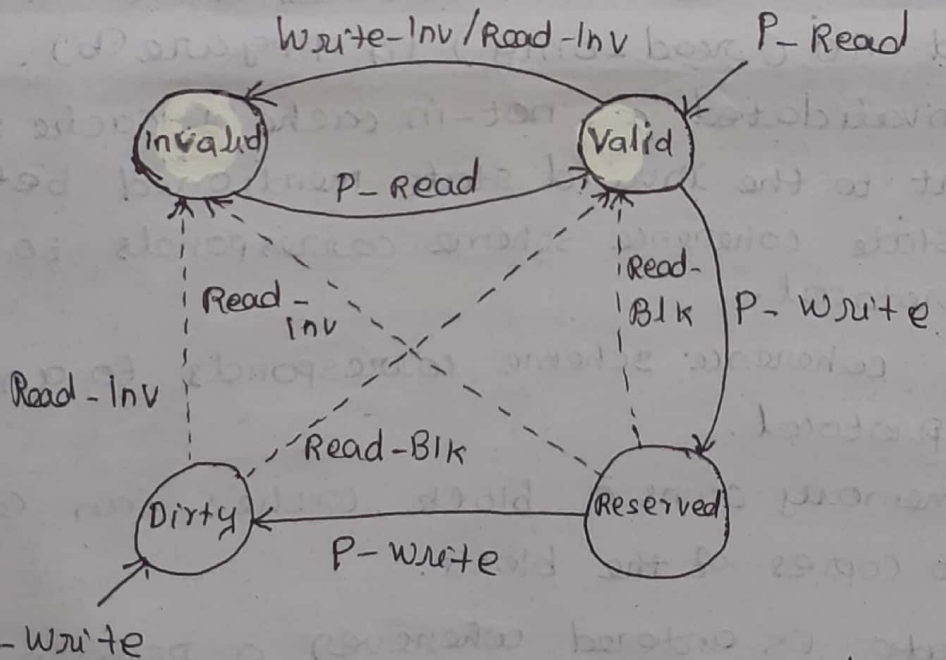
1. The valid state of a write-back cache can be further split into two cache states, labeled RW (read-write) and RO (read-only) in Figure (b).
2. The INV (invalidated or not-in cache) cache state is equivalent to the invalid state mentioned before. This three-state coherence scheme corresponds to an ownership protocol.
3. Three-state coherence scheme corresponds to an ownership protocol.
4. When the memory owns a block, caches can contain only the RO copies of the blocks.
5. The INV state is entered whenever a remote processor writes (W(j)) its local copy or the local processor replaces (Z(i)) its own block copy.
6. The RW state corresponds to only one cache copy existing in the entire system owned by the local processor i.

7. Read (RC(i)) and write (WC(i)) can be safely performed in the RW state.
8. From either the RO state or the INV state, the cache block becomes uniquely owned when a local write (W(i)) takes place.

Write-once Protocol

1. proposed by James Goodman (1983)
2. combines the advantages of both write-through and write-back invalidations.
3. To reduce bus traffic, the very first write of a cache block uses a write-through policy.
4. After the first write, shared memory is updated using a write-back policy.

The four-state transition graph is as follows



Solid lines: Command issued by local processor
Dashed lines: Commands issued by remote processors via the system bus.

Figure 3.16 Goodman's write-once cache coherence protocol using the write-invalidate policy on write-back caches.

States are defined as

1. Valid : The cache block, which is consistent with the memory copy, has been read from shared memory and has not been modified.
2. Invalid : The block is not found in the cache or is inconsistent with the memory copy.
3. Reserved : Data has been written exactly once since being read from shared memory. The cache copy is consistent with the memory copy, which is the only other copy.
4. Dirty : The cache block has been modified (written) more than once, and the cache copy is the only one in the system (thus inconsistent with all other copies).

The protocol requires two different sets of commands to maintain consistency.

1. read-miss, write-hit and write-miss. (solid lines)
   a) whenever a read-miss occurs, the valid state is entered.
   b) The first write-hit leads to the reserved state.
   c) The second write-hit leads to the dirty state, and
   d) all future write-hits stay in the dirty state.
   e) whenever a write-miss occurs, the cache block enters the dirty state.
2. The dashed lines correspond to invalidation commands issued by remote processors via the snoopy bus.
   a) The read-invalidate command reads a block and invalidates all other copies.
   b) The write-invalidate command invalidates all other copies of a block.

c) The bus-read command corresponds to a normal memory read by a remote processor via the bus.

Cache Events and Actions

1. Read miss:

a) When a processor wants to read a block that is not in the cache, a read-miss occurs.

b) A bus-read operation will be initiated.

c) If no dirty copy exists, then main memory has a consistent copy and supplies a copy to the requesting cache.

d) If a dirty copy does exist in a remote cache, that cache will inhibit the main memory and send a copy to the requesting cache.

e) In all cases, the cache copy will enter the valid state after a read-miss.

2. write-hit:

a) If the copy is in the dirty or reserved state, the write can be carried out locally and the new state is dirty.

b) If the new state is valid, a write-invalidate command is broadcast to all caches, invalidating their copies.

c) The shared memory is written through, and the resulting state is reserved after this first write.

d)

3. write-miss:

a) When a processor fails to write in a local cache, the copy must come either from the main memory or from a remote cache with a dirty block.

b) This is accomplished by sending a read-invalidate command which will invalidate all cache copies.

c) The local copy is updated and ends up in a dirty state.

4. Read-hit:
Read-hits can always be performed in a local cache without causing a state transition or using the snoopy bus for invalidation.

5. Block Replacement:
a) If a copy is dirty, it has to be written back to main memory by block replacement.
b) If the copy is clean (i.e., in either the valid, reserved, or invalid state), no replacement will take place.

The Futurebus+ Protocol

1. Goodman's write-once protocol demands special bus lines to inhibit the main memory when the memory copy is invalid, and a bus-read operation is needed after a read-miss.
2. The IEEE Futurebus+ proposed to include this special bus provision.
3. Using a write-through policy after the first write and using a write-back policy in all ~~traditional~~ additional writes will eliminate unnecessary invalidations.
4. Snoopy cache protocols are popular in bus-based multiprocessors because of their simplicity of implementation.
5. The Futurebus+ parallel protocols support both connected and split transactions.
connected - ~~simple~~ cheaper, easier, single bus.
split - complex, expensive, greater concurrency in building large hierarchical-bus multiprocessors.
Bus repeaters are used to support split transactions - a copy of a transaction must be repeated on adjacent buses having the same cache block.

The snoopy coherence protocol works for connected transactions. For split transactions over multiple bus segments, additional interface modules are needed.

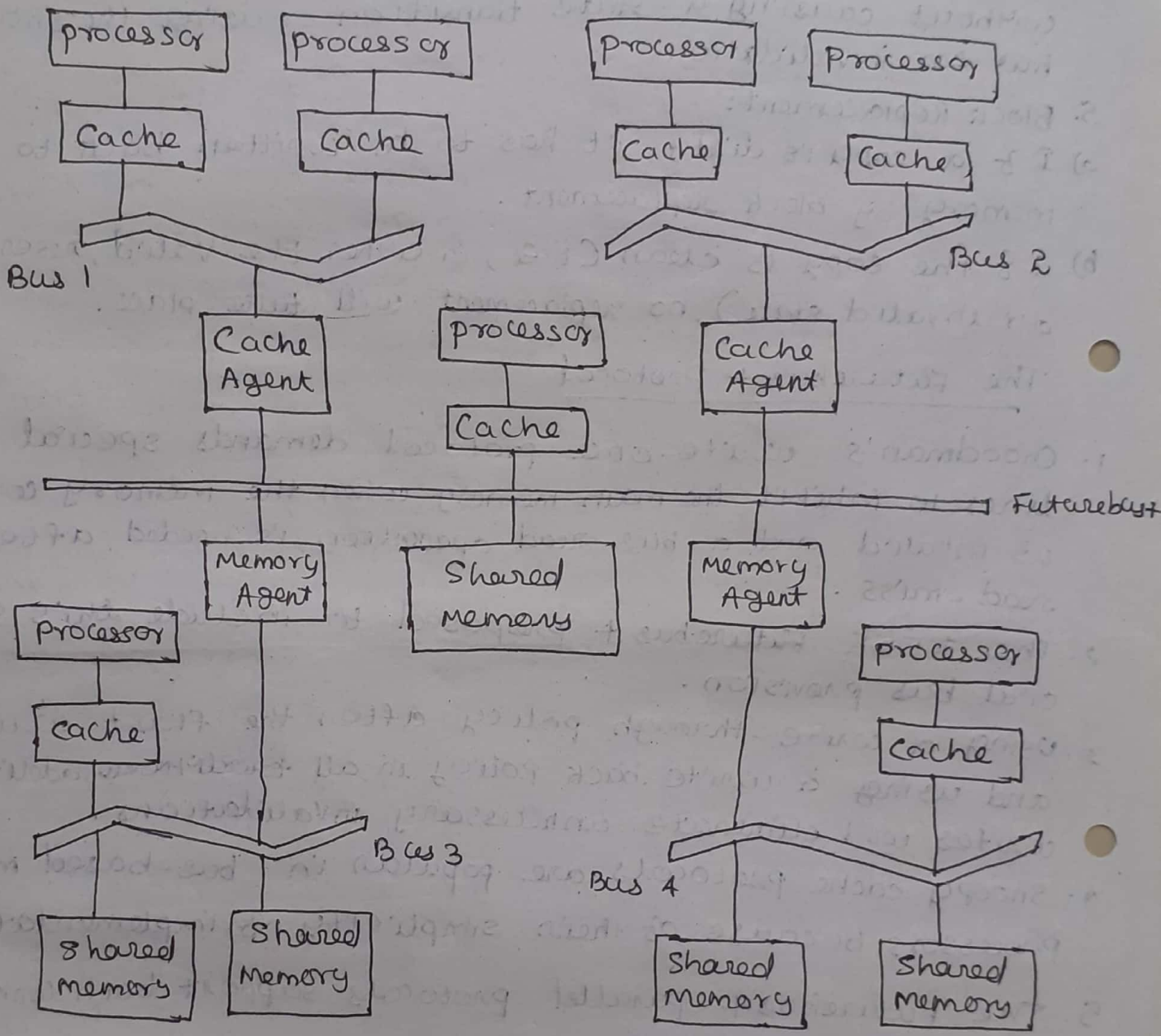


Figure 3.17 Split transactions on a multiprocessor.

1. This system is an extension of the shared-memory architecture with cache agent and memory agent modules.
2. A cache agent uses split transactions to assume all the rights and responsibilities of a number of remote cache modules.

3. Cache agents must maintain tags indicating the cache block states.
4. A memory agent uses split transactions to assume all the rights and responsibilities of some remote memory modules.
5. A memory agent splits a transaction to assign read and write permissions to the location it is handling for the bus segment it is on.
6. Split transactions have been used to delay invalidation completions.
7. Various cache coherence commands are needed on the buses.

Protocol Performance Issues

1. The performance of any snoopy protocol depends on
   1 - the workload patterns and
   2 - implementation efficiency.
2. The main motivation for using the snooping mechanism is to reduce bus traffic, reduce the effective memory-access time.
3. For a uniprocessor system, bus traffic and memory-access time are mainly contributed by cache misses.
4. For a system requiring extensive process migration or synchronization, the write-invalidate protocol will perform better.
5. Bus traffic in a multiprocessor may increase when the block size increases.
6. The write-update protocol requires a bus broadcast capability.

Directory-Based Protocols

1. Write-invalidate protocol – heavy bus traffic.
2. Write-update protocol – update data items in remote caches which will never be used by other processors.
3. For multistage network it is not possible to apply snoopy protocol directly because broadcasting is very expensive in multi stage networks.
4. In multistage network consistency commands will be sent only to those caches that keep a copy of the block – directory-based protocols for network-connected multiprocessors.

a) Directory structures
b) Full-Map Directories
c) Limited Directories
d) Chained Directories.
e) Cache Design Alternatives
f) Shared Caches
g) Noncacheable Data
h) Cache Flushing.

Directory Structures

1. To store information on where copies of cache blocks reside.
2. Various directory-based protocols differ mainly in how the directory maintains information and what information it stores.
3. Tang (1976) – proposed first directory scheme – Central directory containing duplicates of all cache directories – very large, associatively searched. Disadvantages – contention and long search.

4. Censier and Feautrier (1978) – proposed distributed-directory – each memory module maintains a seperate directory.
State information is local,
presence information endicates which caches have a copy of the block.

Cache directory

1. A cache-coherence protocal that does not use broadcasts must store the locations of all cached copies of each block of shared data. This list of chat cached locations, whether centralized or distributed, is called a cache directory.
2. A directory entry for each block of data contains a number of pointers to specify the locations of copies of the block.
3. Each directory entry contains a dirty bit to specify whether a unique cache has permission to write the associated block of data.

Categories of directory protocols are:
i. full-map directories,
ii. limited directories, and
iii. chained directories.

Full-Map Directories.

1. Each directory entry contains N pointers, where N is the number of processors in the system.
2. The full-map protocol implements directory entries with one bit per processor and a dirty bit.
3. Figure 3.18 illustrates three different states of a full-map directory.

1. A cache maintains two bits of state per block. One bit indicates whether a block is valid, and the other indicates whether a valid block may be written.

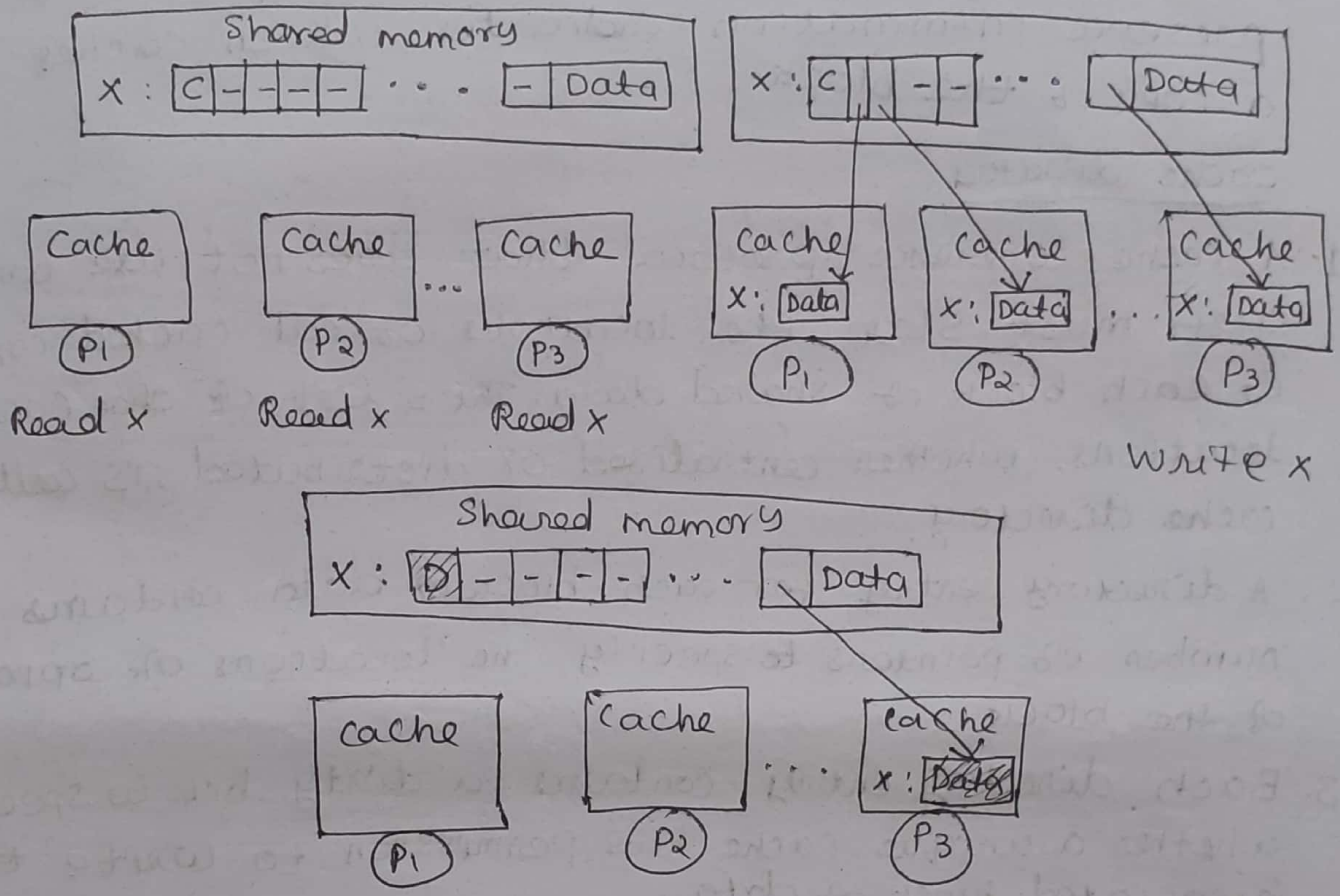


Figure 3.18 Three states of a full-map directory

Figure:

i. In the first state, location x is missing in all of the caches in the system

ii. In the second state results from three caches (C1, C2, and C3) requesting copies of location x.

iii. The third state results from cache C3 requesting write permission for the block.

The dirty bit set to dirty (D), and there is a single pointer to the block of data in cache C3.

The transition from the second state to the third state in more detail

1. Cache C3 detects that the block containing location X is valid but that the processor does not have permission to write to the block, indicated by the block's write_permission bit in the cache.
2. Cache C3 issues a write request to the memory module containing location X and stalls processor P3.
3. The memory module issues invalidate requests to caches C_1 and C_2.
4. Caches C_1 and C_2 receive the invalidate requests, set the appropriate bit to indicate that the block containing location X is invalid, and send acknowledgements back to the memory module.
5. The memory module receives the acknowledgments sets the dirty bit, clears the pointers to caches C_1 and C_2, and sends write permission to cache C3.
6. Cache C3 receives the write permission message, updates the state in the cache, and reactivates processor P3.

Limited Directories.

1. Limited directories have a fixed number of pointers per entry, regardless of the system size.
2. These protocols are designed to solve the directory size problem.
3. If the multiprocessor exhibits processor locality in the sense that in any given interval of time only a small subset of all the processors access a given memory word, then a limited directory is sufficient to capture this small worker set of processors.

4. Figure 3.19 shows the situation when three caches request read copies in a memory system.
5. In this case, we can view the two-pointer directory as a two-way set-associative cache of pointers to shared copies.

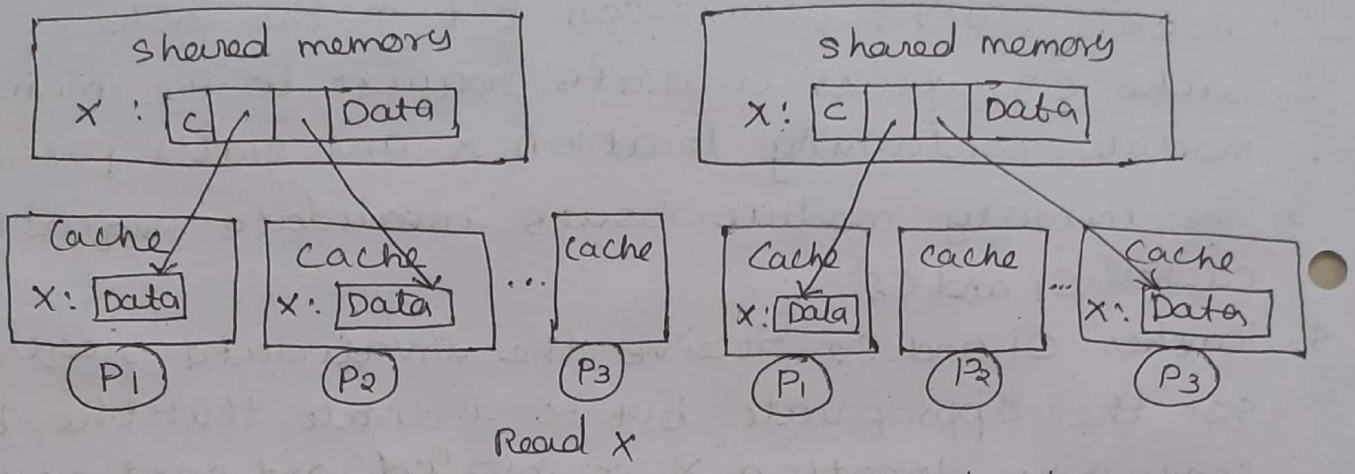


Figure 3.19 Eviction in a limited directory.

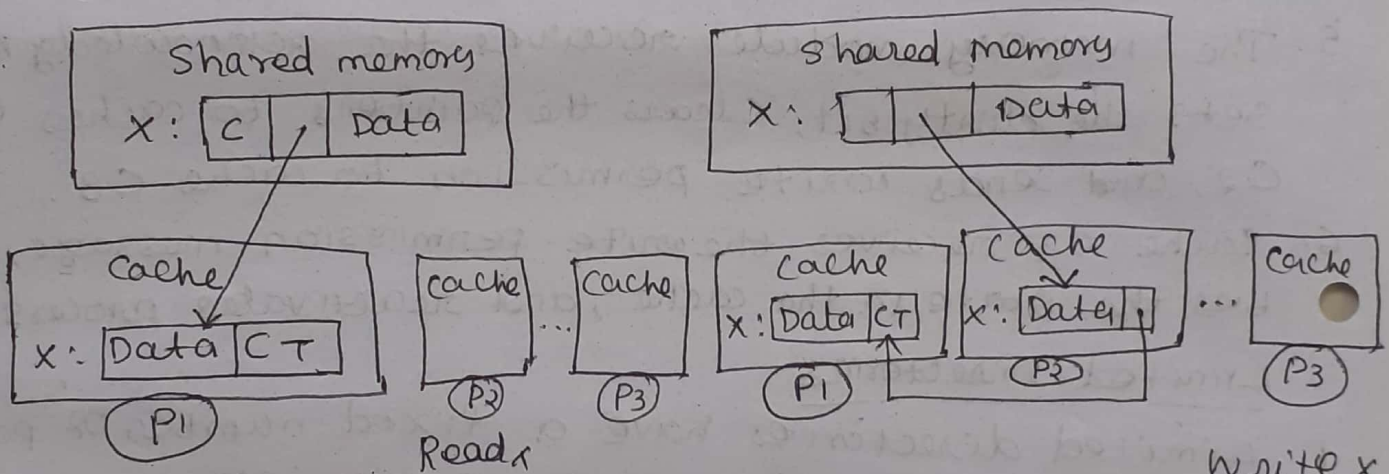


Figure 3.20 The chained directory.

Chained Directories

1. Chained directories emulate the full-map schemes by distributing the directory among the caches.
2. It keeps track of shared copies of data by maintaining a chain of directory pointers.

Figure 3.20 shows the scheme.

3. Suppose there are no shared copies of location X.
4. If processor Pi reads location X, the memory sends a copy to cache Ci, along with a chain termination (CT) pointer.
5. Suppose that caches C1 through CN all have copies of location X and that location X and location Y map to the same (direct-mapped) cache line. If processor Pi reads location Y, it must evict location X from its cache with the following possibilities.

1) send a message down the chain to cache C_{i-1} with a pointer to cache C_{i+1} and splice C_i out of the chain, or
2) Invalidate location X in cache C_{i+1} through cache C_N.

Hardware Synchronization Mechanisms

1. Synchronization is a special form of communication in which control information is exchanged, instead of data, between communicating processes residing in the same or different processors.
2. Synchronization can be implemented in
   i. Software,
   ii. firmware, and
   iii. hardware

   through controlled sharing of data and control information in memory.

Atomic operations

1. read
2. write or
3. read-modify-write

Some interprocessor interrupts can be used for synchronization purposes.

Eg:

```
Test&set (lock)
        temp ← lock; lock ← 1;
        return temp

Reset (lock)
        lock ← 0
```

Test & set is implemented with atomic read-modify-write memory operations.

To synchronize concurrent processes, the software may repeat test & set until the returned value (temp) becomes 0.

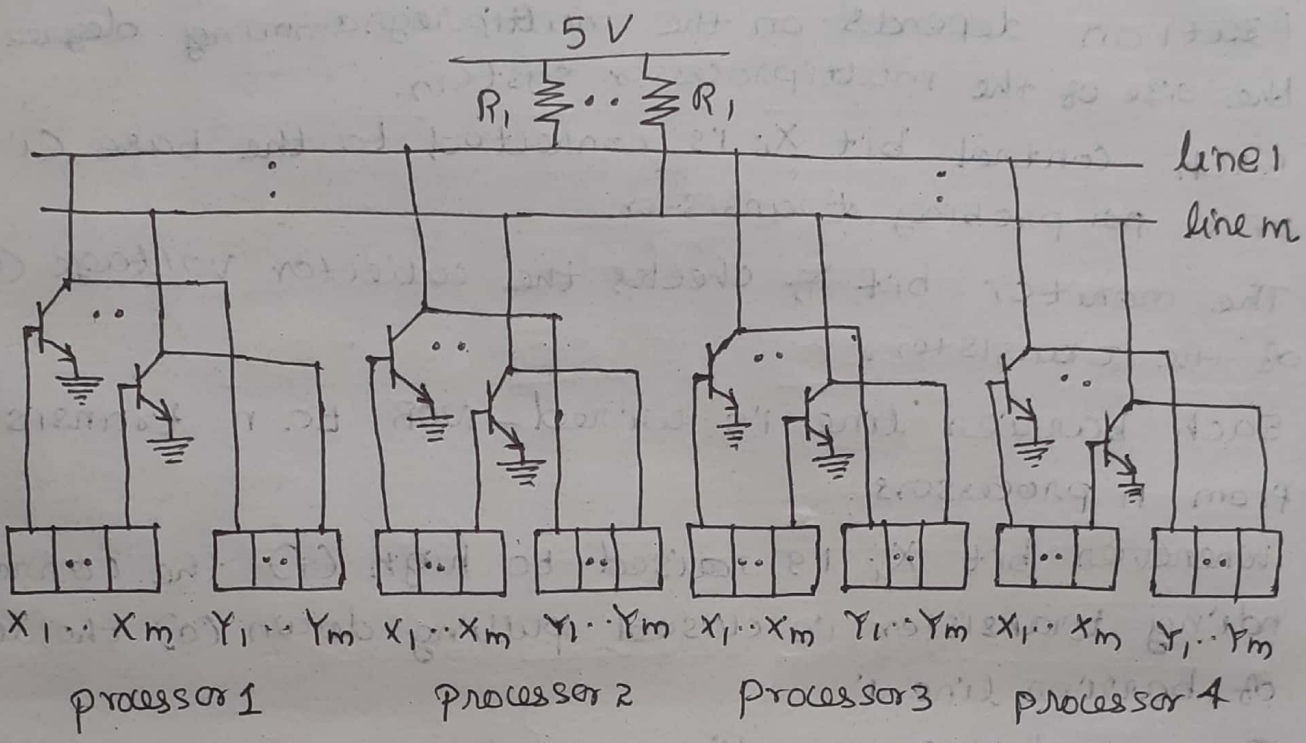
Synchronization on Futurebus+

1. Concurrent processes residing in different processors can be synchronized using barriers.
2. A barrier can be implemented by a shared-memory word which keeps counting the number of processes reaching the barrier.
3. After all processes have updated the barrier counter, the synchronization point has been reached.
4. No processor can execute beyond the barrier until the synchronization process is complete.

Wired Barrier Synchronization

A wired-NOR logic is shogic is shown in Figure 3.21 for implementing a barrier mechanism for fast synchronization.

- Each processor uses a dedicated control vector $X = (X_1, X_2, \cdots, X_m)$ and accesses a common monitor vector $Y = (Y_1, Y_2, \cdots, Y_m)$ in shared memory, when m corresponds to the barrier lines used.



(a) Barrier lines and interface logic

Step 1: Forking (case of one barrier line)

	processor 1	processor 2	processor 3	processor 4
Line 1				
X	1	1	1	1
Y	0	0	0	0

Step 2: process 1 and process 3 reach the synchronization point

X	0	1	0	1
Y	0	0	0	0

Step 3: All processes reach the synchronization point

X	0	0	0	0
Y	1	1	1	1
	process 1	process 2	process 3	process 4

(b) Synchronization steps

Figure 3.21

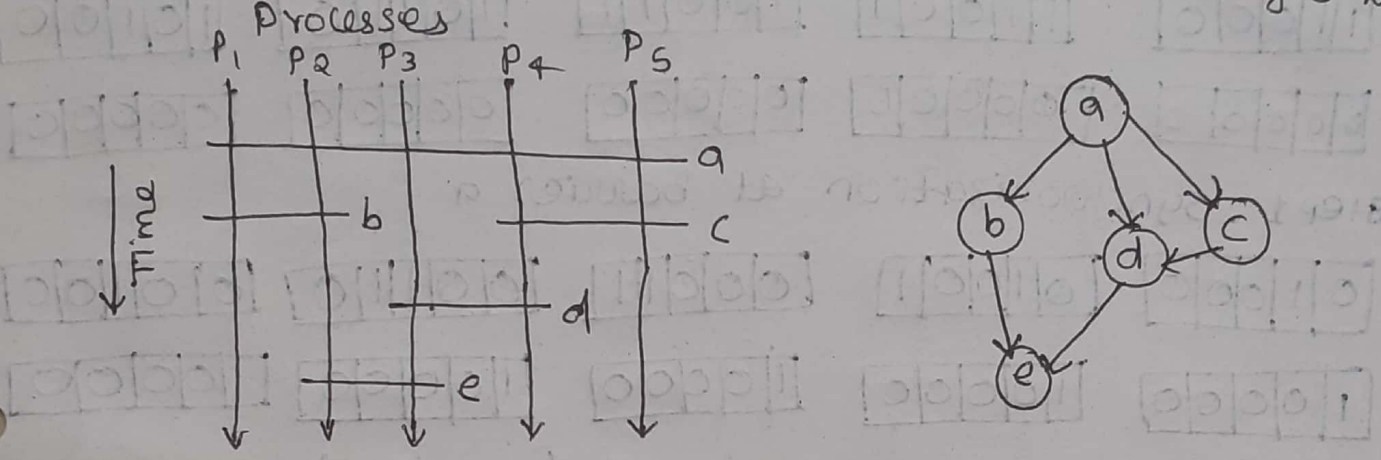
Figure 3.21 The synchronization of four independent processes on four processors using one wired-NOR barrier line.

2. The number of barrier lines needed for synchronization depends on the multiprogramming degree and the size of the multiprocessor system.
3. Each control bit X_i is connected to the base (input) of a probing transistor.
4. The monitor bit Y_i checks the collector voltage (output) of the transistor.
5. Each barrier line is wired-NOR to n transistors from n processors.
6. Whenever bit X_i is raised to high (1), the corresponding transister is closed, pulling down (0) the level of barrier line i.
7. The wired-NOR connection implies that line i will be high (1) only if control bits X_i from all processors are low (0).
8. When all processes finish their jobs, the X_i bits from the participating processors are all set to 0; and the barrier line is then raised to high (1), signaling the synchronization barrier has been crossed.
9. Figure 3.21 shows the synchronization of four processes residing on four processors using one barrier line.

Example.

Wired barrier synchronization of five partially ordered processes (Hwang and Shang, 1991).

If the synchronization pattern is predicted after compile time, then one can follow the precedence graph of a partially ordered set of processes to perform multiple synchronization as demonstrated in Fig. 3.22.



(a) Synchronization patterns. (b) Precedence graph.

Figure 3.22

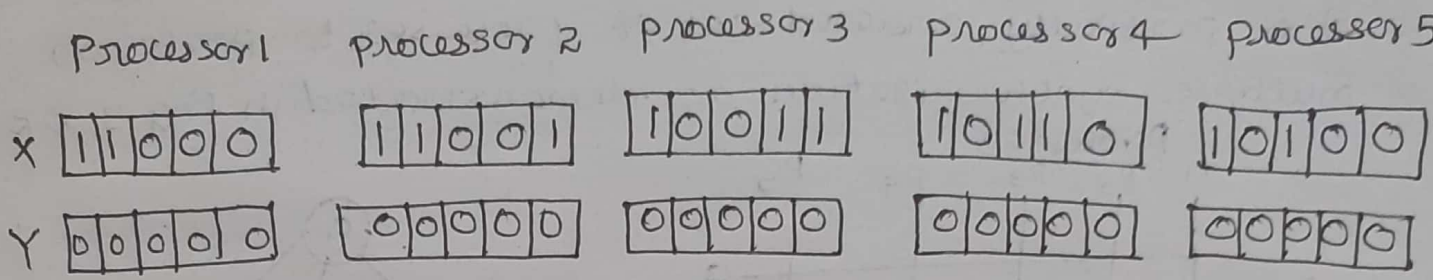
Here five processes ($P_1, P_2, \ldots, P_5$) are synchronized by snooping on five barrier lines corresponding to five synchronization points labeled a, b, c, d, e. At step 0 the control vectors need to be initialized. All five processes are synchronized at point a. The crossing of barrier a is signaled by monitor bit Y_1, which is observable by all processors.

Barriers b and c can be monitored simultaneously using two lines as shown in steps 2a and 2b. Only four steps are needed to complete the entire process.

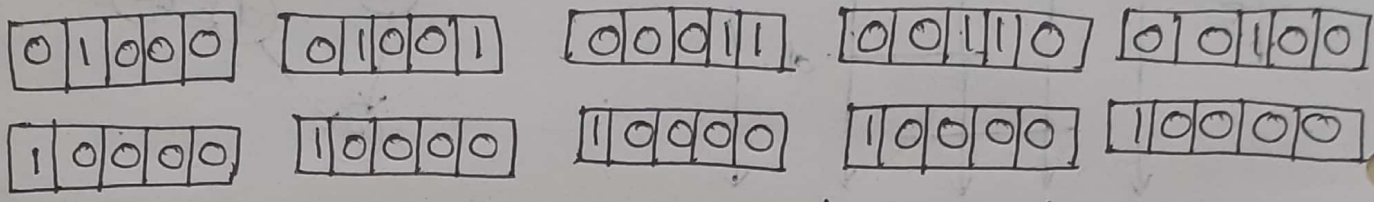
Only one copy of the monitor vector Y is maintained in the shared memory.

The bus interface logic of each processor module has a copy of Y for local monitoring purposes as shown in Fig. 2.2.c.

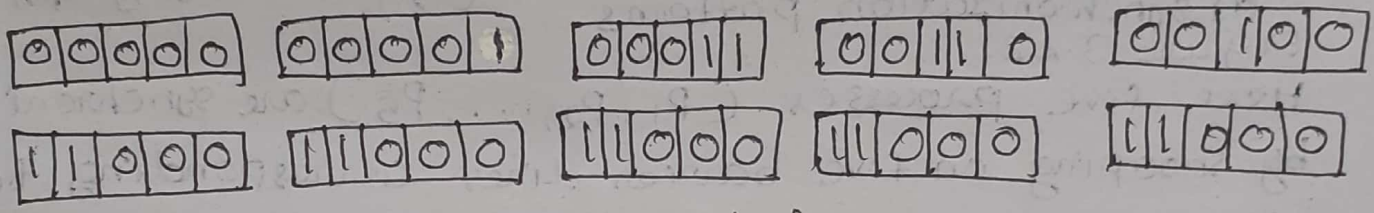
Step 0: Initializing the control vectors (use 5 barrier lines)



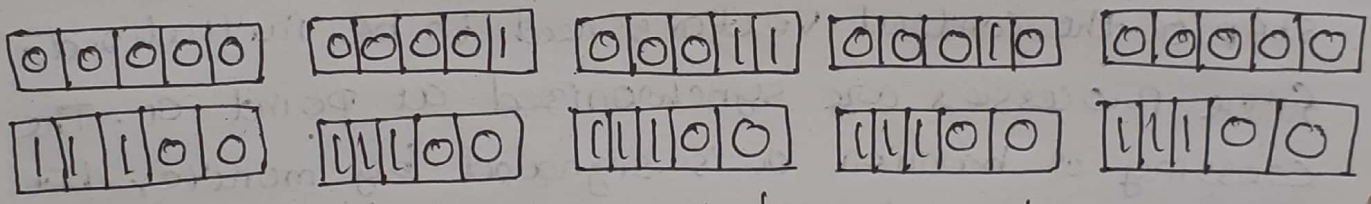
Step 1: Synchronization at barrier a



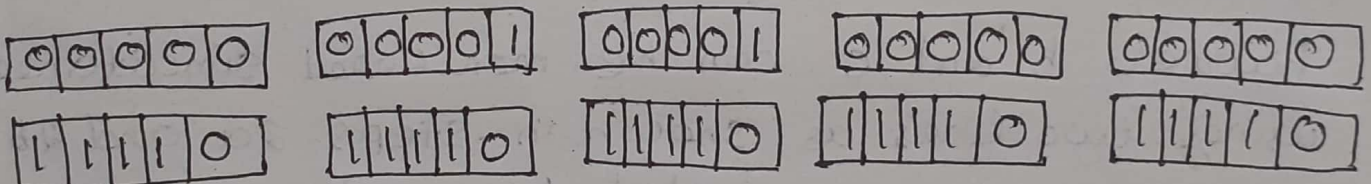
Step 2a: Synchronization at barrier b



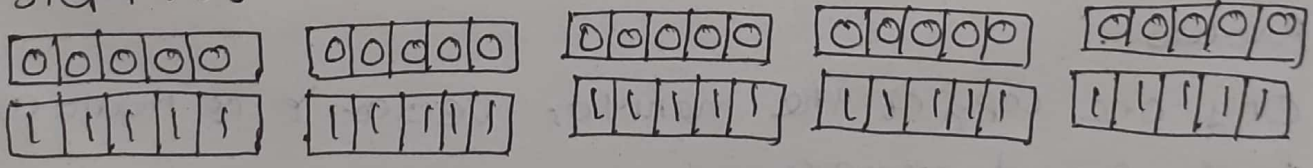
Step 2b: Synchronization at barrier c



Step 3: Synchronization at barrier d



Step 4: Synchronization at barrier e.



(c) Synchronization steps.

Figure 3.22. The synchronization of five partially ordered processes using wired-NOR barrier lines.

10. seperate control vectors are used in local processors.
11. The above dynamic barrier synchronization is possible only if the synchronization pattern is predicted at compile time and process.
12. One can also use the barrier wires along with counting semaphores in memory to support multi-programmed multiprocessors in which preemption is allowed.

Module IV

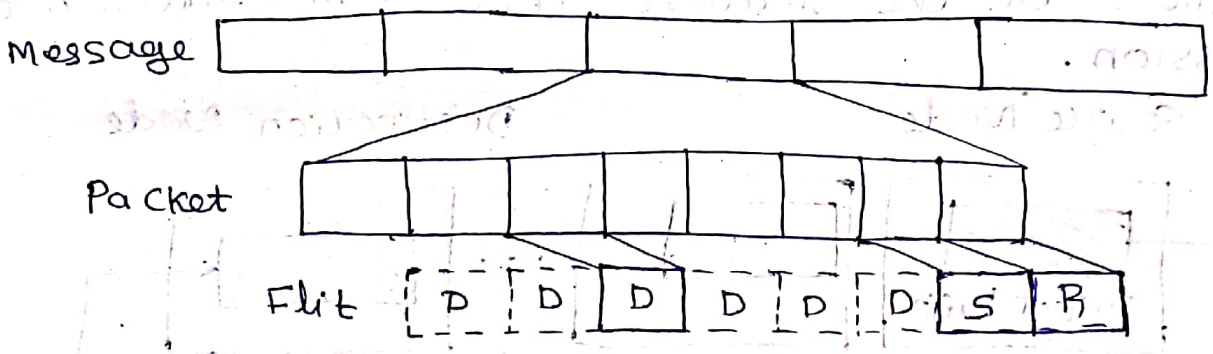
Message passing Mechanisms

Message passing in multicomputer networks demands special hardware and software support.

Message-Routing Schemes

1. Message Formats.
2. Store-and-Forward Routing
3. Wormhole Routing
4. Asynchronous Pipelining
5. Latency Analysis.

1. Message Formats.



R: Routing information
S: Sequence Number
D: Data only flits.

Figure 4.1 The format of message, Packets and flits (control flow digits) used as information units of communication in a message-passing network.

1. A message is the logical unit for internode communication.
   a) assembled by an arbitrary number of fixed-length packets
   b) variable length.

2. A packet is the basic unit containing the destination address for routing purposes.
   a) Different packets arrive at the destination asynchronously.
   b) a sequence number is needed in each packet to allow reassembly of the message transmitted.
3. A packet can be devided into a number of fixed-length flits.
   a) Routing information (destination) and sequence number occupy the header flits.
   b) The remaining flits are the data elements

2. Store-and-Forward Routing

i. In multicomputers with store-and-forward routing, packets are the smallest unit of information transmission.

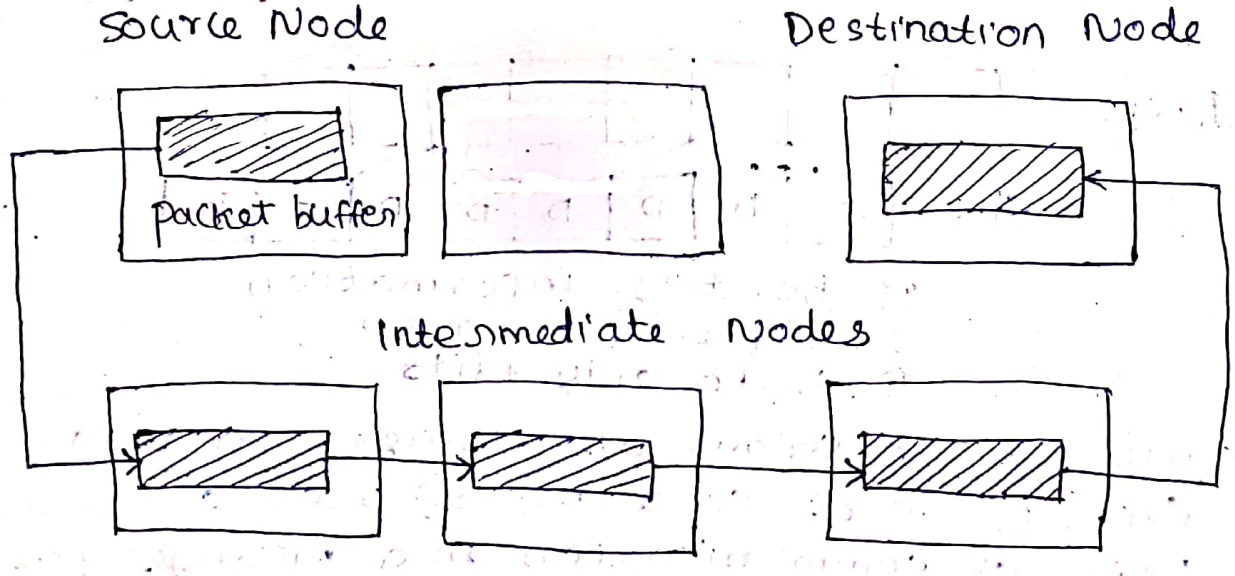


Figure 4.2 Store-and-forward routing using packet buffers in successive nodes

ii. The concept of store-and-forward routing is illustrated in Figure 4.2.
   a) Each node is required to use a packet buffer.

b) A packet is transmitted from a source node to a destination node through a sequence of intermediate nodes.

c) When a packet reaches an intermediate node, it is first stored in the buffer. Then it is forwarded to the next node if the desired output channel and packet buffer in the receiving node are both available.

d) The latency in store-and-forward networks is directly proportional to the distance (the number of hops) between the source and the destination.

e) This routing scheme was implemented in the first-generation of multicomputers.

3. Wormhole Routing

i. In wormhole-routed networks, packets are further sub-divided into flits.

ii. Newer multicomputers implement the wormhole routing scheme as illustrated in Figure 4.3.

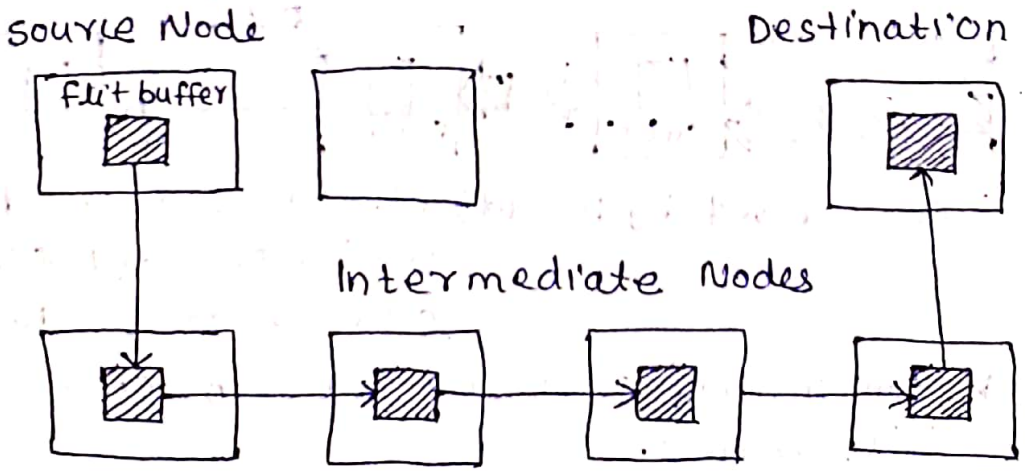


Figure 4.3 Wormhole routing using flit buffers in successive routers.

iii. Flit buffers are used in the hardware routers attached to nodes.

iv. The packet can be visualized as a railroad train with an engine car (the header flit) towing a long sequence of box cars (data flits).

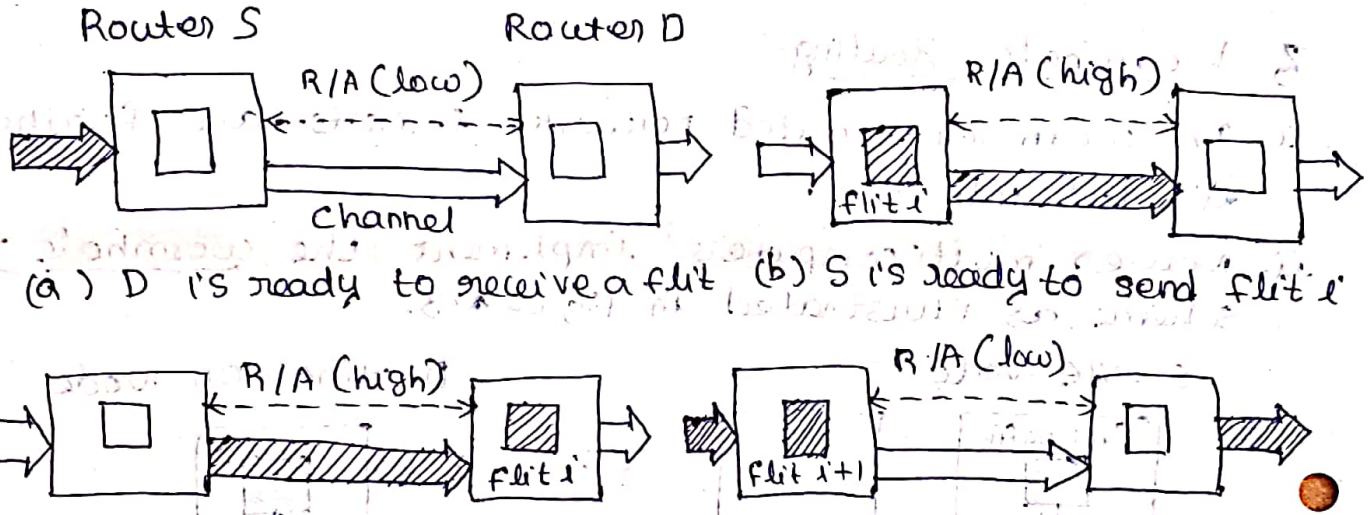
v. Only the header flit knows where the train (packet) is going.

vi. All the data flits (box cars) must follow the header flit.

vii. The flits from different packets cannot be mixed up.

1. Asynchronous Pipelining

The pipelining of successive flits in a packet is done asynchronously using a handshaking protocol as shown in Figure 4.4.



(a) D is ready to receive a flit (b) S is ready to send flit i

(c) Flit i is received by D (d) Flit i is removed from D's buffer and flit i+1 arrives at S's buffer.

Figure 4.4 Handshaking protocol between two wormhole routers.

i. Along the path, a 1-bit ready/request (R/A) line is used between adjacent routers.

ii. When the receiving router (D) is ready (Figure 4.4 a) to receive a flit (i.e., the flit buffer is available), it pulls the R/A line low.

iii. When the sending router (S) is ready (Figure 4.4 b) it raises the line high and transmits flit 1' through the channel.

iv. While the flit is being received by D (Figure 4.4 c) the R/A line is kept high.

v. After flit 1' is removed from D's buffer (i.e., is transmitted to the next node) (Figure 4.4d), the cycle repeats itself for the transmission of the next flit 1'+1 until the entire packet is received.

5. Latency Analysis

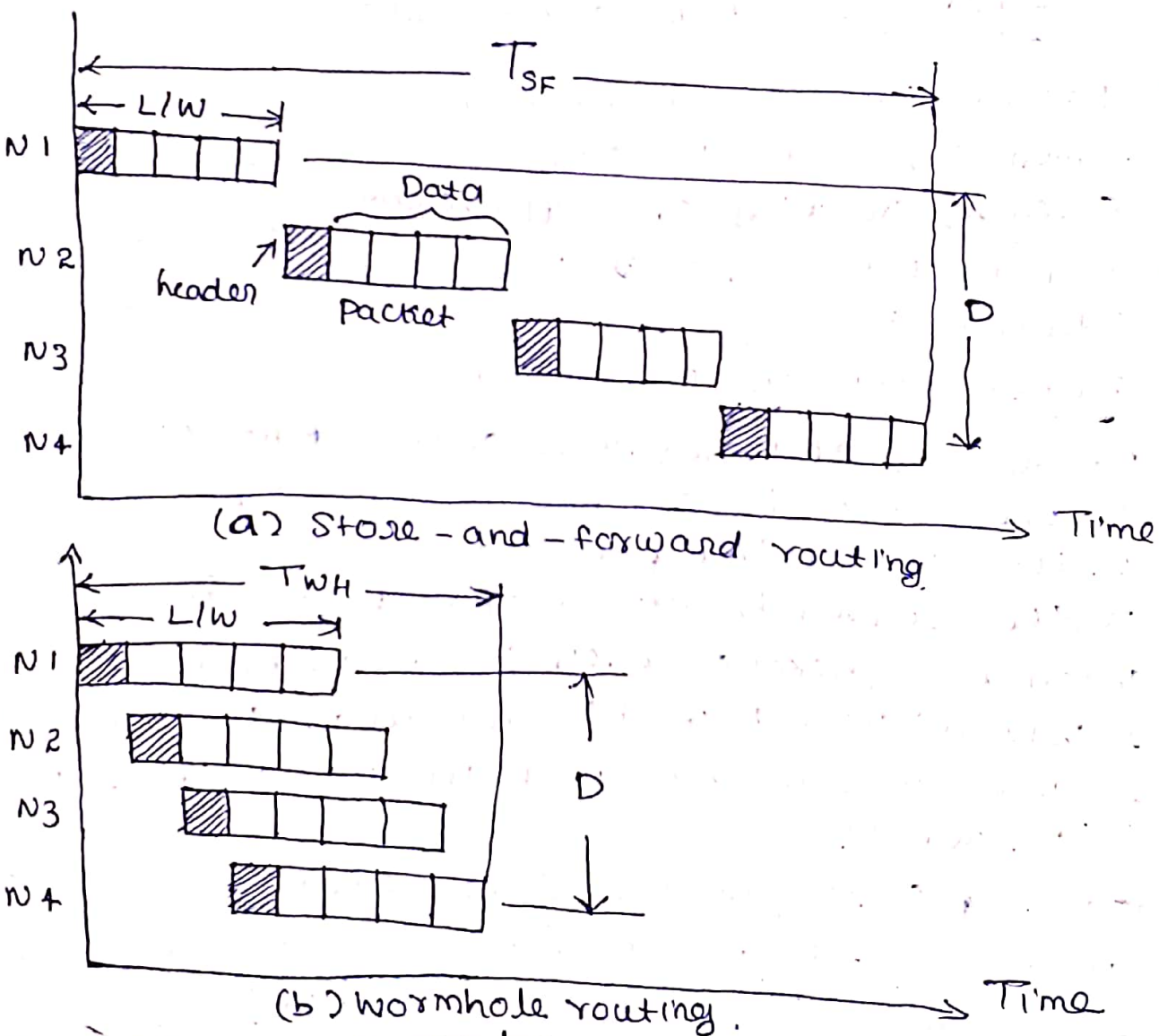


Figure 4.5 Time comparison between the two routing techniques.

Let L be the packet length (in bits), W the channel bandwidth (in bits/s), D the distance (number of nodes traversed minus 1), and F the flit length (in bits).

The communication latency T_{SF} for a store-and-forward network is

$$T_{SF} = \frac{L}{W}(D+1), \quad T_{SF} \propto D$$

$2000 < T_{SF} < 6000\ \mu s.$

The latency T_{WH} for a wormhole-routed network is

$$T_{WH} = \frac{L}{W} + \frac{F}{W} \times D, \quad L >> F$$

$T_{WH} < 5\ \mu s.$

FLOW CONTROL STRATEGIES

1. Packet Collision Resolution
2. Dimension-Order Routing.
3. E-Cube Routing on Hypercube
4. X-Y Routing on a 2D Mesh
5. Adaptive Routing

1. Packet Collision Resolution

To move a flit between adjacent nodes:

1. the source buffer holding the flit,
2. the channel being allocated, and
3. the receiver buffer accepting the flit.

When two packets reach the same node:

i. which packet will be allocated the channel and

ii. what will be done with the packet being denied the channel?

Figure 1.6 illustrates four methods for resolving the conflict between two packets competing for the use of the same outgoing channel at an intermediate node.

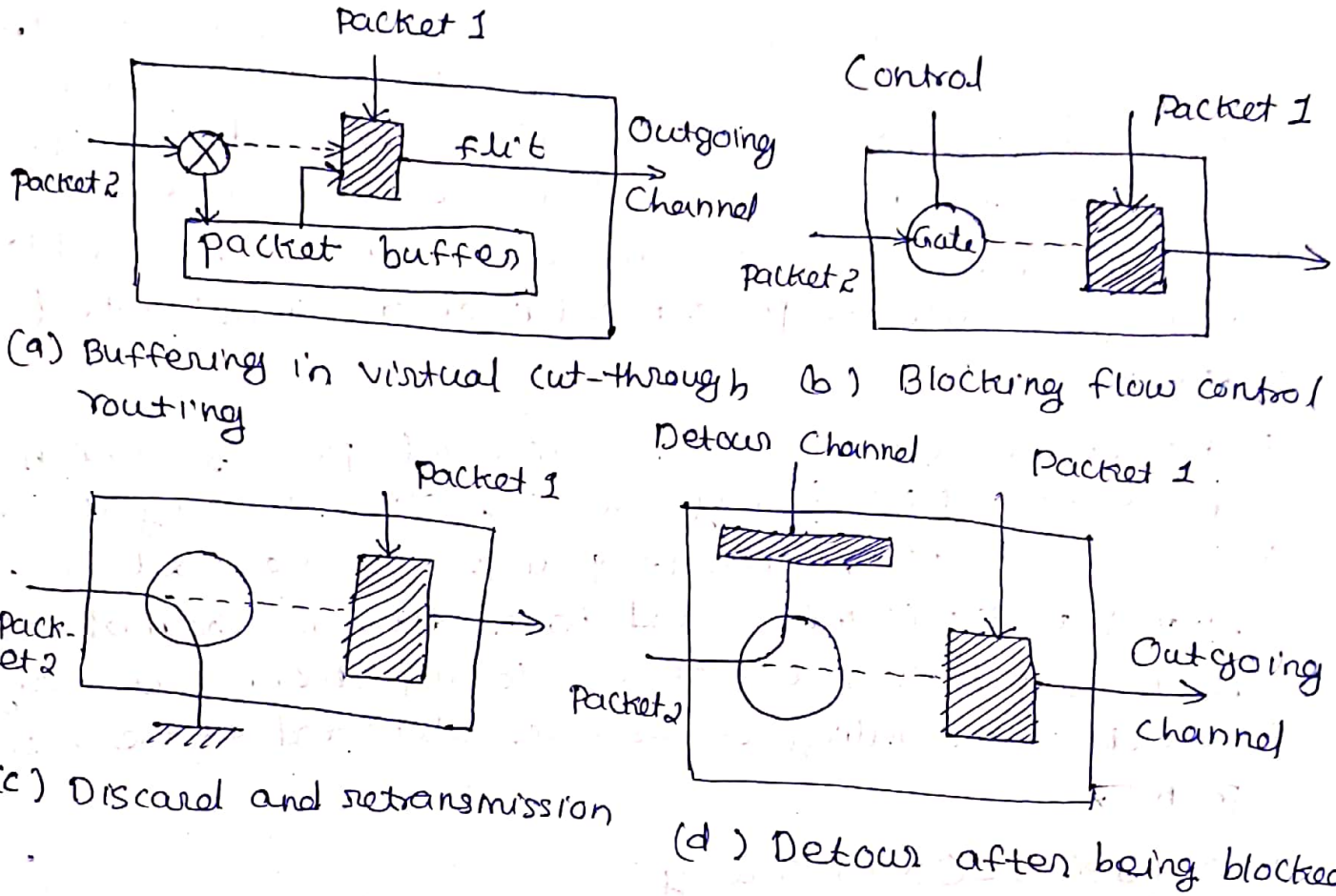


Figure 1.6 Flow control methods for resolving a collision between two packets requesting the same outgoing channel (packet 1 being allocated the channel and packet 2 being denied).

1. The virtual cut-through method offers a compromise by combining the store-and-forward and wormhole routing schemes.
   i) When collisions do not occur, the scheme should perform as wormhole routing.
   ii) In the worst case, it will behave like a store-and-forward network.
2. Figure 1.6a : packet 1 is being allocated the channel, and packet 2 being denied.
3. A buffering method has been proposed with the virtual cut-through routing scheme. Packet 2 is

temporarily stored in a packet buffer. When the channel becomes available later, it will be transmitted then. This buffering approach has the advantage of not wasting the resources already allocated.

4. Figure 4.6b: Pure wormhole routing uses a blocking policy in case of packet collision as illustrated in Figure 4.6b.
5. Figure 4.6c: Shows the discard policy, which simply drops the packet being blocked from passing through.
6. Figure 4.6d: The fourth policy is called detour. The blocked packet is misrouted to a detour channel. The blocking policy is economical to implement but may result in the idling of resources allocated to the blocked packet.

2. Dimension-Order Routing

i) Types of packet routing:

1. Deterministic
2. Adaptive

Deterministic routing:

The routing path is uniquely predetermined in advance, independent of network condition.

Adaptive routing:

Depend on network conditions, and alternate paths are possible.

ii) Dimension-ordering routing requires the selection of successive channels to follow a specific order based on the dimensions of a multidimensional network.

1. In the case of a two-dimensional mesh network, the scheme is called X-Y routing because a routing path along the

X-dimension is decided first before choosing a path along the Y-dimension.

2. For hypercube (or n-cube) networks, the scheme is called E-cube routing.

3. E-cube Routing on Hypercube

n-cube with $N = 2^n$ nodes.

$b = b_{n-1} b_{n-2} \cdots b_2 b_0$, where b is node

b is binary coded.

∴ Source node $s = s_{n-1} \cdots s_1 s_0$ and

destination node $d = d_{n-1} \cdots d_1 d_0$.

To determine a route from s to d with a minimum number of steps.

we denote the n dimensions as $i = 1, 2, \ldots, n$

where the ith dimension corresponds to the $(i-1)$st bit in the node address.

Let $v = v_{n-1} \cdots v_1 v_0$ be any node along the route.

The route is uniquely determined as follows:

1. Compute the direction bit $r_i = s_{i-1} \oplus d_{n-1}$ for all n dimensions $(i = 1, \ldots, n)$.

Start the following with dimension $i = 1$ and $v = s$.

2. Route from the current node v to the next node $v \oplus 2^{i-1}$ if $r_i = 1$. Skip this step if $r_i = 0$.

3. Move to dimension $i+1$ (i.e., $i \leftarrow i+1$).

if $i \leq n$, goto step 2, else quit.

Example

E-cube routing on a four-dimensional hypercube.

The above E-cube routing algorithm is illustra-

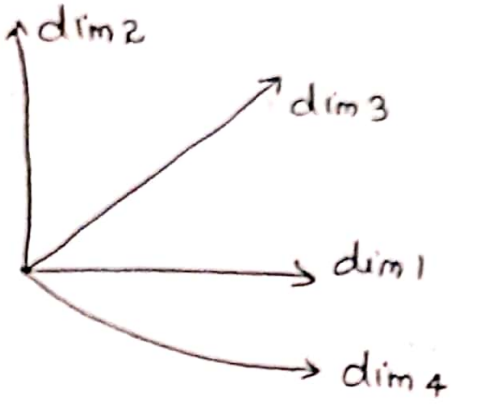
with the example in Figure 4.7

Now $n=4$, $S=0110$, and $d=1101$. Thus $r = r_4 r_3 r_2 r_1 = 1011$

Route from S to $S \oplus 2^0 = 0111$ since $r_1 = 0 \oplus 1 = 1$.

Route from $V=0111$ to $V \oplus 2^1 = 0101$ since $r_2 = 1 \oplus 0 = 1$.

Skip dimension $i=3$ because $r_3 = 1 \oplus 1 = 0$. Route from $V = 0101$ to $V \oplus 2^3 = 1101 = d$ since $r_4 = 1$.



source : S = 0110
Destination : d = 1101
Route :
0110 → 0111 → 0101 → 1101

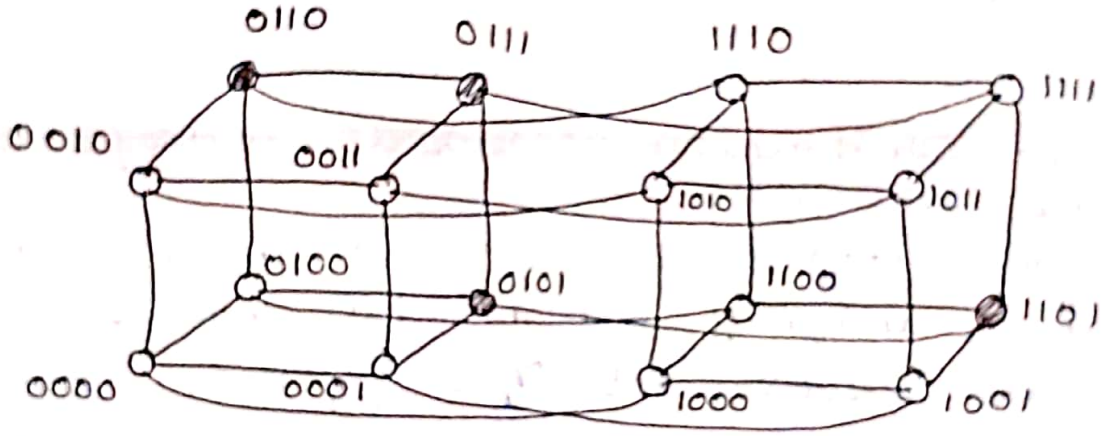


Figure 4.7 E-cube routing on a hypercube computer with 16 nodes.

The route selected is shown in Figure 4.7 by arrows. Note that the order route is determined from dimension 1 to dimension 4 in order. If the i-th bit of S and d agree, no routing is needed along dimension i. Otherwise, move from the current node to the other node along the same dimension. The procedure is repeated until the destination is reached.

4. X-Y Routing on a 2D Mesh

X-Y routing is illustrated by the example in Figure 4.8.

From any source node $s = (x_1, y_1)$ to any destination node $d = (x_2, y_2)$, route from s along the X-axis first until it reaches the column y_2, where d is located. Then route to d along the Y-axis.

There are four possible X-Y routing patterns corresponding to the east-north, east-south, west-north, and west-south paths chosen.

Example: X-Y routing on a 2D mesh-connected multicomputer.

Four (source, destination) pairs are shown in Figure 4.8 to illustrate the four possible routing patterns on a two-dimensional mesh.

An east-north route is needed from node (2,1) to node (7,6). An east-south route is set up from node (0,7) to node (4,2). A west-south route is needed from node (5,4) to (2,0). The fourth route is west-north bound from node (6,3) to node (1,5).

If the X-dimension is always routed first and then Y-dimension, a deadlock or circular wait situation will not exist.

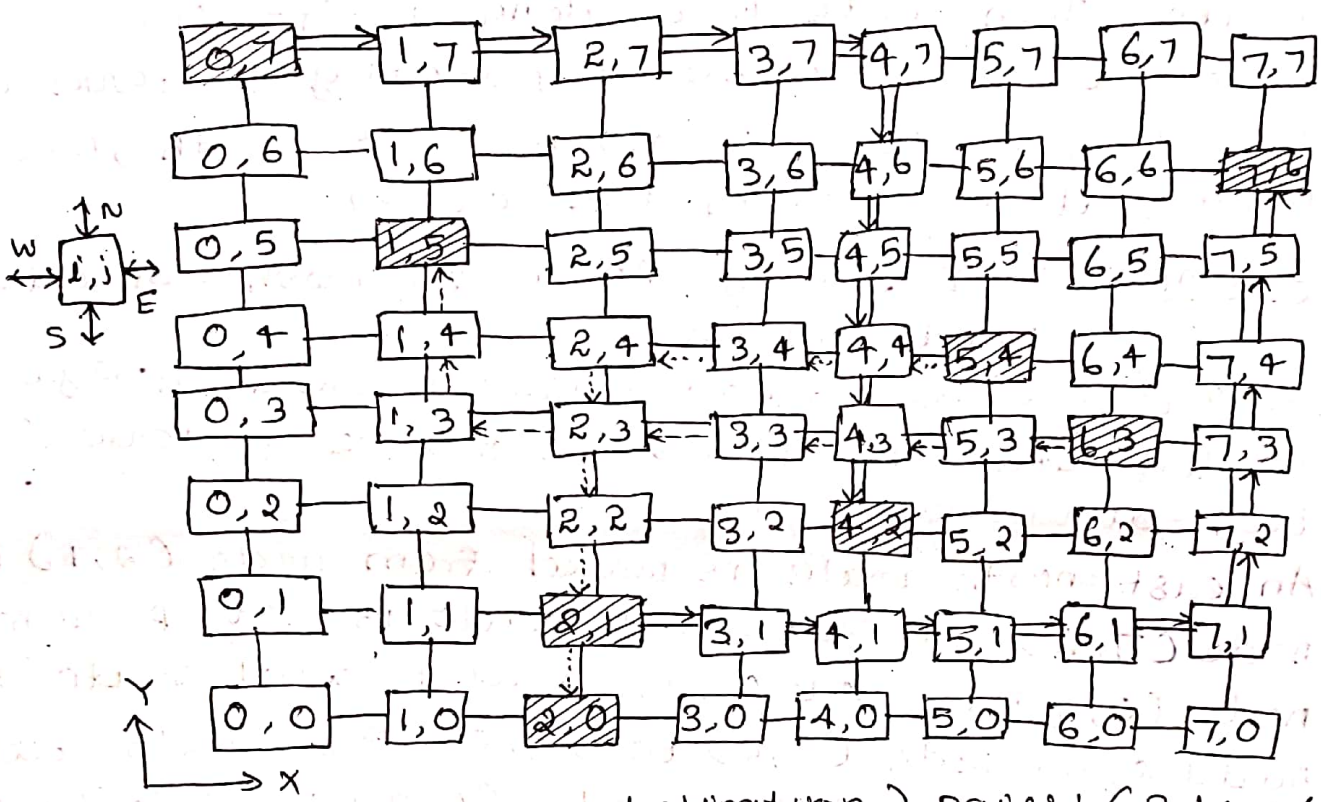
5. Adaptive Routing

i. The main purpose of using adaptive routing is to avoid deadlock.

ii. The concept of virtual channels adaptive routing is more economical and feasible to implement.

iii. Example : Adaptive X – Y routing using virtual channels.

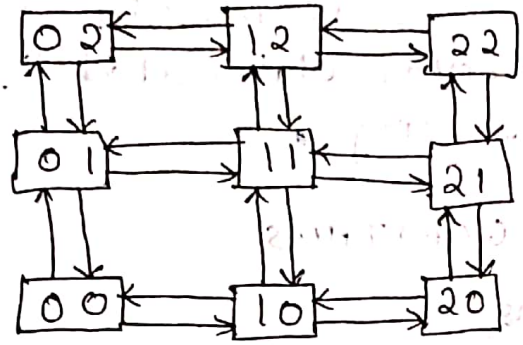
Figure 4.9c can be used to avoid deadlocks because all eastbound X-channels are not in use. Similarly, the virtual network in Figure 4.9d supports only eastbound



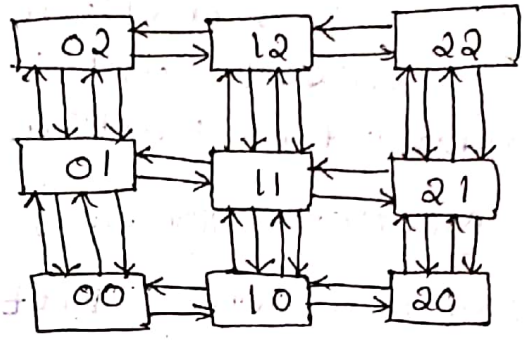
Four (source ; destination) pairs : (2,1 ; 7,6) → (0,7 ; 4,2) → (5,4 ; 2,0) → (6,3 ; 1,5)

Figure 4.8 X–Y routing on a 2D mesh computer with 8×8 = 64 nodes.

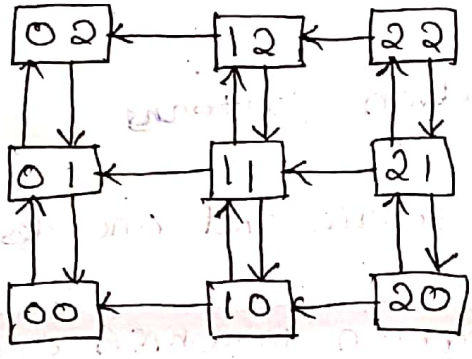
traffic using a different set of virtual Y-channels. The two virtual networks are used at different times; thus deadlocks can be adaptively avoided.



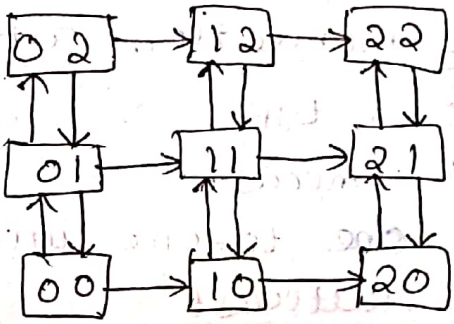
(a) Original mesh without virtual channel



(b) Two pairs of virtual channels in Y-dimension.



(c) For a westbound message



d) For an eastbound message.

Figure 4.9 Adaptive X-Y routing using virtual channels to avoid deadlocks; only westbound and eastbound traffic are deadlock-free.

This simple example uses two pairs of virtual channels in the Y-dimension of a mesh using X-Y routing. For westbound traffic, the virtual networks in Fig. 4.9c can be used to avoid deadlocks because all eastbound X-channels are not in use. Similarly, the virtual network in Fig. 4.9d supports only eastbound traffic using a different set of virtual Y-channels.

The two virtual networks are used at different times; thus deadlock can be adaptively avoided. This concept will be further elaborated for achieving deadlock-free multicast routing.

MULTICAST ROUTING ALGORITHMS

1. Communication Patterns
2. Routing Efficiency
3. Virtual Networks
4. Network Partitioning.

1. Communication patterns

Four types of communication patterns.

1. Unicast
   One-to-one with one source and one destination.
2. Multicast
   one-to-many communication in which one source sends the same message to multiple destinations.
3. Broadcast
   One-to all communication.
4. Conference
   Many-to-many communication.

All patterns can be implemented with multiple unicasts sequentially, or even simultaneously if resource conflicts can be avoided.

2. Routing Efficiency

Two efficiency parameters are
1. Channel traffic and
2. Communication latency.

Channel traffic:

The channel traffic at any time instant (or during any time period) is indicated by the number of channels used to deliver the messages involved.

Latency

The latency is indicated by the longest packet transmission time involved.

1. An optimally routed network should achieve both minimum traffic and minimum latency for the communication patterns involved.
2. Achieving minimum traffic may not necessarily achieve minimum latency at the same time, and vice versa.
3. Latency is the more important issue in a store-and-forward network.
4. Traffic demand affects efficiency more in a wormhole routed network.

Example

Multicast and broadcast on a mesh-connected computer.

Multicast routing is implemented on a 3×3 mesh in Figure 4.10. The source node is identified as S, which transmits a packet to five destinations labeled D_i for $i = 1, 2, \ldots, 5$.

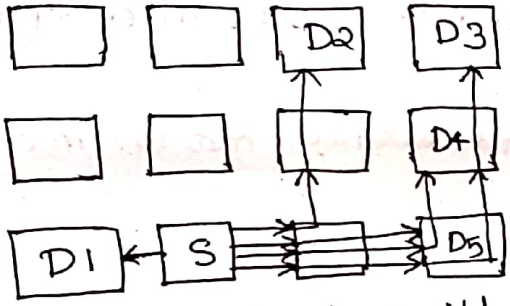
This five-destination multicast can be implemented by five unicasts, as shown in Figure 4.10a. The X-Y routing traffic requires the use of $1+3+4+3+2 = 13$ channels, and the latency is 4 for the longest path leading to D3.

A multicast can be implemented by replicating the packet at an intermediate node, and multiple copies of the packet reach their destinations with significantly reduced channel traffic.

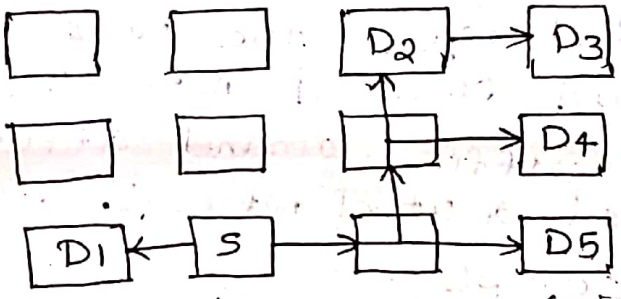
Two multicast routes are given in Figures 4.10b and 4.10c, resulting in traffic of 7 and 6, respectively.

1. On a wormhole-routed network, the multicast route in Figure 4.10c is better.
2. For a store-and-forward network, the route in Figure 4.10b is better and has a shorter latency.

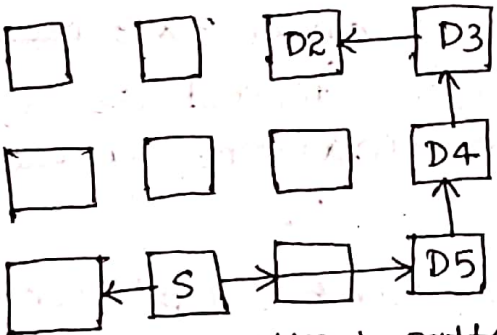
A four-level spanning tree is used from node S to broadcast a packet to all the mesh nodes in Figure 4.10d.



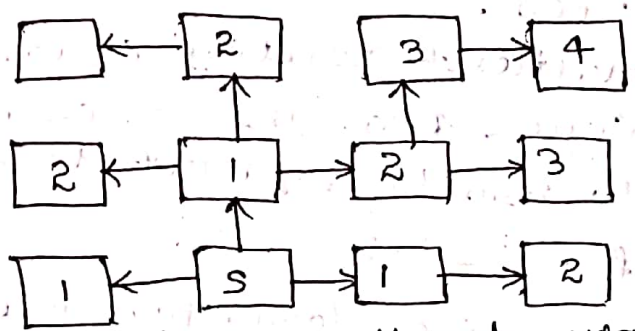
a) Five unicasts with traffic = 13 and distance=4



(b) A multicast pattern with traffic = 7 and distance = 4



c) Another multicast pattern with traffic = 6 and distance = 5



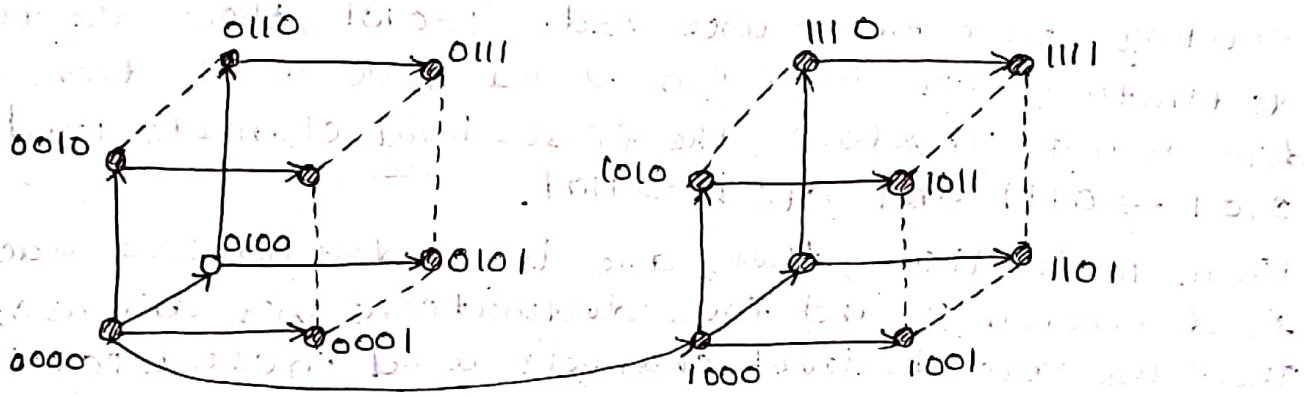
d) Broadcast to all nodes via a tree (numbers in nodes correspond to levels of the tree)

Figure 4.10 Multiple unicasts, multicast patterns, and a broadcast tree on a 3 × 4 mesh computer.

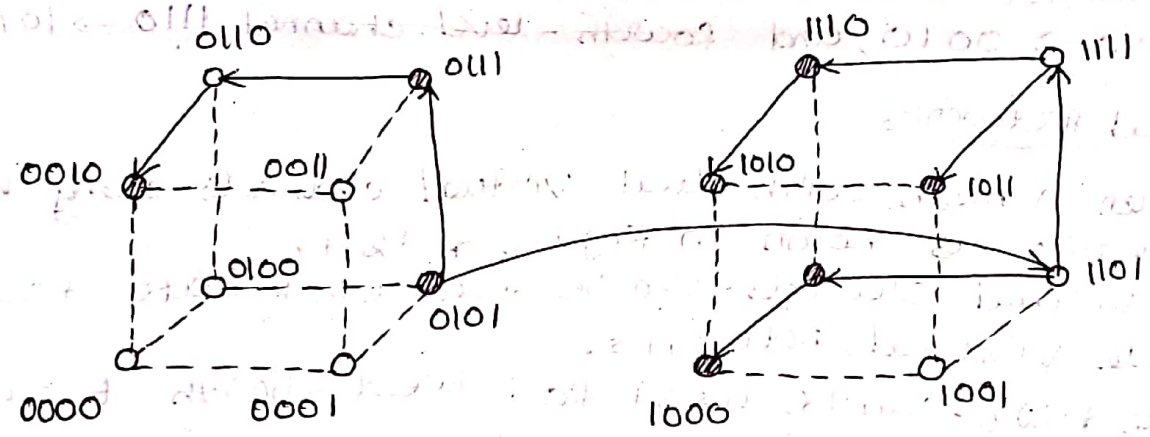
3. Nodes reached at level 'i' of the tree have latency 'i'. This broadcast tree should result in minimum latency as well as in minimum traffic.

Example

Multicast and broadcast on a hypercube computer.



(a) Broadcast tree for a 4-cube rooted at node 0000



(b) A multicast tree from node 0101 to seven destination nodes 1100, 0111, 1010, 1110, 1011, 1000, and 0010

Figure 4.11 Broadcast tree and multicast tree on a 4-cube using a greedy algorithm.

To broadcast on an n-cube, a similar spanning tree is used to reach all nodes within a latency of n. This is illustrated in Figure 4.11 a for a 4-cube rooted at node 0000. Again, minimum traffic should result

with a broadcast tree for a hypercube.

A greedy multicast tree is shown in Figure 4.11 b- for sending a packet from node 0101 to seven destination nodes. The greedy multicast algorithm is based on sending the packet through the dimension (s) which can reach the most number of remaining destinations.

Starting from the source node S=0101, there are two destinations via dimension 2 and five destinations via dimension 4. Therefore, the first-level channels used are 0101 → 0111 and 0101 → 1101.

From node 1101, there are three destinations reachable in dimension 2 and four destinations via dimension 1. Thus the second-level channels used include 1101 → 1111, 1101 → 1100, and 0111 → 0110.

Similarly, the remaining destinations can be reached with third-level channels 1111 → 1110, 1111 → 1011, 1100 → 1000 and 0110 → 0010, and fourth-level channel 1110 → 1010.

3. Virtual Networks.

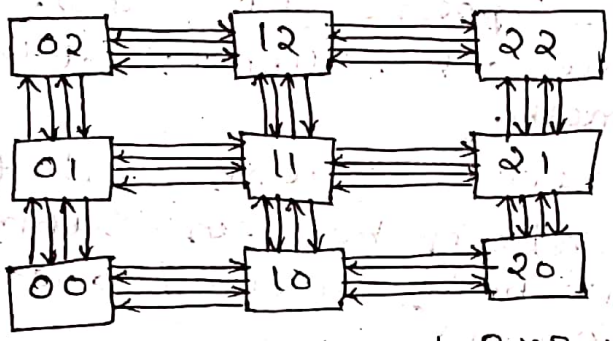
1. Consider a mesh with dual virtual channels along both dimensions as shown in Figure 4.12a.
2. These virtual channels can be used to generate four possible virtual networks.
3. Figure 4.12b can be used for west-north traffic virtual network.

4. No cycle is possible on any of the virtual networks.
5. Deadlock can be completely avoided when X-Y routing is implemented on these networks.
6. If both pairs between adjacent nodes are physical channels, then any two of the four virtual networks can be simultaneously used without conflict.

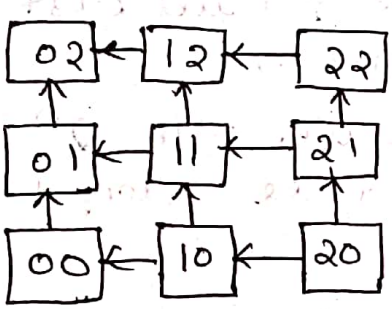
7. If only one pair of physical channels is shared by the dual virtual channels between adjacent nodes, then only (b) and (e) or (c) and (d) can be used simultaneously.

8. Other combinations, such as (b) and (c), or (b) and (d) or (c) and (e), or (d) and (e), cannot coexist at the same time due to a shortage of channels.

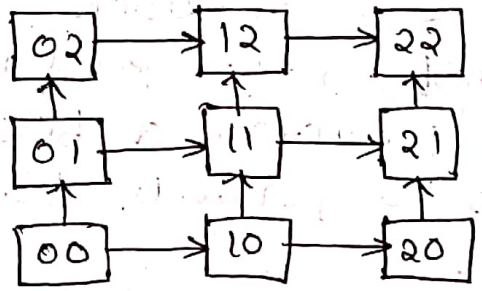
9. Adding channels to the network will increase the adaptivity in making routing decisions.



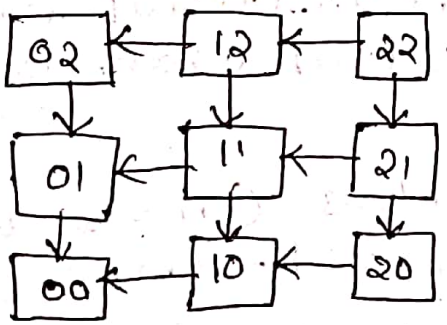
(a) A dual-channel 3×3 mesh



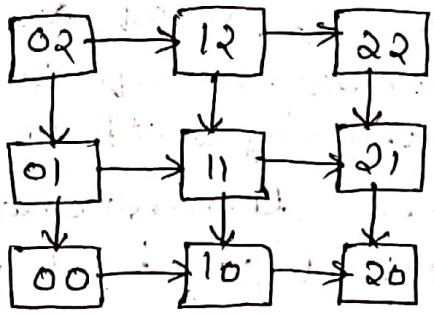
(b) West-north subnet



(c) East-north subnet



(d) West-south subnet



(e) East-south subnet

Figure 4.12 Four virtual networks implementable from a dual-channel mesh.

4- Networks Partitioning

1- The concept of virtual networks leads to the partitioning of a given physical network into logical subnetworks for multicast communications. The idea is illustrated in Figure 4.13.
a) Suppose source node (4,2) wants to transmit to a subset of nodes in the 6 x 8 mesh.
b) The mesh is partitioned into four logical subnets.
c) All traffic heading for east and north uses the subnet at the upper right corner. Similarly, one constructs three other subnets at the remaining corners of the mesh.
d) Nodes in the fifth column and third row are along the boundary between subnets.
e) The traffic is being directed outward from the center node (4,2).
f) There is no deadlock if an x-y multicast is performed in the partitioned mesh.

2. One can partition a binary n-cube into 2^{n-1} subcubes to provide deadlock-free adaptive routing.
3. Each subcube has n+1 levels with 2^n virtual channels per level for the bidirectional network.
4. The number of required virtual channels increases rapidly with n.
5. For low-dimensional cubes (n = 2 to 4) this method is best for general-purpose routing.

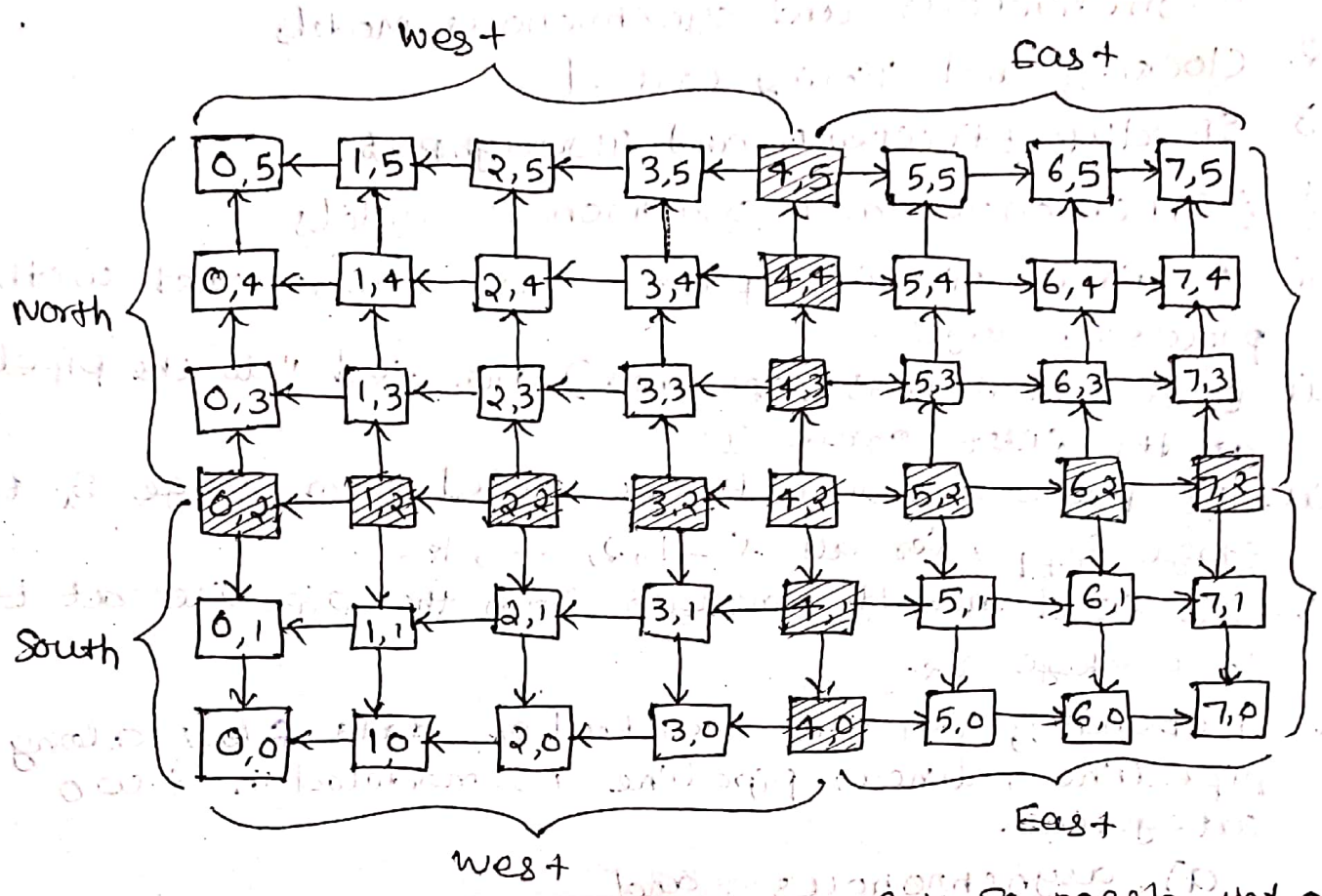


Figure 4.13 partitioning of a 8 x 8 mesh into four subnets for a multicast from source node (4,2). Shaded nodes are along the boundary of adjacent subnets.

PIPELINING AND SUPERSCALAR TECHNIQUES

1. Linear Pipeline Processors
2. Nonlinear Pipeline Processors.

1. Linear Pipeline processors

i. 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.

ii. Linear pipelines are applied for
   a) instruction execution,
   b) arithmetic computation, and
   b) memory-access operations.

1. Asynchronous and synchronous Models
2. Clocking and Timing control
3. Speedup Efficiency, and Throughput.

1. Asynchronous and synchronous models

i. A linear pipeline processor is constructed with k processing stages.
ii. External inputs (operands) are fed into the pipeline at the first stage S_1.
iii. The processed results are passed from stage S_i to stage S_{i+1}, for all $i = 1, 2, \ldots, k-1$.
iv. The final result emerges from the pipeline at the last stage S_k.
v. Depending on the control of data flow along the pipeline, linear pipeline is modeled in two categories.
   a) asynchronous and
   b) synchronous.

Asynchronous Model

1. As shown in Figure 1.14a, data flow between adjacent stages in an asynchronous pipeline is controlled by a handshaking protocol.
2. When stage S_i is ready to transmit, it sends a ready signal to stage S_{i+1}.
3. After stage S_{i+1} receives the incoming data, it returns an acknowledge signal to S_i.
4. Asynchronous pipelines are useful in designing communication channels in message-passing multicomputers where pipelined wormhole routing is practiced.

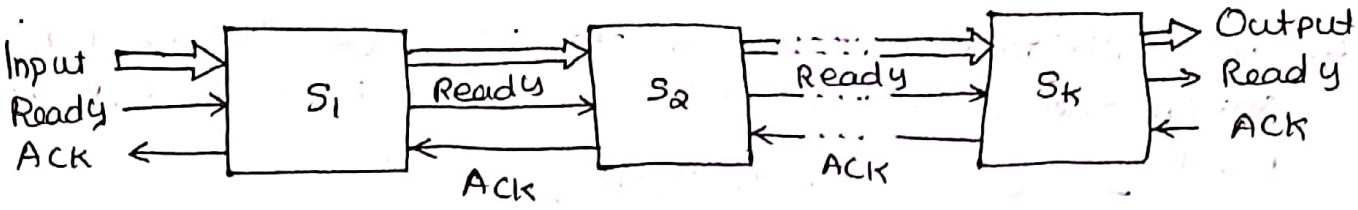


Fig. 4.14 (a) An asynchronous pipeline model.

5. Asynchronous pipelines may have a variable throughput rate.
6. Different amounts of delay may be experienced in different stages.

Synchronous Model

Synchronous pipelines are illustrated in Figure 4.15

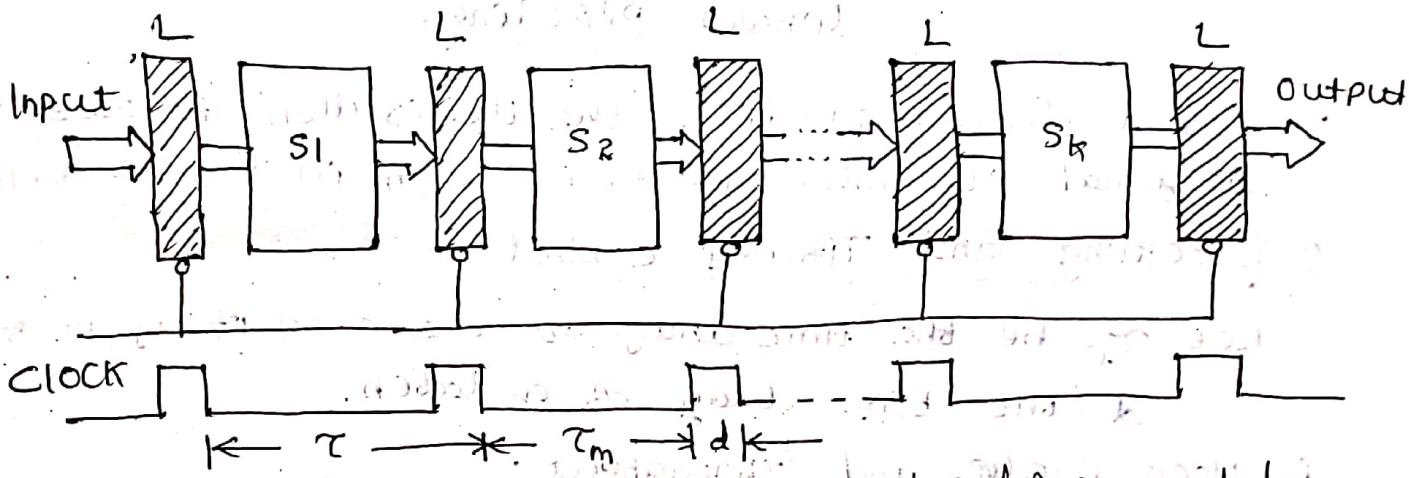
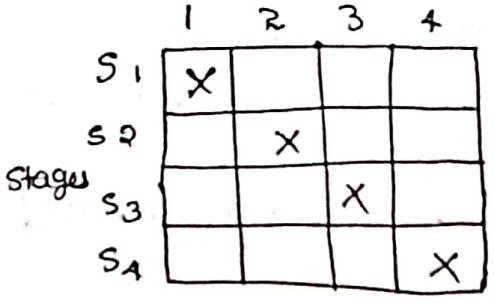


Figure 4.15 A synchronous pipeline model

1. Clocked latches are used to interface between stages.
2. The latches are made with master-slave flip-flops, which can isolate inputs from outputs.
3. Upon the arrival of a clock pulse, all latches transfer data to the next stage simultaneously.
4. The pipeline stages are combinational logic circuits.
5. It is desired to have approximately equal delays in all stages. These delays determine the clock period and thus the speed of the pipeline.

6. The utilization pattern of successive stages in a synchronous pipeline is specified by a reservation table.



S_i = Stage i
L = Latch
τ = Clock period
T_m = Maximum stage delay
d = Latch delay
Ack = Acknowledge signal.

Figure 4.16 Reservation table of a four-stage linear pipeline.

7. For a linear pipeline, the utilization follows the diagonal streamline pattern shown in Figure 4.16.

2. Clocking and Timing Control

Let τ_i be the time delay of the circuitry in stage S_i.
d the time delay of a latch.

1. Clock Cycle and Throughput:

T_m is the maximum stage delay.

$$\tau = \max_i \{\tau_i\}_1^k + d = T_m + d$$

The clock pulse has a width equal to d.
The pipeline frequency is defined as the inverse of the clock period:

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

f represents the maximum throughput of the pipeline. Depending on the initiation rate of successive tasks entering the pipeline, the actual throughput of the pipeline may be lower than f.

Clock Skewing

1. Ideally, we expect the clock pulses to arrive at all stages (latches) at the same time.
2. Due to a problem known as clock skewing, the same clock pulse may arrive at different stages with a time offset of s.

t_{max} – the time delay of the longest logic path within a stage and
t_{min} – the time delay of the shortest path within a stage.

3. To avoid a race in two successive stages

$$\tau_m \geq t_{max} + s \quad \text{and}$$
$$d \leq t_{min} - s$$

$$\therefore d + t_{max} + s \leq \tau \leq \tau_m + t_{min} - s$$

In the ideal case $s = 0$, $t_{max} = \tau_m$, and $t_{min} = d$.

$$\therefore \tau = \tau_m + d.$$

Speedup, Efficiency, and Throughput

A linear pipeline of k stages can process n tasks in $k + (n-1)$ clock cycles.

i. k cycles are needed to complete the execution of the very first task and
ii. the remaining $(n-1)$ tasks require $(n-1)$ cycles.

$\therefore$ Total time required $T_k = [k + (n-1)]\tau$
where τ is the clock period.

The amount of time a nonpipelined processor takes to execute n tasks is $T_1 = nk\tau$.

Speedup Factor

The speedup factor of a k-stage pipeline over an equivalent non-pipelined processor is defined as

$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{k\tau + (n-1)\tau} = \frac{nk}{k + (n-1)}.$$

Optimal Number of stages

The finest level of pipelining is called micropipelining.

micropipelining :- A subdivision of pipeline stages at the logic gate level.

macropipelining :- pipeline stages conducted at the processer level.

The optimal choice of the number of pipeline stages should be able to maximize a performance / cost ratio.

A pipeline performance / cost ratio (PCR) has been defined by Larson (1973):

$$PCR = \frac{f}{C + kh} = \frac{1}{(t/k + d)(C + kh)}$$

where,

t - total time required for a nonpipelined sequential program of a given function.

$t/k+d$ - clock period of a k-stage pipeline with an equal flow-through delay t.

$C + kh$ - The total pipeline cost.

C - cost of all logic gates

h - the cost of each latch.

f - maximum throughput of the pipeline.

Figure 4.17 plots the PCR as a function of k. The peaks of the PCR curve corresponds to an optimal choice for the number of desired pipeline stages: $k_0 = \sqrt{\frac{t \cdot c}{d \cdot h}}$

where,

t - the total flow-through delay of the pipeline.
c - the total stage cost,
d - the latch delay,
h - the latch cost.

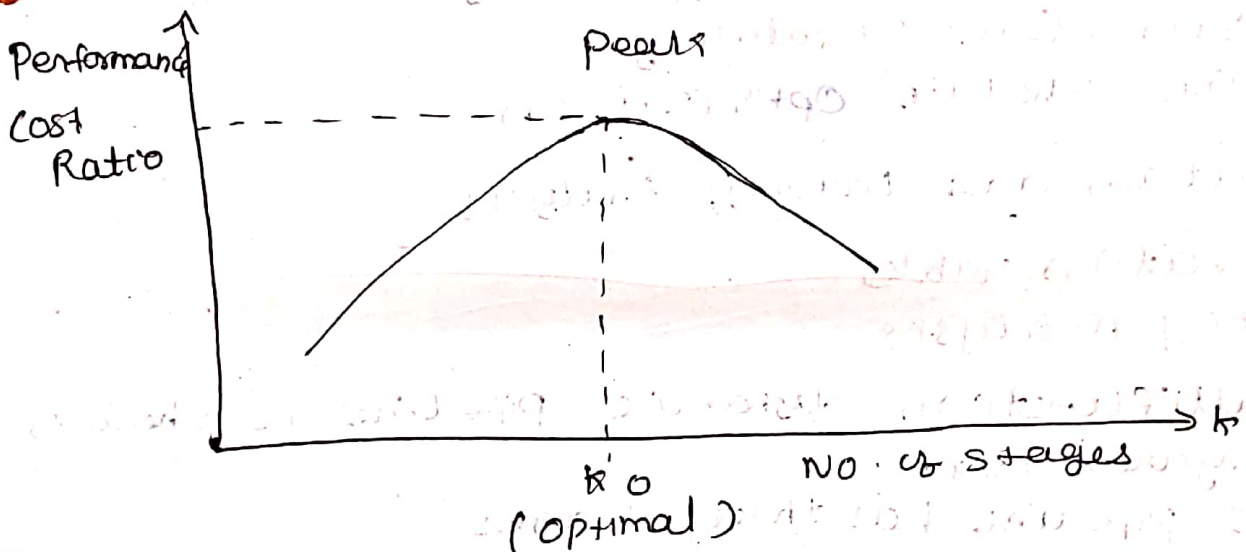


Figure 4.17 optimal number of pipeline stages.

Efficiency and Throughput

The efficiency E_k of a linear k-stage pipeline is defined as

$$E_k = \frac{S_k}{k} = \frac{n}{k + (n-1)}$$

The pipeline throughput H_k is defined as the number of tasks (operations) performed per unit time.

$$H_K = \frac{n}{[k+(n-1)]\tau} = \frac{nf}{k+(n-1)}$$

Nonlinear Pipeline Processors

1. Linear pipelines are static pipelines because they are used to perform fixed functions.
2. Nonlinear pipelines are dynamic pipelines because they are used to perform variable functions at different times.

i. Reservation and Latency Analysis
ii. Collision-Free Scheduling
iii. Pipeline Schedule Optimization

i. Reservation and Latency Analysis

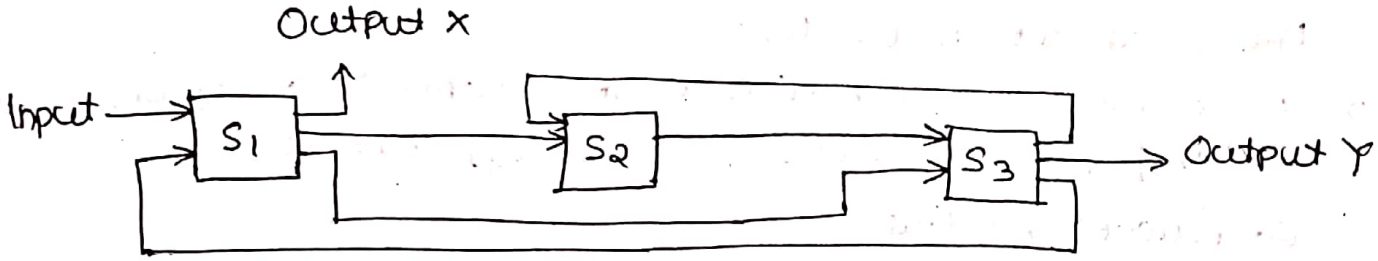
1. Reservation Tables
2. Latency Analysis

1. A multifunction dynamic pipeline is shown in Figure 4.18a.
   a) This pipeline has three stages.
   b) Besides the streamline connections from S_1 to S_2 and from S_2 to S_3, there is a feedforward connection from S_1 to S_3 and two feedback connections from S_3 to S_2 and from S_3 to S_1.
2. The output of the pipeline is not necessarily from the last stage.
3. Following different data-flow patterns, one can use the same pipeline to evaluate different functions.

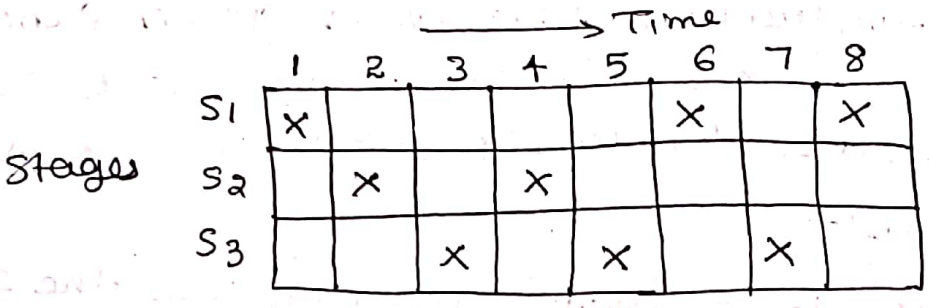
Reservation Tables

1. For a given pipeline configuration, multiple reservation tables can be generated for the evaluation

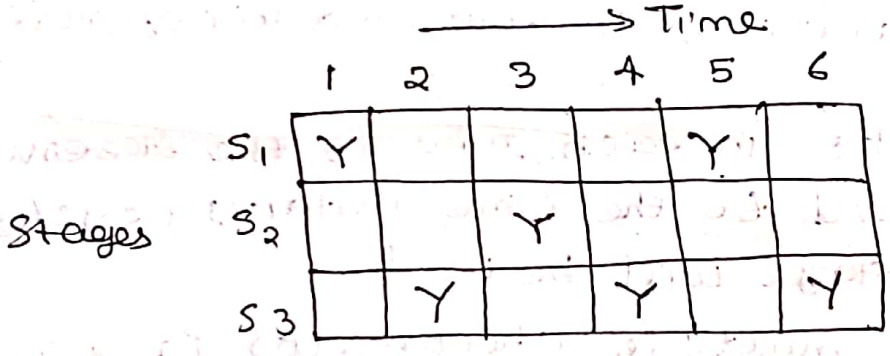
of different functions.



(a) A three-stage pipeline



(b) Reservation table of function X



(c) Reservation table for function Y.

Figure 1.18 A dynamic pipeline with feed forward and feedback connections for two different functions.

2. Two reservation tables are given in Figures 1.18b and 1.18c, corresponding to a function X and a function Y, respectively.
3. Each reservation table displays the time-space flow of data through the pipeline for one function evaluation.

4. Different functions may follow different paths on the reservation table.
5. A number of pipeline configurations may be represented by the same reservation table.

Evaluation time:

The number of columns in a reservation table is called the evaluation time of a given function. eg: The function X requires eight clock cycles to evaluate, and function Y requires six cycles.

Initiation table:

All initiations to a static pipeline use the same reservation table. A dynamic pipeline allow different initiations to follow a mix of reservation tables.

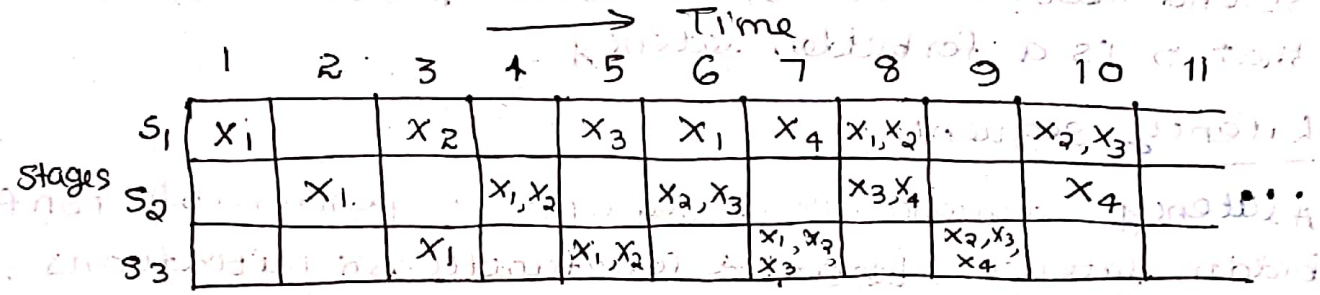
6. The checkmarks in each row of the reservation table correspond to the time instants (cycles) that a particular stage will be used.
7. There may be multiple checkmarks in a row, which means repeated usage of the same stage in different cycles.
8. Multiple checkmarks in a column mean that multiple stages are used in parallel during a particular clock cycle.

Latency Analysis

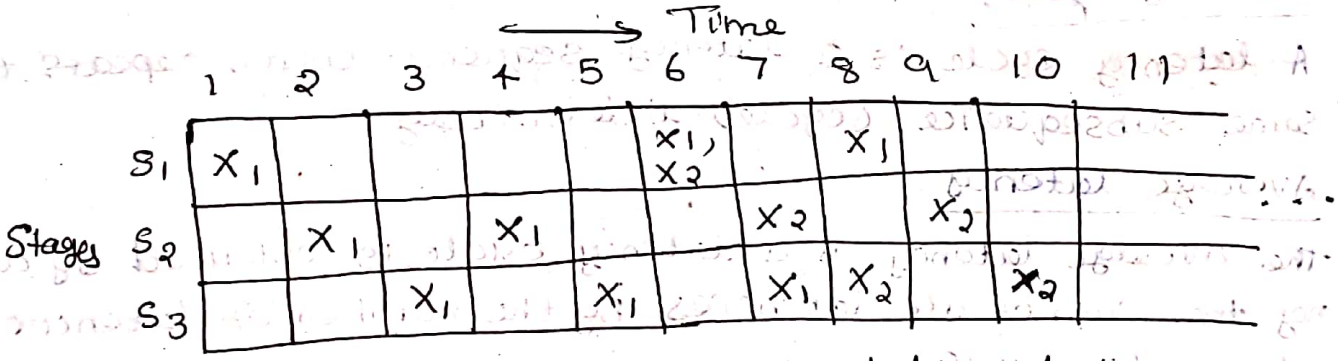
1. Latency:
   The number of time units (clock cycles) between two initiations of a pipeline is the latency between them.
2. A latency of k means that two initiations are

separated by k clock cycles.

3. A collision implies resource conflicts between two initiations in the pipeline.
4. Some latencies will cause collisions, and some will not.
5. Latencies that cause collisions are called forbidden latencies.
6. In using the pipeline in Figure 4.18 to evaluate the function X, latencies 2 and 5 are forbidden as illustrated in Figure 4.19.



(a) Collision with scheduling latency 2



(b) Collision with scheduling latency 5

Figure 4.19 Colling with forbidden latencies 2 and 5 in using the pipeline in Figure 4.18 to evaluate the function X.

7. The ith initiation is denoted as X_i in Figure 4.19.
   a) With latency 2, initiations X_1 and X_2 collide in stage 2 at time 4.
   b) At time 7, these initiations collide in stage 3.
   c) Other collisions are shown in times 5, 6, 8, ..., etc.

d) The collision patterns for latency 5 are shown in Figure 4.19b, where x_1 and x_2 are scheduled 5 clock cycles apart.

e) Their first collision occurs at time 6.

8. To detect a forbidden latency, one needs simply to check the distance between any two checkmarks in the same row of the reservation table.

eg: the distance between the first mark and the second mark in row S_1 in Figure 4.18b is 5, implying that 5 is a forbidden latency.

9. Latency sequence:

A latency sequence is a sequence of permissible non-forbidden latencies between successive task initiations.

10. Latency cycle:

A latency cycle is a latency sequence which repeats the same subsequence (cycle) indefinitely.

11. 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.

The latency cycle (1, 8) thus has an average latency of $(1+8)/2 = 4.5$.

12. Constant cycle:

A constant cycle is a latency cycle which contains only one latency value.

The average latency of a constant cycle is simply the latency itself.

Collision - Free scheduling

1. Collision vectors
2. State diagrams.
3. Greedy Cycles.

1. Collision vectors.

i. For a reservation table with n columns, the maximum forbidden latency $m \leq n-1$. The permissible latency P should be as small as possible. $1 \leq P \leq m-1$.

ii. $P = 1$: ideal case.

iii. The combined set of permissible and forbidden latencies can be easily displayed by a collision vector.

collision vector is an m-bit binary vector $C = (C_m C_{m-1} \cdots C_2 C_1)$. The value of $C_i = 1$ if latency i causes a collision and $C_i = 0$ if latency i is permissible. $C_m = 1$ -- maximum forbidden latency.

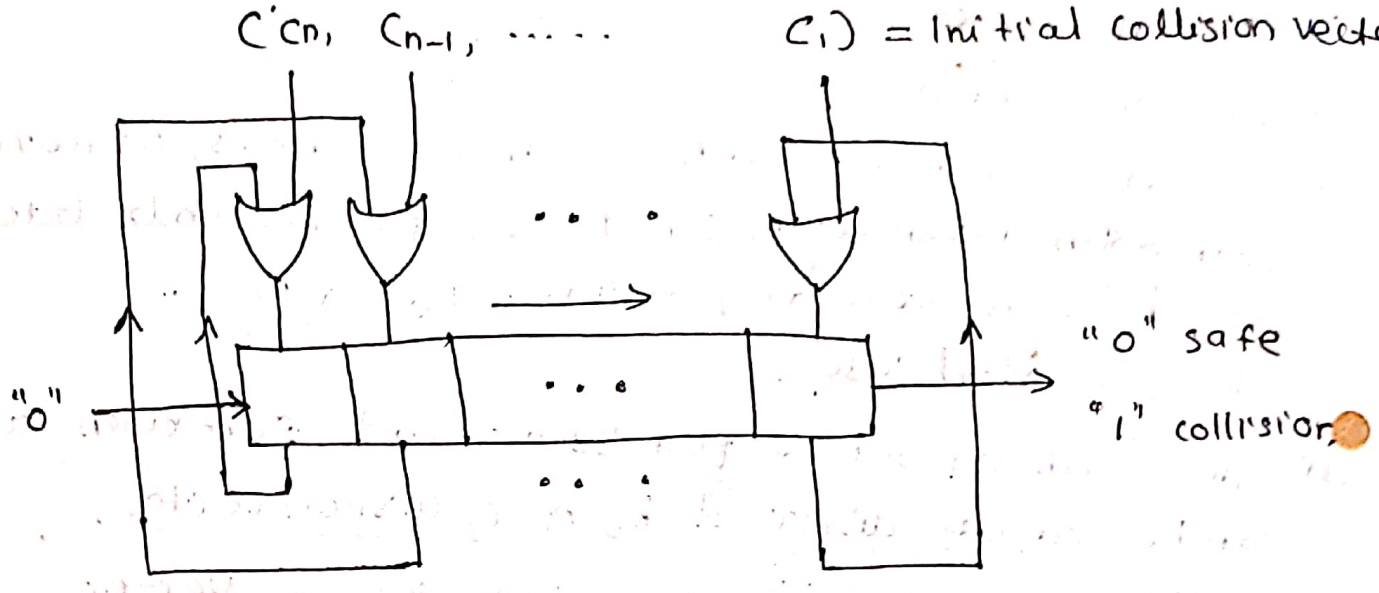
For the two reservation tables in Figure 4.18, the collision vector $C_X = (1011010)$ is obtained for function X, and $C_Y = (1010)$ for function Y. From C_X, we can immediately tell that latencies 7, 5, 4, and 2 are forbidden and latencies 6, 3, and 1 are permissible.

2. State diagrams

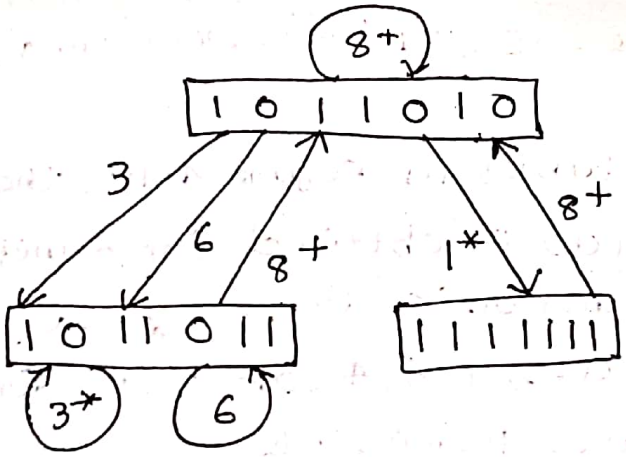
i. From the collision vector, one can construct a state diagram specifying the permissible state transitions among successive initiations.

ii. Initial collision vector is the initial state of the pipeline at time 1.

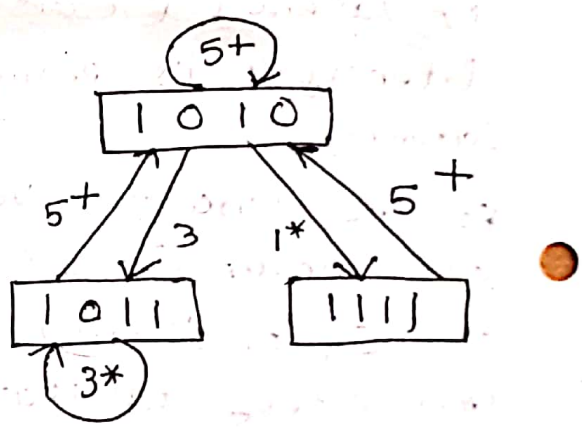
iii. The next state of the pipeline at time t+P is obtained with the assistance of an m-bit right shift register as in Figure 4.20a.



(a) State transition using an n-bit right shift register, where n is the maximum forbidden latency.



(b) State diagram for function X



(c) State diagram for function Y.

Figure 4.20 Two state diagrams obtained from the two reservation tables in Figure 4.18, respectively.

1. The initial collision vector C is initially loaded into the register.
2. The register is then shifted to the right.

3. Each 1-bit shift corresponds to an increase in the latency by 1.
4. When a 0 bit emerges from the right end after p shifts, it means p is a permissible latency.
5. A 1 bit being shifted out means a collision, and thus the corresponding latency should be forbidden.
6. Logical 0 enters from the left end of the shift register. The next state after p shifts is obtained by bitwise ORing the initial collision vector with the shifted register contents.
7. A state diagram is obtained in Figure 4.20b for function X.
   i. From the initial state (1011010), only three outgoing transitions are possible, corresponding to the three permissible latencies 6, 3, and 1 in the initial collision vector.
   ii. From state (1011011), one reaches the same state after either three shifts or six shifts.
8. When the number of shifts is m+1 or greater, all transitions are redirected back to the initial state.
9. In Figure 4.20c, a state diagram is obtained for the reservation table in Figure 4.18c using a 4-bit shift register.
10. Different reservation tables may produce different collision vectors and thus different state diagrams.
11. The state diagram covers all permissible state transitions that avoid collisions.

3. Greedy Cycles

1. From the state diagram, we can determine optimal latency cycles which result in MAL (Minimal average latency).
2. There are infinitely many latency cycles one can trace from the state diagram.
3. Simple cycle

A simple cycle is a latency cycle in which each state appears only once.
In Figure 4.20b, only (3), (6), (8), (1,8), (3,8), and (6,8) are simple cycles.

4. Greedy Cycles

A greedy cycle is one whose edges are all made with minimum latencies from their respective starting states.
Eg: In Figure 4.20b the cycles (1,8) and (3) are greedy cycles.

5. The minimum-latency edges in the state diagrams are marked with asterisks.
6. At least one of the greedy cycles will lead to the MAL.

3. Pipeline Schedule Optimization.

1. Bounds on the MAL
2. Delay Insertion.
3. Pipeline Throughput
4. Pipeline Efficiency.

The idea is to insert noncompute delay stages into the original pipeline. This will modify the reservation table, resulting in a new collision vector and an improved state diagram. The purpose is to yield an optimal latency

cycle, which is absolutely the shortest.

Bounds on the MAL

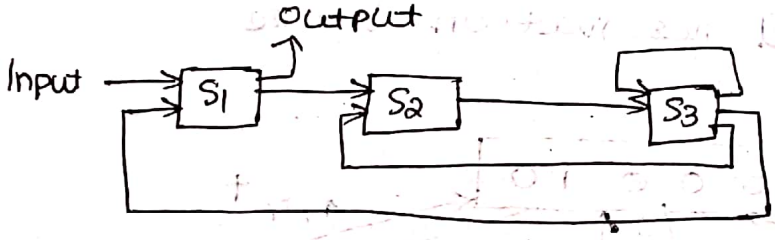
(1) The MAL is lower-bounded by the maximum number of check marks in any row of the reservation table.

(2) The MAL is lower than or equal to the average latency of any greedy cycle in the state diagram.

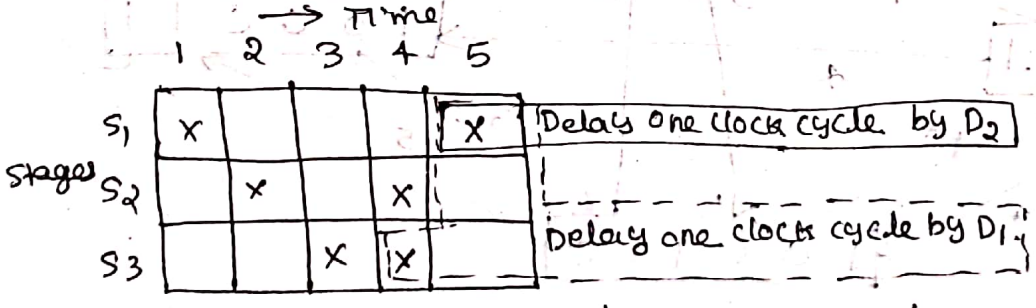
(3) The average latency of any greedy cycle is upper-bounded by the number of 1's in the initial collision vector plus 1. This is also an upper bound on the MAL.

Delay Insertion

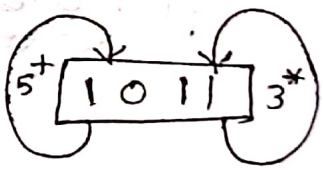
1. The purpose of delay insertion is to modify the reservation table, yielding a new collision vector. This leads to a modified state diagram, which may produce greedy cycles meeting the lower bound on the MAL.



(a) A three-stage pipeline

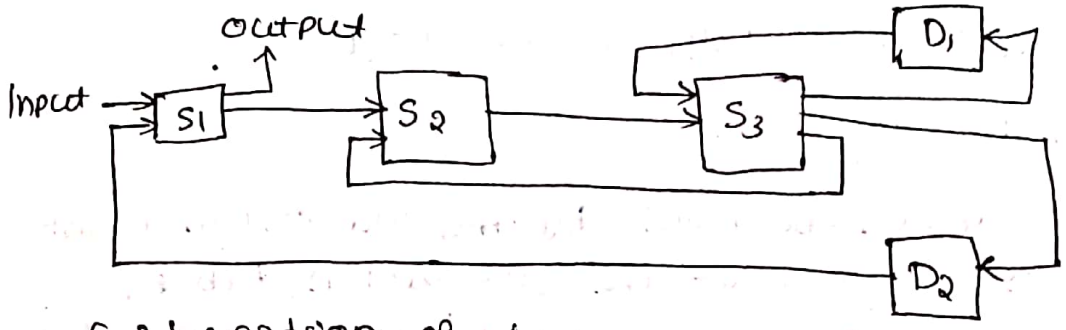


(b) Reservation table and operations being delayed

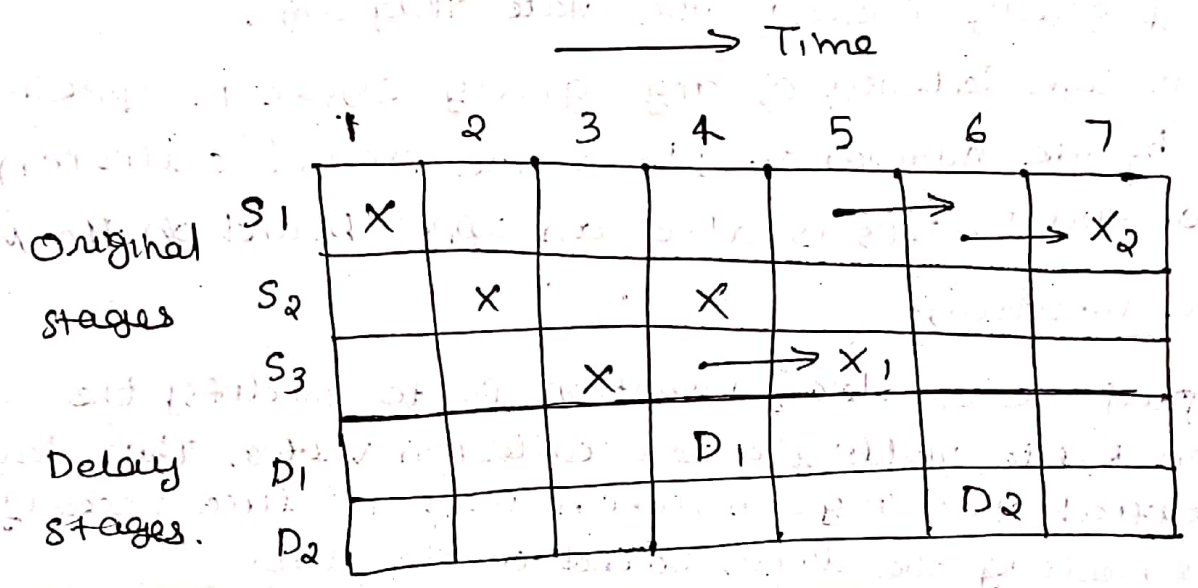


(c) New state transition diagram with MAL = 3.

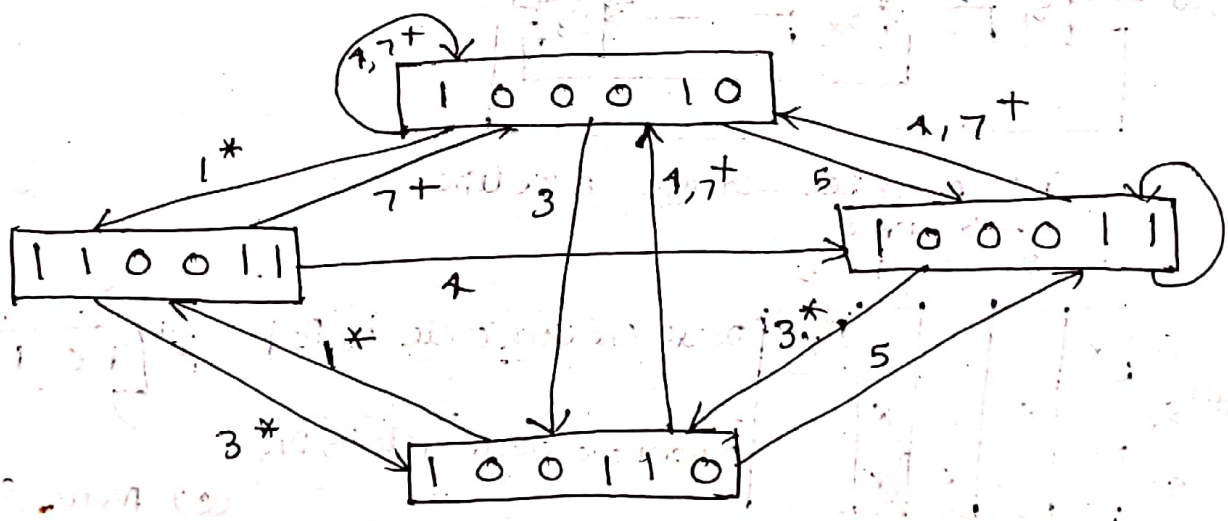
Figure 4.21 A pipeline with a minimum average latency of 3.



(a) Insertion of two nonompute delay stages.



(b) Modified reservation table



(c) Modified state diagram with a reduced

MAL = (1+3)/2 = 2.

Figure 4-22 Insertion of two delay stages to obtain an optimal MAL for the pipeline in Figure 4-21.

Module V

INSTRUCTION PIPELINE DESIGN

A stream of instructions can be executed by a pipeline in an overlapped manner.

1. Instruction Execution Phases
2. mechanisms for Instruction Pipelining.
3. Dynamic Instruction scheduling.
4. Branch Handling Techniques.

1. Instruction Execution Phases.

a) Instruction execution consists of
   i. Instruction fetch,
   ii. decode,
   iii. operand fetch,
   iv. execute, and
   v. write-back.

These phases are ideal for overlapped execution on a linear pipeline.

a) Pipelined instruction processing

b) A typical instruction pipeline is depicted in Figure 5.1.

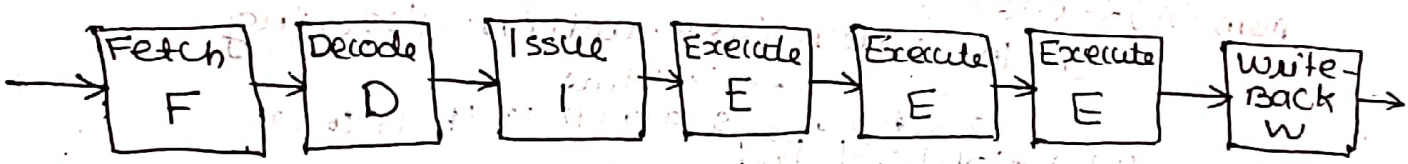


Figure 5.1 A seven-stage instruction pipeline.

1. Fetch stage (F)
   Fetches instructions from a cache memory one per cycle.
2. Decode stage (D)
   Reveals the instruction function to be performed and identifies the resources needed.

3. Issue stage (I)

i Reserve resources.

ii pipeline controlled interlocks are maintained.

iii The operands are read from registers during the issue stage.

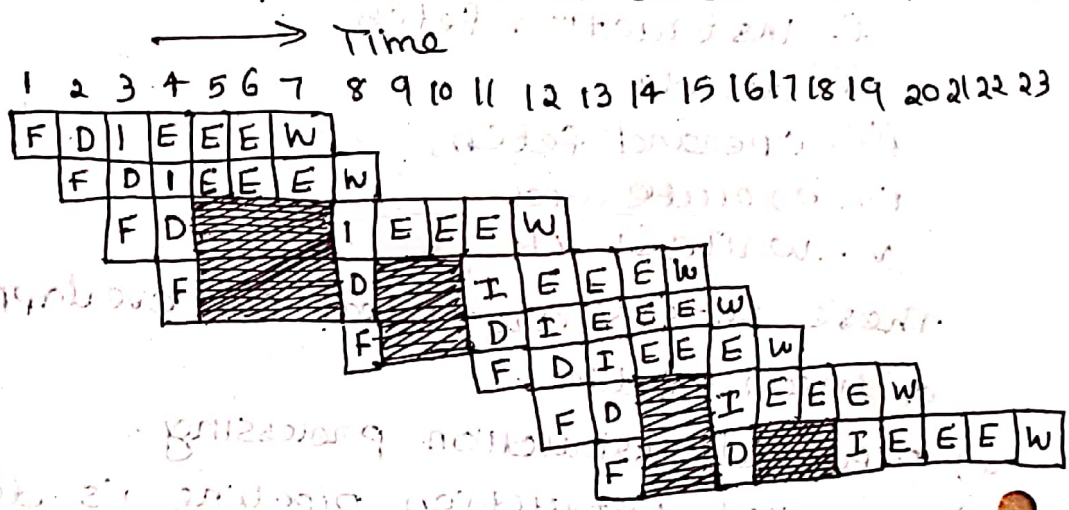
4. Execute Stages (E).

The instructions are executed in one or several execute stages.

5. write back stage (W)

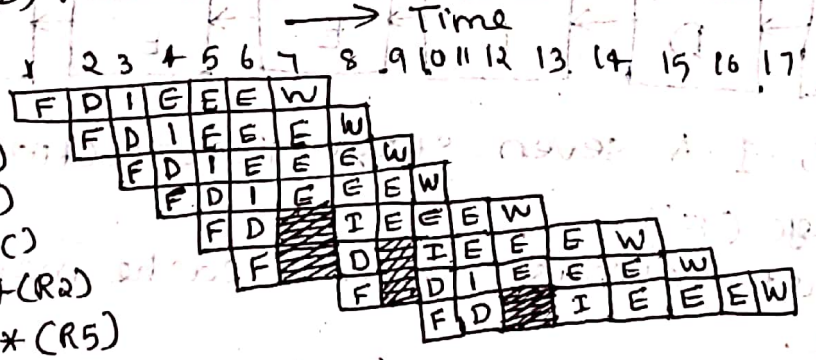
Used to write results into the registers.

Memory load or store operations are treated as part of execution.



$R_1 \leftarrow Mem(Y)$
$R_2 \leftarrow Mem(Z)$
$R_3 \leftarrow (R_1) + (R_2)$
$Mem(X) \leftarrow (R_3)$
$R_4 \leftarrow Mem(B)$
$R_5 \leftarrow Mem(C)$
$R_6 \leftarrow (R_4) * (R_5)$
$Mem(A) \leftarrow (R_6)$

(b) In-order instruction issuing



$R_1 \leftarrow Mem(Y)$
$R_2 \leftarrow Mem(Z)$
$R_4 \leftarrow Mem(B)$
$R_5 \leftarrow Mem(C)$
$R_3 \leftarrow (R_1) + (R_2)$
$R_6 \leftarrow (R_4) * (R_5)$
$Mem(X) \leftarrow (R_3)$
$Mem(A) \leftarrow (R_6)$

(c) Reordered instruction issuing

Figure 5:2 Pipelined execution of $X = Y + Z$ and $A = B \times C$.

1. Figure 5.2b illustrates the issue of instructions following the original program order.
2. The shaded boxes correspond to idle cycles when instruction issues are blocked due to resource latency or due to conflicts or due to data dependences.
3. The first two load instructions issue on consecutive cycles.
4. The add is dependent on both loads and must wait three cycles before the data (Y and Z) are loaded in.
5. Figure 5.2c shows an improved timing after the instruction issuing order is change to eliminate unnecessary delays due to dependence.
   i. The idea is to issue all four load operations in the beginning.
   ii. Both the add and multiply instructions are blocked fewer cycles due to this data prefetching.
   iii. The reordering should not change the end results.

2. Mechanisms for Instruction Pipelining

1. prefetch Buffers
2. Multiple Functional Units.
3. Internal Data Forwarding.
4. Hazard avoidance

1. prefetch Buffers

3 buffers to match the instruction fetch rate to the pipeline consumption rate.

i. sequential buffers
ii. target buffers
iii. loop buffers.

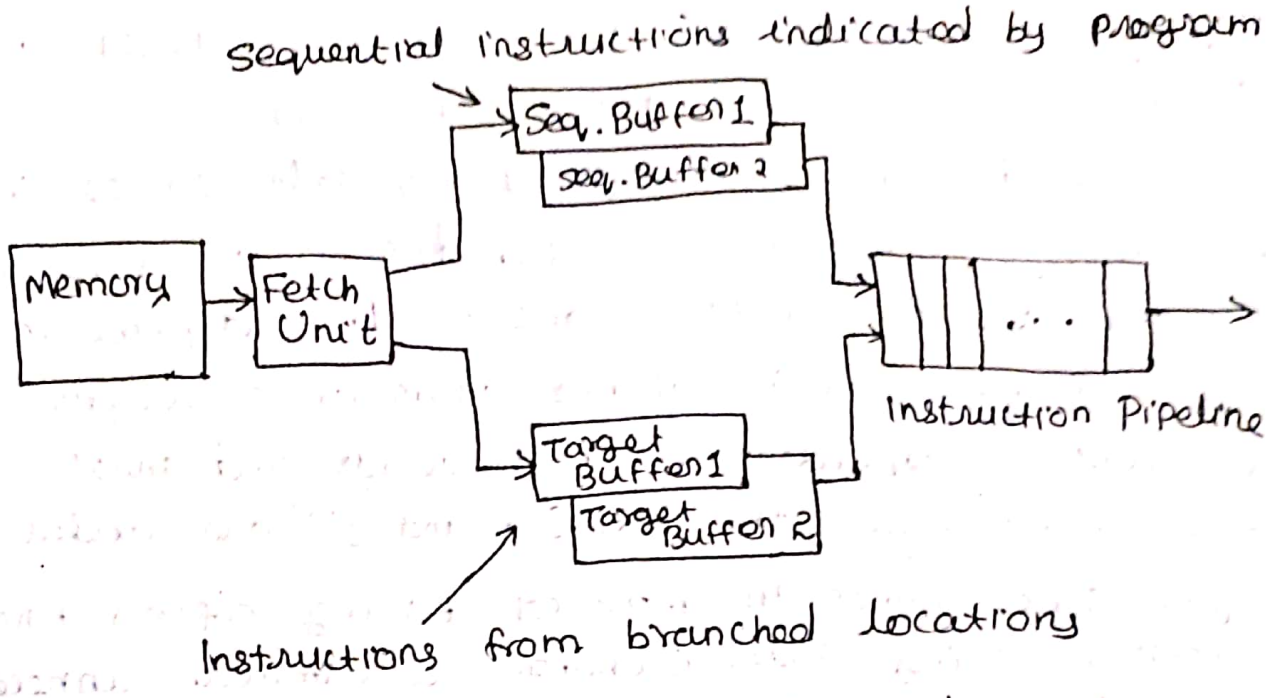


Figure 5.3 The use of sequential and target buffers.

i. sequential instructions are loaded into a pair of sequential buffers for in-sequence pipelining.

ii. Instructions from a branch target are loaded into a pair of target buffers for out-of-sequence pipelining.

iii. Both buffers operate in a first-in-first-out fashion.

iv. A third type of prefetch buffer is known as a loop buffer.

The loop buffers are maintained by the fetch stage of the pipeline. prefetched instructions in the loop body will be executed repeatedly until all iterations complete execution.

v. In one-memory-access time, a block of consecutive instructions are fetched into a prefetch buffer as illustrated in Figure 5.3.

2. Multiple Functional Units.

i. Sometimes a certain pipeline stage becomes the bottleneck.

ii. Bottleneck stage corresponds to the row with the maximum number of checkmarks in the reservation table.

iii. This bottleneck problem can be solved by using multiple copies of the same stage simultaneously.

iv. This leads to the use of multiple execution units in a pipelined processor design -

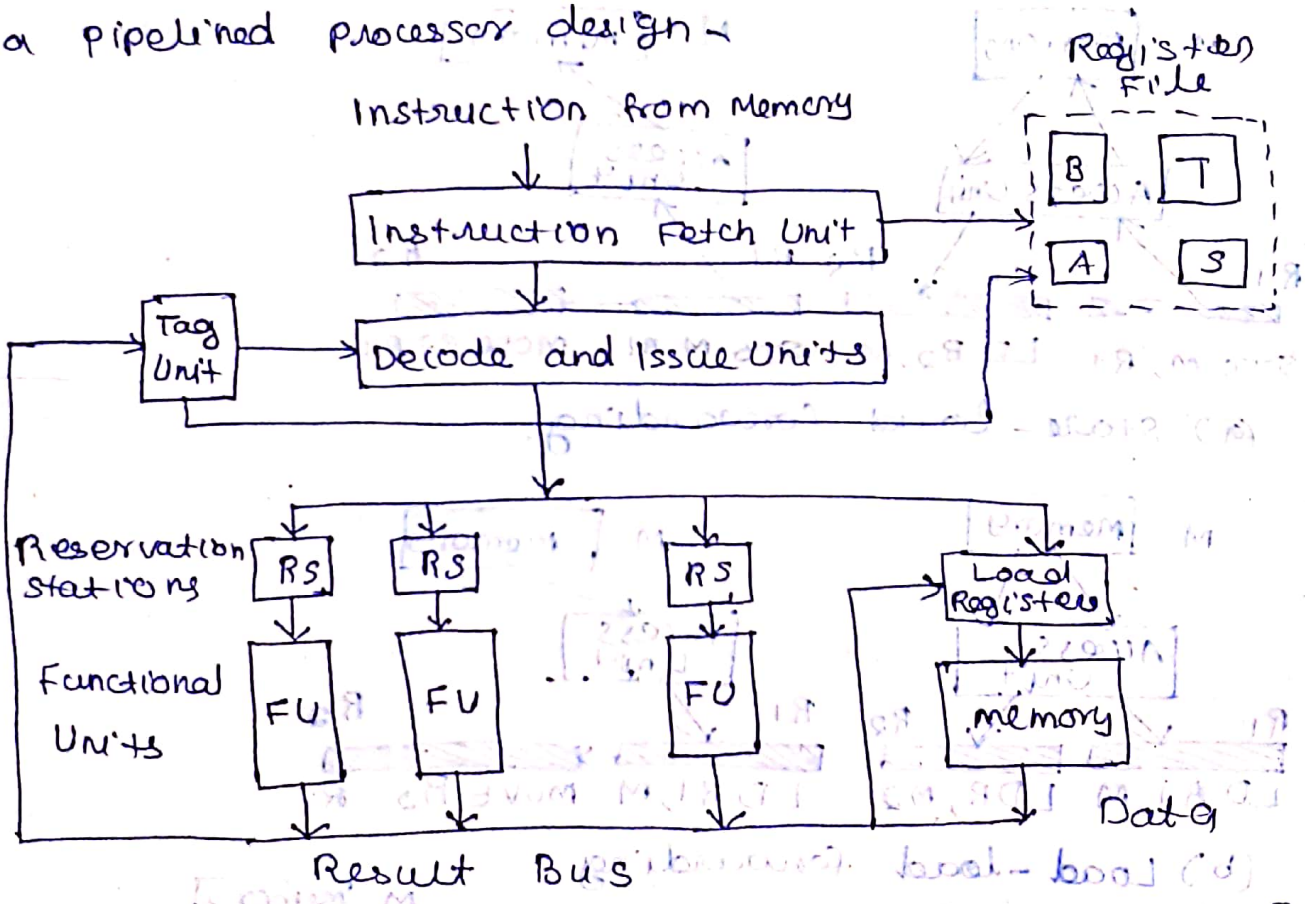


Figure 5.1 A pipelined processor with multiple functional units and distributed reservation stations supported by tagging

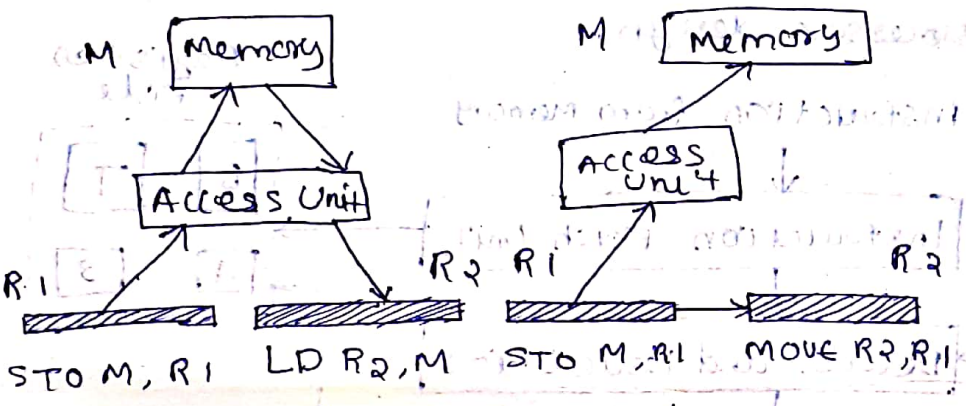
v. Operands can wait in the RS until its data dependences have been resolved.

vi. Each RS is uniquely identified by a tag, which is monitored by a tag unit.

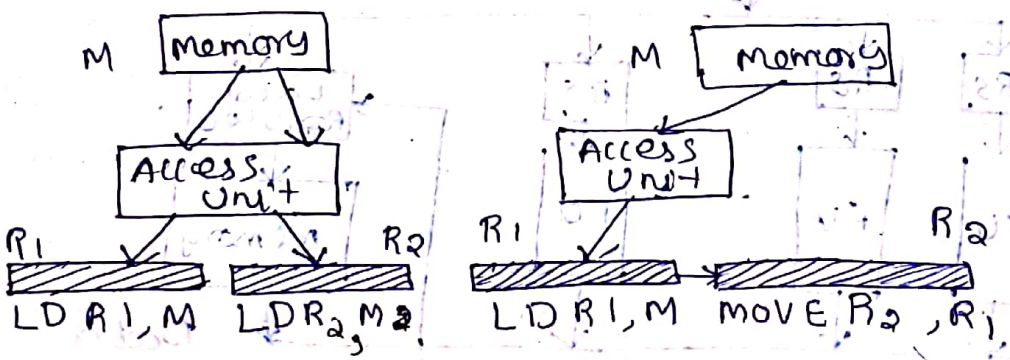
vii. The multiple functional units are supposed to operate in parallel, once the dependences are resolved.

3. Internal Data Forwarding.

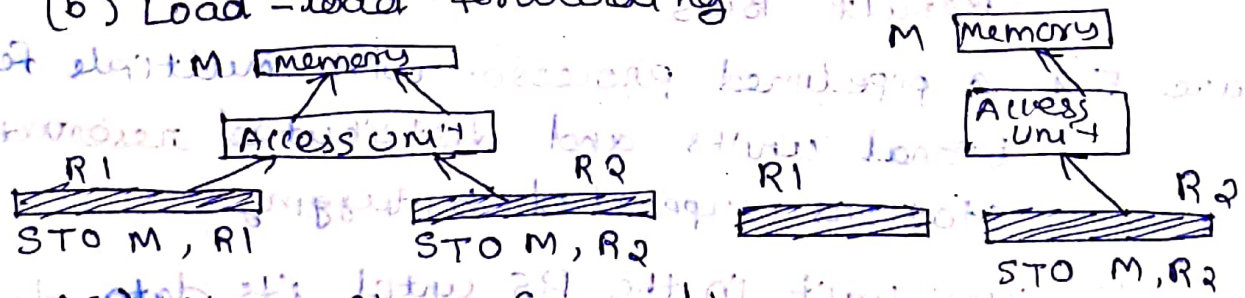
Internal data forwarding among multiple functional units.



(a) Store-load forwarding.



(b) Load-load forwarding



(c) Store-store forwarding due to overlapping.

Figure 5.5 Internal data forwarding by replacing memory-access operations with register transfer operations

Store-load forwarding

The load operation from memory to register R2 can be replaced by the move operation from register R1 to register R2.

Load-Load forwarding

Eliminates the second load operation and replaces it with the move operation.

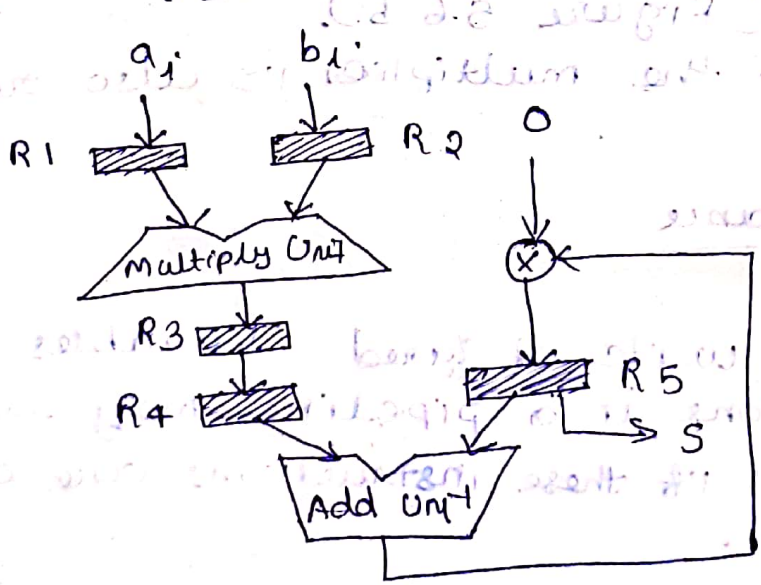
Store-Store forwarding

The two stores are executed immediately, one after another. Therefore, the second store overwrites the first store. The first store becomes redundant and thus can be eliminated without affecting the outcome.

Eg:

Implementing the dot-product operation with internal data forwarding between a multiply unit and an add unit.

$$S = \sum_{i=1}^{n} a_i b_i$$

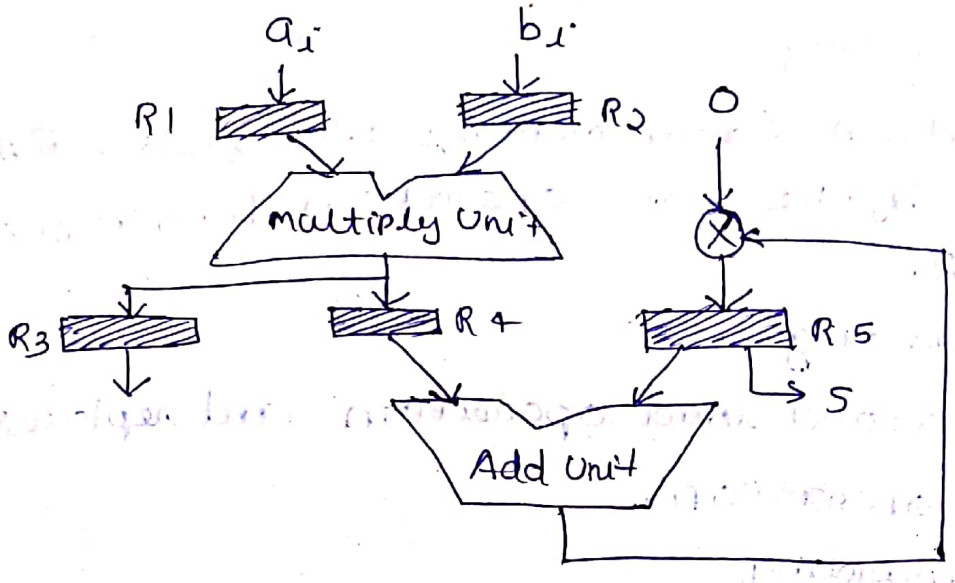


I_1 : R3 ← (R1) * (R2)

I_2 : R4 ← (R3)

I_3 : R5 ← (R5) + (R4)

(a) Without data forwarding



I_1' : $R_3 \leftarrow (R1) * (R2)$
I_2' : $R4 \leftarrow (R1) * (R2)$
I_3' : $R5 \leftarrow (R4) + (R5)$

I_1' and I_2' can be executed simultaneously with internal data forwarding.

(b) With internal data forwarding.

Figure 5.6 Internal data forwarding for implementing the dot-product operation.

i. Without internal data forwarding between the two functional units, the three instructions must be sequentially executed in a looping structure (Figure 5.6 a).

ii. With data forwarding, the output of the multiplier is fed directly into the input register R4 of the adder (Figure 5.6 b).

iii. The output of the multiplier is also routed to register R3.

4. Hazard Avoidance

Hazard:

The read and write of shared variables by different instructions in a pipeline may lead to different results if these instructions are executed out of order.

Three types of logic hazards are.

1. Read-after-write (RAW) hazard
2. write-after-write (WAW) hazard
3. Write-after-Read (WAR) hazard.

Let:

D(I) : domain of an instruction I. The domain contains the input set (such as operands in registers or in memory.) to be used by instruction I.

R(I) : range of an instruction I.

The range corresponds to the output set of instruction I.

Conditions under which possible hazards can occur:

$R(I) \cap D(J) \neq \phi$ for RAW hazard

$R(I) \cap R(J) \neq \phi$ for WAW hazard

$D(I) \cap R(J) \neq \phi$ for WAR hazard

These conditions are necessary but not sufficient. This means the hazard may not appear even if one or more of the conditions exist.

RAW - corresponds to the flow dependence,

WAR - corresponds to the anti-dependence, and

WAW - corresponds to the output dependence.

1. A special tag bit can be used with each operand register to indicate safe or hazard-prone. Successive read or write operations are allowed to set or reset

the tag bit to avoid hazards.

3. Dynamic Instruction Scheduling

Three methods for scheduling instructions through an instruction pipeline:

i. Static scheduling

ii. Dynamic scheduling

a. Tomasulo's register-tagging (IBM 360/91)

b. scoreboarding (CDC 6600).

i. Static scheduling.

consider the following code fragment:

Stage delay :	Instruction:
2 cycles	Add R0, R1 / R0 ← (R0) + (R1) /
1 cycle	Move R1, R5 / R1 ← (R5) /
2 cycles	Load R2, M(∝) / R2 ← (Memory(∝)) /
2 cycles	Load R3, M(β) / R3 ← (Memory(β)) /
3 cycles	Multiply R2, R3 / R2 ← (R2) x (R3) /

The two loads, since they are independent of the add and move, can be moved ahead to increase the spacing between them and the multiply instruction. After modification:

Load R2, M(∝)	2 to 3 cycles
Load R3, M(β)	2 cycles due to overlapping
Add R0, R1	2 cycles
Move R1, R5	1 cycle
multiply R2, R3	3 cycles.

i. Data dependences in a sequence of instructions create interlocked relationships among them.

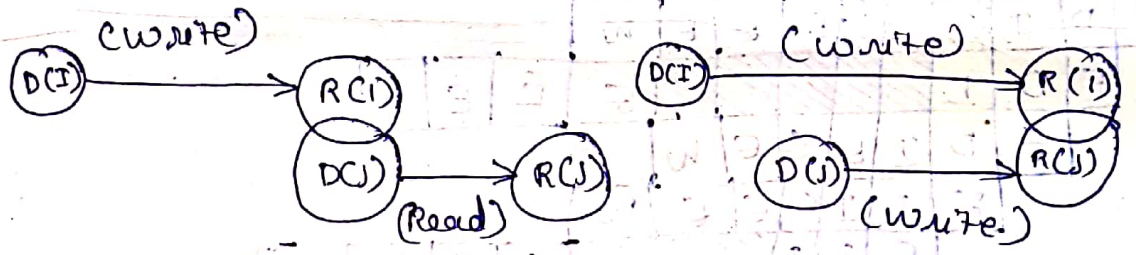
ii. Interlocking can be resolved through a compiler-based static scheduling approach.

iii a. Tomasulo's Algorithm.

1. Hardware dependence-resolution scheme.
2. The scheme resolves resource conflicts as well as data dependences using register tagging to allocate or deallocate the source and destination registers.

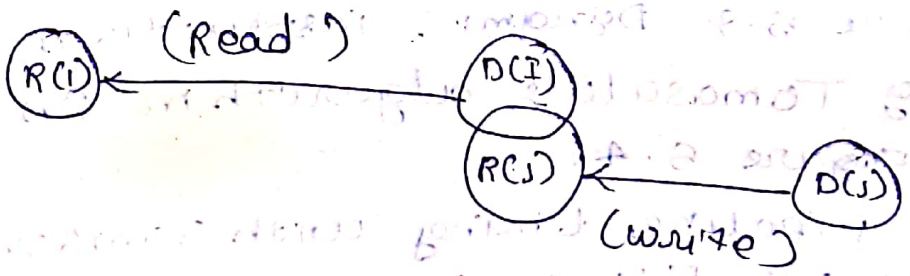
Eg: Tomasulo's algorithm for dynamic instruction scheduling.

Figure 5.8a shows a minimum-register machine code for computing $X = Y + Z$ and $A = B \times C$.



(a) Read-after-write (RAW) hazard

(b) Write-after-write (WAW) hazard

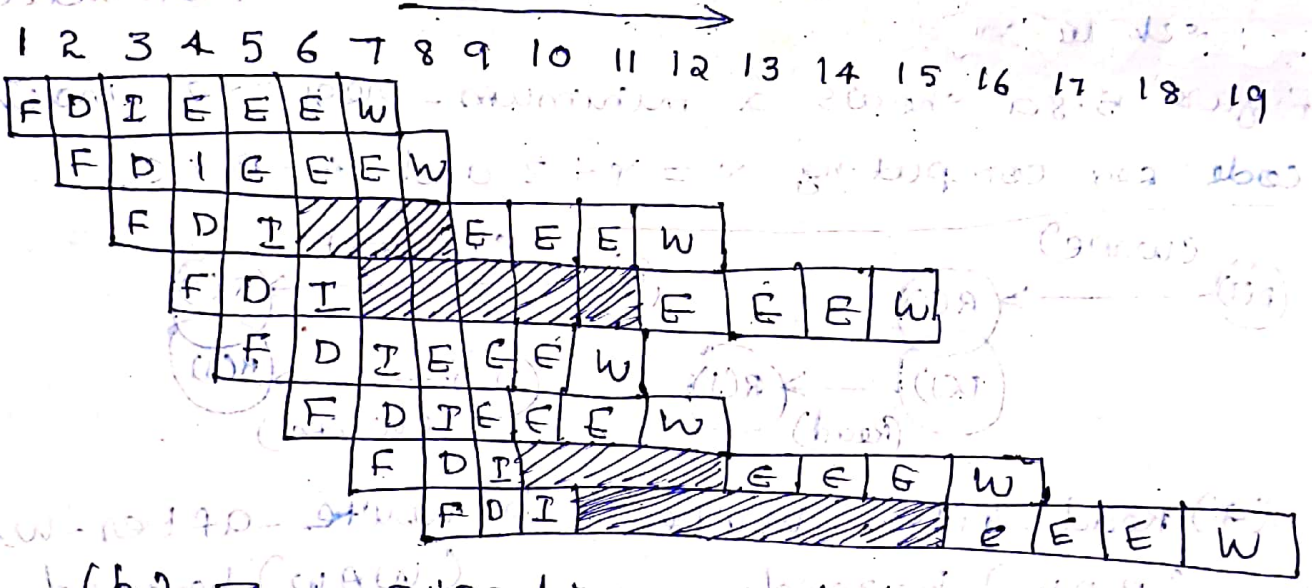


(c) Write-after-Read (WAR) hazard.

Figure 5.7 Possible hazards between read and write operations in an instruction pipeline (instruction I is ahead of instruction J in program order).

R1 ← Mem (Y)
R2 ← Mem (Z)
R3 ← (R1) + (R2)
Mem (X) ← (R3)
R1 ← Mem (B)
R2 ← mem (C)
R3 ← (R1) * (R2)
Mem (A) ← (R3)

(a) Minimum register machine code.



(b) The pipeline schedule

Figure 5.8 Dynamic instruction scheduling using Tomasulo's algorithm on the processor in Figure 5.4.

11. The pipeline timing with Tomasulo's algorithm appears in Figure 5-8 b.

12. The total execution time is 13 cycles counting from cycle 1 to cycle 15 by ignoring the pipeline startup and draining times.

1. An issued instruction whose operands are not available is forwarded to an RS associated with the functional unit it will use.
2. It waits until its data dependences have been resolved and its operands become available.
3. The dependence is resolved by monitoring the result bus.
4. When all operands for an instruction are available it is dispatched to the functional unit for execution.
5. ~~If~~ All working registers are tagged.
6. If a source register is busy when an instruction reaches the issue stage, the tag for the source register is forwarded to an RS.
7. When the register becomes available, the tag can signal the availability.

ii.b scoreboarding

1. The CDC 6600 was an early high-performance computer that used dynamic instruction scheduling hardware.
2. To control the correct routing of data between execution units and registers, the CDC 6600 used a centralized control unit known as the scoreboard.
3. This unit kept track of the registers needed by instructions waiting for the various functional units.
4. When all registers had valid data, the scoreboard enabled the instruction execution.

5. When a functional unit finished, it signaled the scoreboard to release the resources.

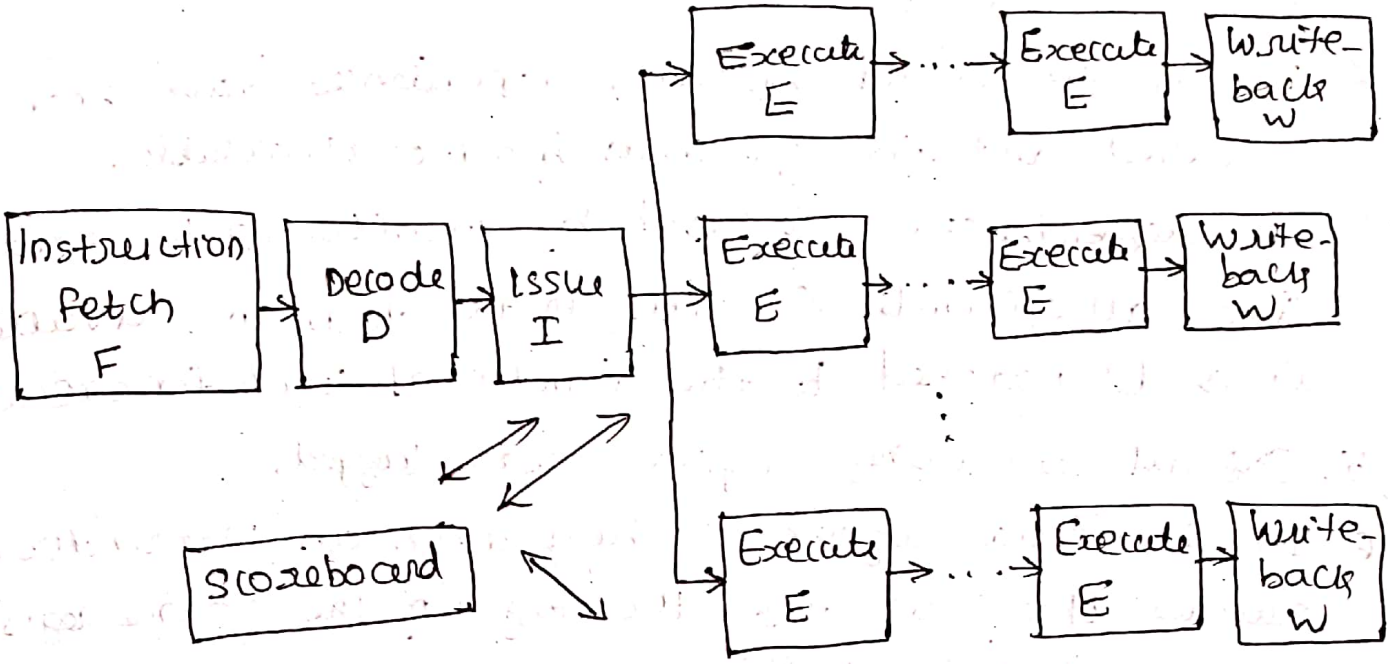


Fig. 5.9 A CDC 6600-like processor.

6. The scoreboard is a centralized control logic which keeps track of the states of registers and multiple functional units.
7. When functional units generate new results, some data dependences can be resolved and thus a higher degree of parallelism can be explored with scoreboarding.
8. Scoreboarding implements a kind of data-driven mechanism to achieve dataflow computations.

4. Branch Handling Techniques.

i. Effect of Branching
ii. Branch Prediction
iii. Delayed Branches.

i. Effect of Branching

1. Analysis of branching effects

a). Branch taken

The action of fetching a nonsequential or remote instruction after a branch instruction is called a branch taken.

b) Branch target

The instruction to be executed after a branch taken is called a branch target.

c) Delay slot

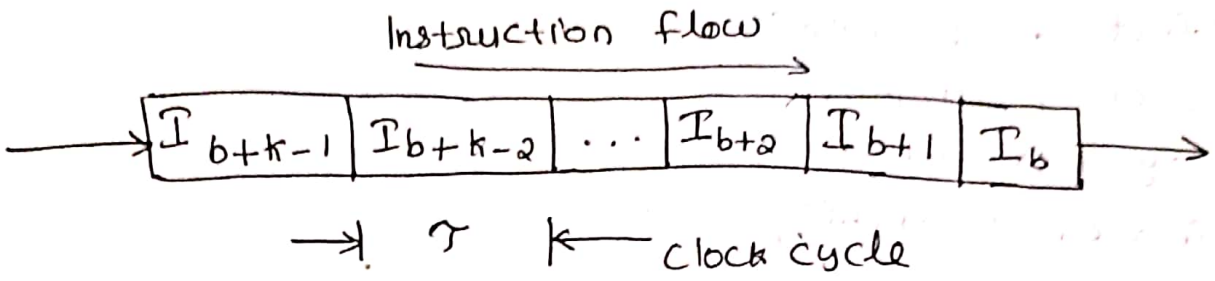
The number of pipeline cycles wasted between a branch taken and its branch target is called the delay slot, denoted by b.

$0 \leq b \leq k-1$, where k is the number of pipeline stages.

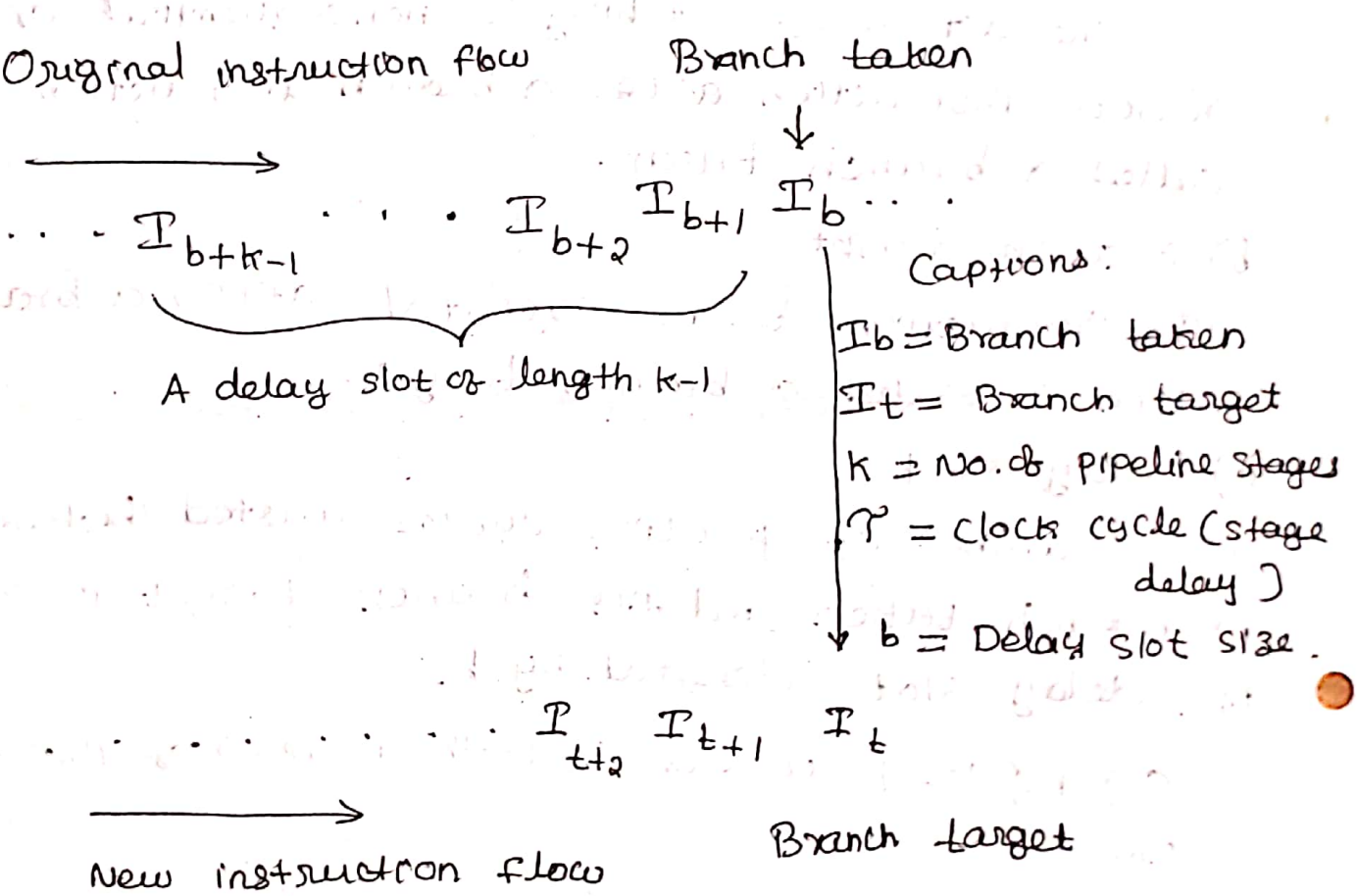
2. When a branch taken occurs, all the instructions following the branch in the pipeline become useless and will be drained from the pipeline.

3. A branch taken causes the pipeline to be flushed, losing a number of useful cycles.

Figure 5.10 illustrates the branch effects



(a) A k-stage pipeline



(b) An instruction stream containing a branch taken.

Figure 5.10 The decision of a branch taken at the last stage of an instruction pipeline causes $b \leq k-1$ previously loaded instructions to be drained from the pipeline

ii. Branch prediction

Branch can be predicted based on:

a) branch code types statically or

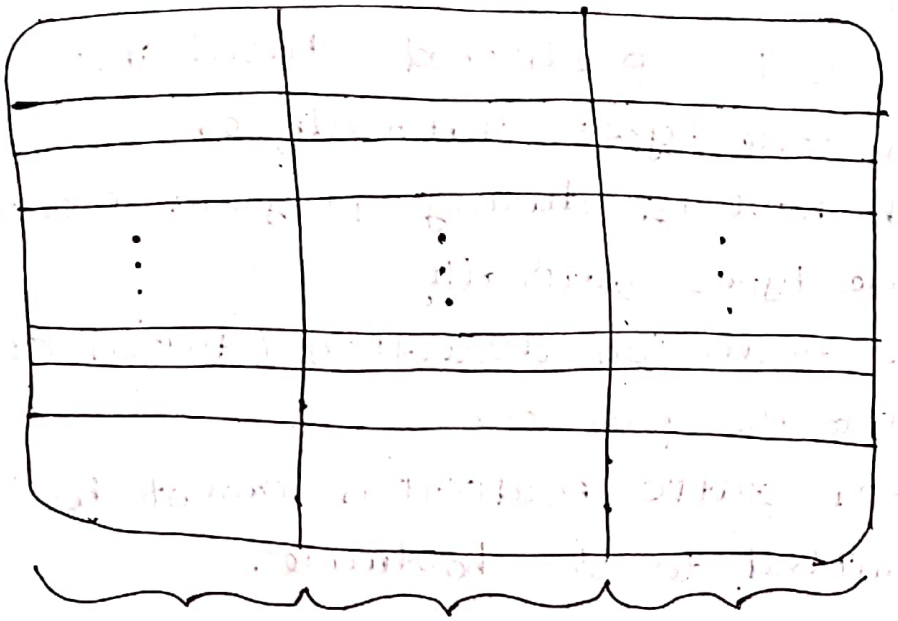
b) branch history during program execution.

a) Branch code types statically.

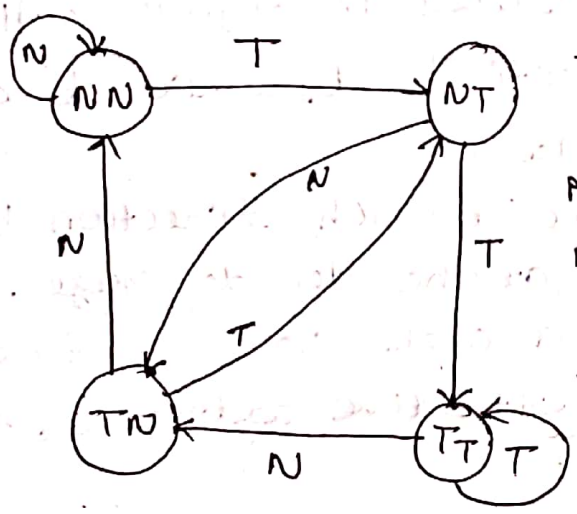
1. The static prediction direction (taken or not taken) is wired into the processor.
2. The wired-in static prediction cannot be changed once committed to the hardware.

b) Dynamic branch strategy.

1. Uses recent branch history to predict whether or not the branch will be taken next time, when it occurs.
2. Dynamic prediction is determined with limited recent history, as illustrated in Figure 5.11.
3. Cragon (1992) has classified dynamic branch strategies into three classes:
   i. One class predicts the branch direction based upon information found at the decode stage.
   ii. The second class uses a cache to store target addresses at the stage the effective address of the branch target is computed.
   iii. The third scheme uses a cache to store target instructions at the fetch stage.
4. Lee and Smith (1984) shown the use of a branch target buffer (BTB) to implement branch prediction. (Figure 5.11a).



(a) Branch target buffer organization



Captions:

T = Branch taken

N = Not-taken branch

NN = Last two branches not taken

NT = Not branch taken and previous taken

TT = Both last two branch taken

TN = Last branch taken and previous not taken

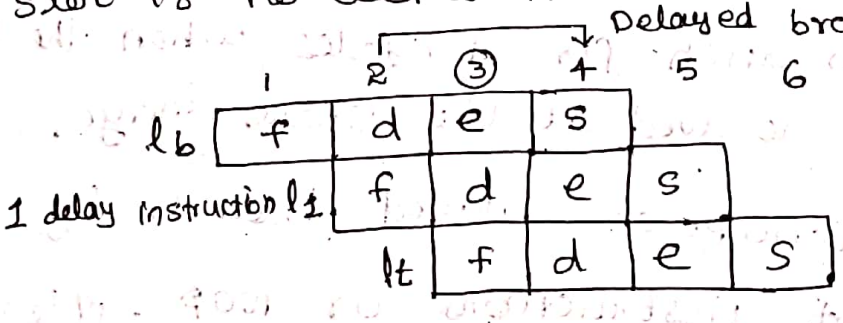
b) A typical state diagram.

Figure 5.11 Branch history buffer and a state transition diagram used in dynamic branch prediction.

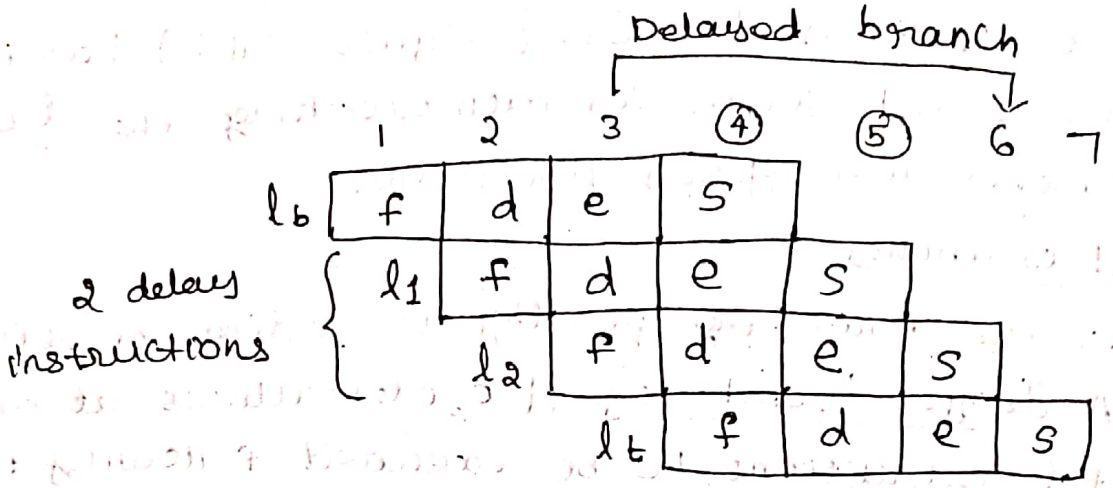
A state transition diagram (Figure 5.11 b) has been given by Lee and Smith for back tracking the last two branches in a given program.

(iii) Delayed Branches

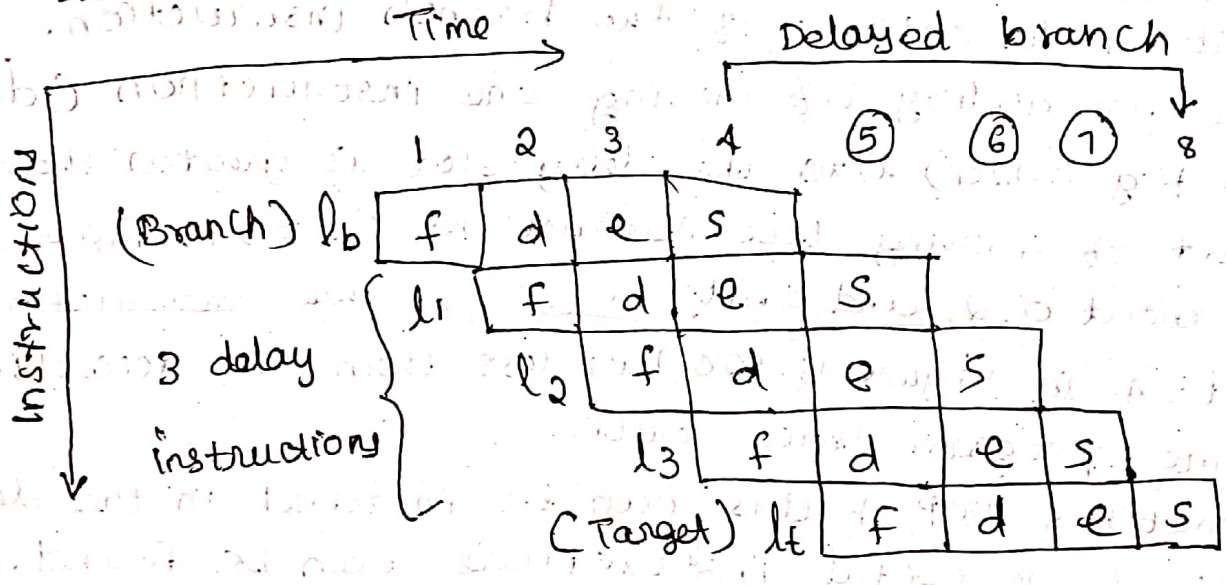
1. To reduce the branching penalty in coding microinstructions. A delayed branch of d cycles allows at most d-1 useful instructions to be executed following the branch taken.
2. The execution of these instructions should be independent of the outcome of the branch instruction.
3. The probability of moving one instruction (d=2 in Fig. 5.12a) into the delay slot is greater than 0.6, that of moving two instructions (d=3 in Figure 5.12b) is about 0.2, and that of moving three instructions (d=4 in Figure 5.12c) is less than 0.1, according to some program trace results.
4. Sometimes NOP fillers can be inserted in the delay slot if no useful instructions can be found.



(9) A delayed branch for 2 cycles when the branch condition is resolved at the decode stage.



b) A delayed branch for 3 cycles when the branch condition is resolved at the execute stage.



(c) A delayed branch for 4 cycles when the branch condition is resolved at the store stage.

Figure 5.12. The concept of delayed branch by moving independent instructions or NOP fillers into the delay slot of a four-stage pipeline.

Arithmetic Pipeline Design

1. Pipelining techniques can be applied to speed up numerical arithmetic computations.

i. Computer Arithmetic Principles
   a) Fixed point Operations
   b) Floating-Point Numbers
   c) Floating-Point Operations
   d) Elementary Functions.

ii. Static Arithmetic Pipelines
   a) Arithmetic Pipeline stages
   b) Multiply Pipeline Design
   c) Convergence Division

iii. Multifunctional Arithmetic Pipelines

i. Computer Arithmetic Principles.

The Institute of Electrical and Electronics Engineers (IEEE) has developed standard formats for 32- and 64-bit floating numbers known as the IEEE 754 standard. This standard has been adopted for most of today's computers.

a) Fixed-Point Operations.

1. Fixed-point numbers are represented internally in machines in
   1. sign-magnitude,
   2. One's complement, or
   3. two's complement notation.
2. Most computers use the two's complement notation because of its unique representation of all numbers,

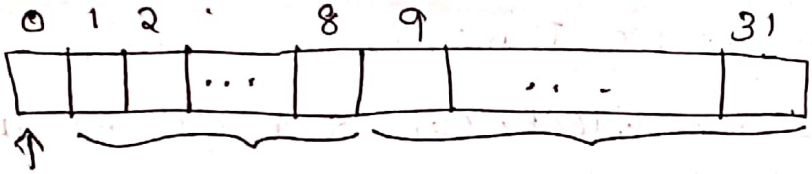
3. Four arithmetic operations

1. Add, } n-bit integers (or fractions) produces an
2. subtract } n-bit result with at most one carry-out.
3. multiply, and - n-bit numbers produces a 2n-bit result
4. divide - 2n bit divident and a 2n-bit divisor to yield a 2n-bit quotient.

b) Floating-Point Numbers

1. A floating-point number X is represented by a pair (m, e), where m is the mantissa (or fraction) and e is the exponent with an implied base (or radix).
2. The algebraic value is represented as $X = m \times r^e$.
3. The sign of X can be embedded in the mantissa.

Eg: The IEEE 754 floating-Point standard



Sign | Exponent e (Excess 127 code) | Mantissa m

(-127, 128)
(0, 255)

c) Floating-point operations.

The four primitive arithmetic operations are defined below for a pair of floating-point numbers represented by $X = (m_x, e_x)$ and $Y = (m_y, e_y)$. $e_x < e_y$ and base $r = 2$.

$$X + Y = (m_x \times 2^{e_x - e_y} + m_y) \times r^{e_y}$$

$$X - Y = (m_x \times 2^{e_x - e_y} - m_y) \times r^{e_y}$$

$$X \times Y = (m_x \times m_y) \times 2^{e_x + e_y}$$

$$X \div Y = (m_x \div m_y) \times 2^{e_x - e_y}$$

These operations identify the number of arithmetic operations involved in each floating-point function. These can be divided into two halves

1. one half is for exponent operations (comparing their relative magnitudes or adding / subtracting them);
2. The second half is for mantissa operations, including four-types of fixed-point operations.

d) Elementary Functions

Includes

1. trigonometric,
2. exponential,
3. logarithmic, and
4. transcendental functions.

Truncated polynomials or power series can be used to evaluate the elementary functions such as $\sin x$, $\ln x$, e^x, $\cos x$, $\tan^{-1} y$, $\sqrt{x}$, x^3 etc.

e. Static Arithmetic Pipelines:

1. Most of today's arithmetic pipelines are designed to perform fixed functions.

a) Arithmetic Pipeline Stages

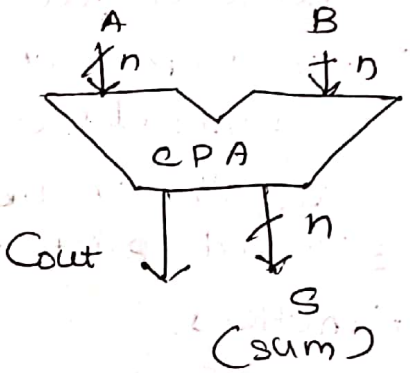
1. Since all arithmetic operations can be implemented with the basic add and shifting operations, the core arithmetic stages require some form of hardware to add or shift.
2. High-speed addition requires either the use of a

carry-propagation adder (CPA) which adds two numbers and produces an arithmetic sum as shown in Figure 5.13a.

3. To add three input numbers and produce one sum output and a carry output Carry-save adder can be used as shown in Figure 5.13b.

eg n=4

A = 1011
+B = 0111
S = 10010 = A+B

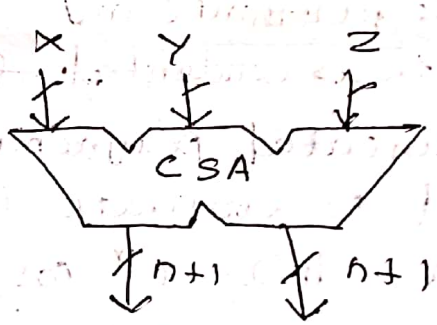


(a) An n-bit carry-propagate adder (CPA) which allows either carry propagation or applies the carry-lookahead technique

e.g n = 4

X = 0 0 1 0 1 1
Y = 0 1 0 1 0 1
(+) Z = 1 1 1 1 0 1
S^b = 0 1 0 0 0 1 1

(+) C = 0 1 1 1 0 1 0
S = 1 0 1 1 1 0 1 = S^b + C = X + Y + Z



(S^b (Bitwise sum), C (carry vector))

b) An n-bit carry-save adder (CSA), where S^b is the bitwise sum of X, Y, and Z, and C is a carry vector generated without carry propagation between digits.

Figure 5.13 Distinction between a carry-propagate adder (CPA) and a carry-save adder (CSA).

4. In a CPA, the carries generated in successive digits are allowed to propagate from the low end to the high end, using the ripple carry propagation or some carry lookahead technique.

5. In CSA, the carries are not allowed to propagate but instead are saved in a carry vector.

6. The CSA performs bitwise operations simultaneously on all columns of digits to produce two n-bit output numbers denoted as

$S^b = (0, S_{n-1}, S_{n-2}, \cdots, S_1, S_0)$ and

$C = (C_n, C_{n-1}, \cdots, C_1, 0)$.

7. Note that the leading bit of the bitwise sum S^b is always a 0, and the tail bit of the carry vector C is always a 0.

8. The input-output relationships can be expressed as

$$S_i = x_i \oplus y_i \oplus z_i$$

$$C_{i+1} = x_i y_i \vee y_i z_i \vee z_i x_i$$

$$S = S^b + C$$

9. CPA and CSA used to implement the pipeline stages of a fixed-point multiply unit.

b) Multiply Pipeline Design.

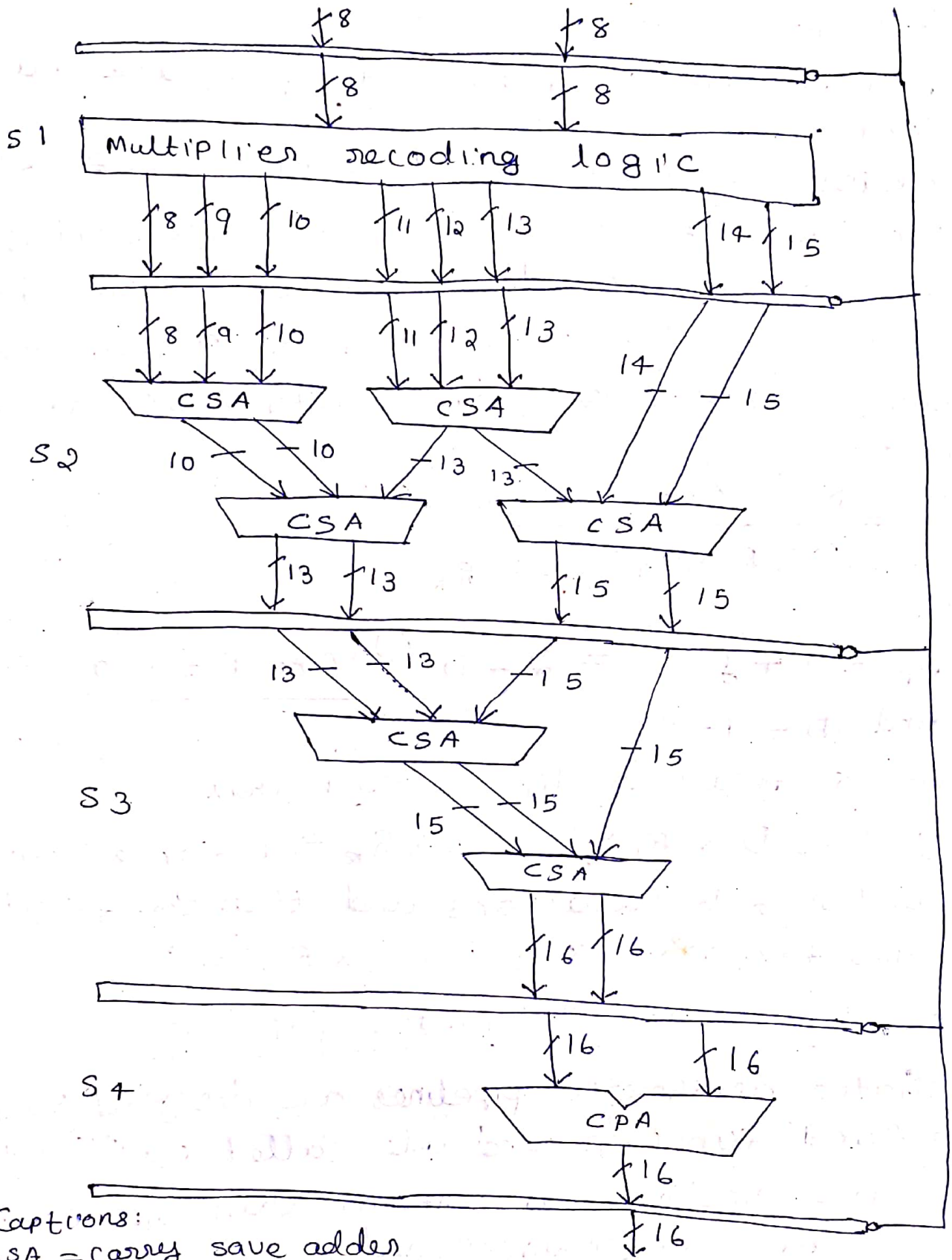
The partial product P_j is obtained by multiplying the multiplicand A by the jth bit of B and then shifting the result j bits to the left for

$j = 0, 1, 2, \ldots 7$. Thus P_j is $(8+j)$ bits long with j trailing zeros. The summation of the eight partial products is done with a Wallace tree of CSAs plus a CPA at the final stage as shown in Figure 5.14

```
                1 0 1 1 0 1 0 1  = A
            x)  1 0 0 1 0 0 1 1  = B
-------------------------------------
                1 0 1 1 0 1 0 1  = P0
              1 0 1 1 0 1 0 1 0  = P1
            0 0 0 0 0 0 0 0 0 0  = P2
          0 0 0 0 0 0 0 0 0 0 0  = P3
        0 0 0 0 0 0 0 0 0 0 0 0  = P4
      1 0 1 1 0 1 0 1 0 0 0 0 0  = P5
    0 0 0 0 0 0 0 0 0 0 0 0 0 0  = P6
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  = P7
+)1 0 1 1 0 1 0 1 0 0 0 0 0 0 0
-------------------------------------
  0 1 1 0 0 1 1 1 1 1 0 1 1 1 1  = P
```

In Figure 5.14

1. The first stage (S_1) generates all eight partial products, ranging from 8 bits to 15 bits, simultaneously.
2. The second stage (S_2) is made up of two levels of four CSAs, which essentially merges eight numbers into four numbers ranging from 13 to 15 bits.
3. The third stage (S_3) consists of two CSAs, which merges four numbers into two 16-bit numbers.
4. The final stage (S_4) is a CPA, which adds up the last two numbers to produce the final product P.



Captions:
CSA = Carry save adder
CPA = Carry propagate adder. $P = A \times B$

Figure 5.14 A pipeline unit for fixed-point multiplication of 8-bit integers.

c) Convergence Division

1. Division can be carried out by repeated multiplications.
2. Mantissa division is carried out by a convergence method.
3. convergence division obtains the quotient $Q = M/D$ of two normalized fractions $0.5 \leq M < D < 1$ in two's complement notation by performing two sequences of chain multiplications as follows:

$$Q = \frac{M \times R_1 \times R_2 \times \cdots \times R_k}{D \times R_1 \times R_2 \times \cdots \times R_k}$$

where

$R_i = 1 + \delta^{2^{i-1}} = 2 - D^{(i)}$ for $i = 1, 2, \ldots, k$

and $D = 1 - \delta$

for k iterations R_i's such that

$D^{(k)} = D \times R_1 \times R_2 \times \cdots \times R_k \to 1$ for a sufficient number of k iterations, and then the resulting numerator $M \times R_1 \times R_2 \times \cdots \times R_k \to Q$.

iii. Multifunctional Arithmetic Pipelines

1. Static arithmetic pipelines are designed to perform a fixed function and are called unifunctional.
2. When a pipeline can perform than one function, it is called multifunctional.
3. A multifunctional pipeline can be either static or dynamic.

4. Static pipelines perform one function at a time but different functions can be performed at different times.
5. A dynamic pipeline allows several functions to be performed simultaneously through the pipeline.

(Advanced Scientific Computer)

Eg: The TI / ASC, arithmetic processor design.

1. A static multifunctional pipeline.
2. There are four pipeline arithmetic units built into the TI-ASC system, as shown in Figure 5.15.
3. The instruction-processing unit handles the fetching and decoding of instructions.
4. There are two sets of operand buffers $\{X, Y, Z\}$ and $\{X', Y', Z'\}$, in each arithmetic unit.
5. X', X, Y', and are used for input operands.
6. $\{X', Y', Z'\}$ Z' and Z are used to output results.
7. Each pipeline arithmetic unit has eight stages as shown in Figure 5.16a.
8. The PAU is a static multifunction pipeline which can perform only one function at a time.
9. The PAU also supports vector arithmetic operations.
10. Fixed-point multiplication requires the use of only segments S_1, S_6, S_7, and S_8 as shown in Figure 5.16b.

11. This dot product is implemented by essentially the following accumulated summation of a sequence of multiplications through the pipeline:

$$Z \leftarrow A_i \times B_i + Z$$

where the successive operands (A_i, B_i) are fed through the X- and Y- buffers, and the accumulated sums through the Z-buffer recursively.

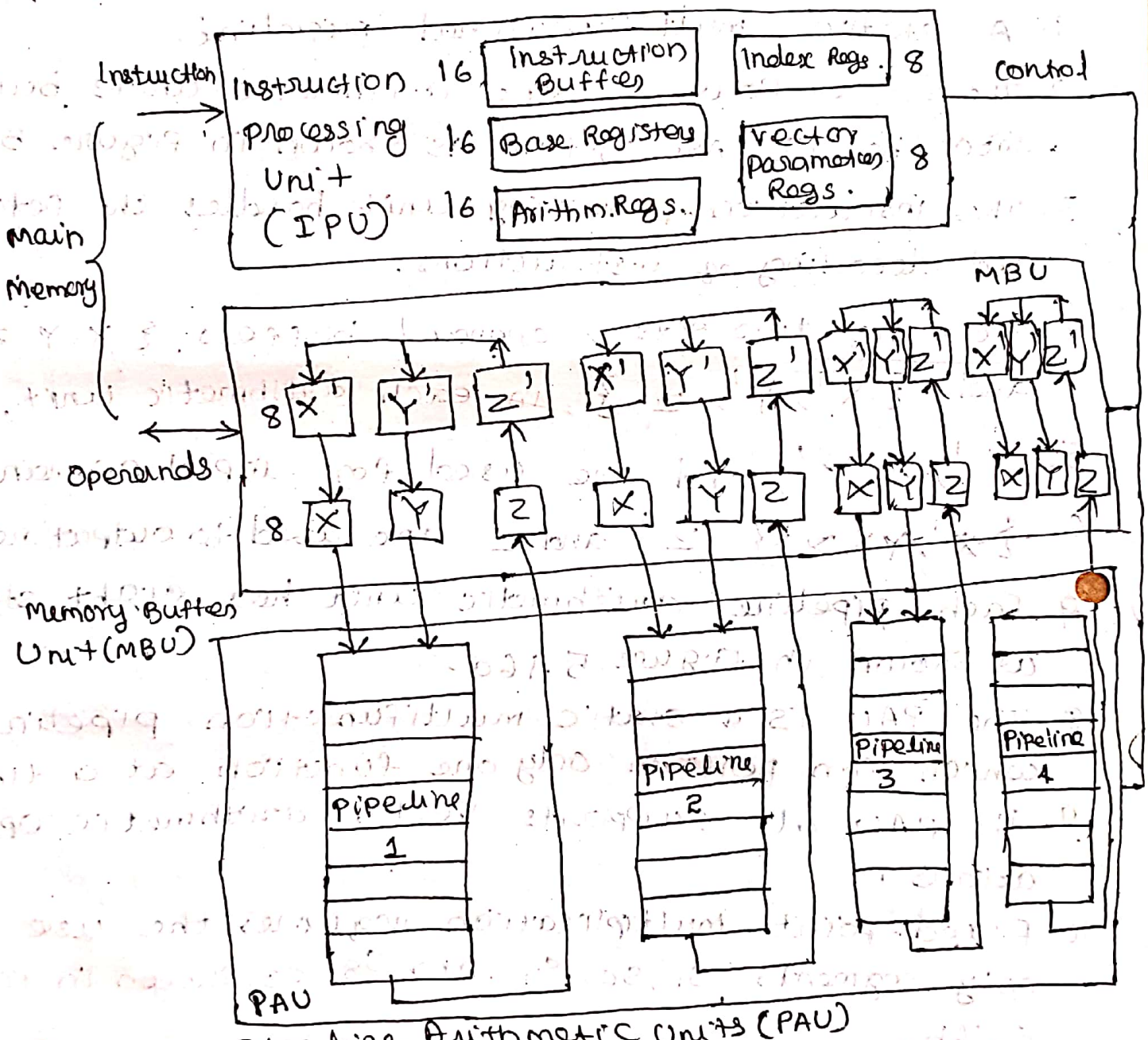
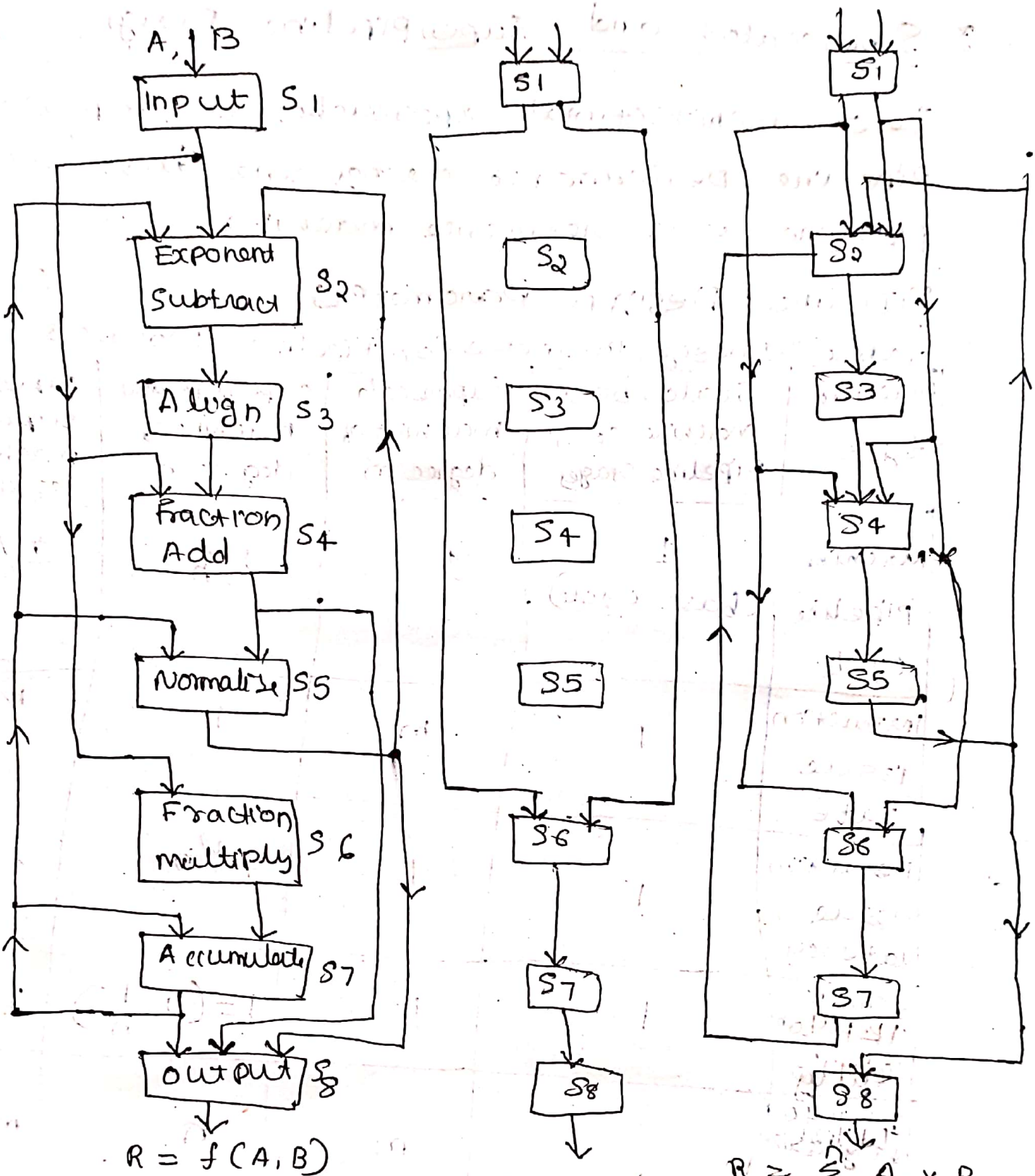


Figure 5.15. The architecture of the TI Advanced Scientific computer (ASC)



(a) Pipeline stages and interconnections

(b) Fixed-point multiplication.

(c) Floating-point dot product

Figure 5.16 The multiplication arithmetic pipeline of the TI Advanced scientific computer and the interstage connections for two representative functions.

3. Superscalar and Superpipeline Design.

Two architectural approaches to improve pipeline performance using the base scalar pipeline as a reference machine.

Pipeline Design Parameters

Table 5.1 Design Parameters for Pipeline Processors.

Machine type	Scalar base machine of k pipeline stages	Superscalar machine of degree m	superpipelined machine of degree n	superpipelined superscalar machine of degree (m, n)
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 (= n · 1/n)	1 (= n · 1/n)
I.L.P to fully utilize the pipeline	1	m	n	mn

Note: All timing is relative to the base cycle for the scalar base machine. ILP: Instruction level parallelism.

1. Superscalar Pipeline Design
2. Superpipelined Design.
3. Supersymmetry and Design Tradeoffs

1. Superscalar Pipeline Design

Structure of super scalar pipelines, the data dependence problem, the factors causing pipeline stalling, and multiinstruction-issuing mechanisms for achieving parallel pipelining operations.

a) Superscalar Pipeline structure, a.1 Data Dependences
b) Pipeline Stalling
c) Multipipeline scheduling
d) In-order Issue
e) Out-of-Order Issue
f) Motorola 88110 Architecture
g) Superscalar Performance.

a) Superscalar Pipeline structure

For a superscalar machine of degree m, m instructions are issued per cycle and the ILP should be m in order to fully utilize the pipeline.

Resource-Shared multiple-pipeline structure is illustrated by a design example in Figure 5.19.

In this design:

1. the processor can issue two instructions per cycle if there is no resource conflict and no data dependence problem.
2. Two pipelines in the design
3. Both pipelines have four processing stages labeled fetch, decode, execute, and store, respectively.

4. The two instruction streams flowing through the two pipelines are retrieved from a single source stream (the I-cache).
5. The fan-out from a single instruction stream is subject to resource contraints and a data dependence relationship among the successive instructions.
6. Four functional units, multiplier, adder, logic unit, and load unit, are available for use in the execute stage.
7. These functional units are shared by the two pipelines on a dynamic basis.
8. The multiplier itself has three pipeline stages, the adder has two stages, and the others each have only one stage.
9. The two store units (S1 and S2) can be dynamically used by the two pipelines, depending on availability at a particular cycle.
10. It is difficult to schedule multiple pipelines simultaneously, especially when the instructions are retrieved from the same source.

9.1. Data Dependences

Consider the example 5.17b:

1. Because the register content in R1 is loaded by I1 and then used by I2, we have flow dependence: $I1 \rightarrow I2$.
2. Because the result in register R4 after executing I4 may affect the operand register R4 used by I3, we have antidependence: $I3 \leftarrow I4$.
3. Since both I5 and I6 modify the register R6

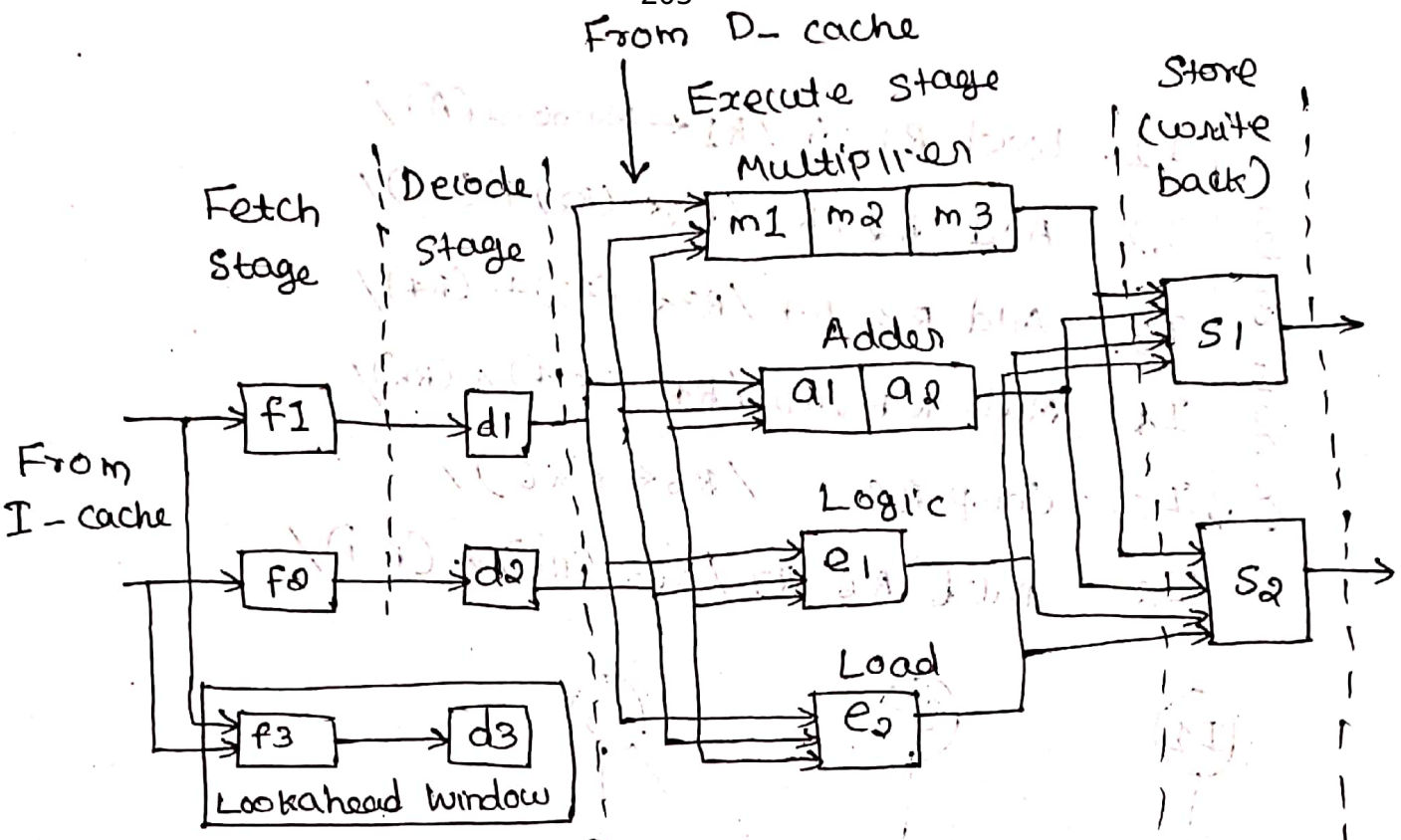


Figure 5.17(a) A dual-pipeline, superscalar processor with four functional units in the execution stage and a lookahead window producing out-of-order issues.

and R6 supplies an operand for I6, we have both flow and output dependence:

I5 → I6 and I5 ⟶ I6 as shown in the dependence graph.

a. To schedule instructions through one or more pipelines, these data dependences must not be violated.

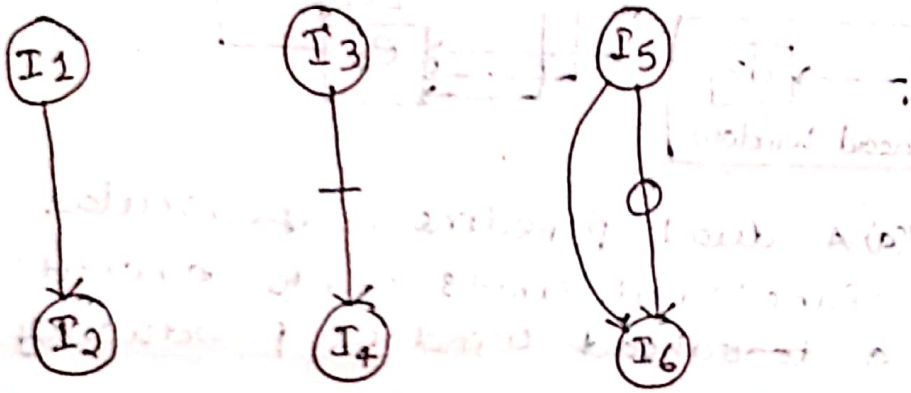
b. Pipeline Stalling:

1. Hazards are problems with the instruction pipeline in CPU microarchitectures when the next instruction cannot execute in the following clock cycle.

2. A pipeline stall is a delay in execution of an instruction in order to resolve a hazard.

Program order

I1. Load R1, A /R1 ← Memory(A)/
I2. Add R2, R1 /R2 ← (R2) + (R1)/
I3. Add R3, R4 /R3 ← (R3) + (R4)/
I4. Mul R4, R5 /R4 ← (R4) * (R5)/
I5. Comp R6 /R6 ← (R6)'/
I6. MUL R6, R7 /R6 ← (R6) * (R7)/



Flow dependence

Anti-dependence

Output-dependence, also flow dependence

Figure 5.17b A simple program and its dependence graph, where I2 and I3 share the adder and I4 and I6 share the same multiplier.

Figure 5.17 A two-issue superscalar processor and a sample program for parallel execution.

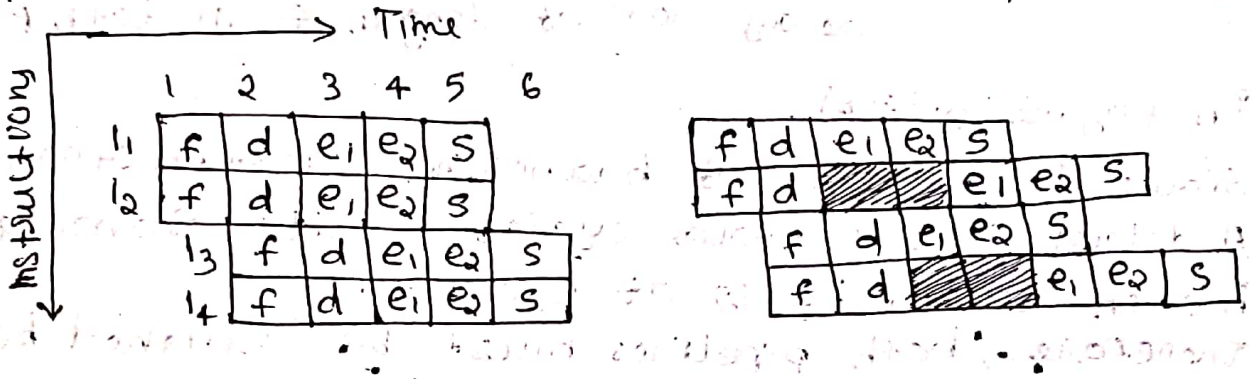
3. pipeline stalling problem may seriously lower pipeline utilization.
4. proper scheduling avoids pipeline stalling.
5. The problem exists in both scalar and super-scalar processors.

6. It is more serious in a superscalar pipeline.

7. Stalling can be caused by data dependences or by resource conflicts among instructions already in the pipeline or about to enter the pipeline.

Example: To illustrate the conditions causing pipeline stalling.

Consider the scheduling of two instruction pipelines in a two-issue superscalar processor.



(No data dependence) (I2 uses data generated by I1)

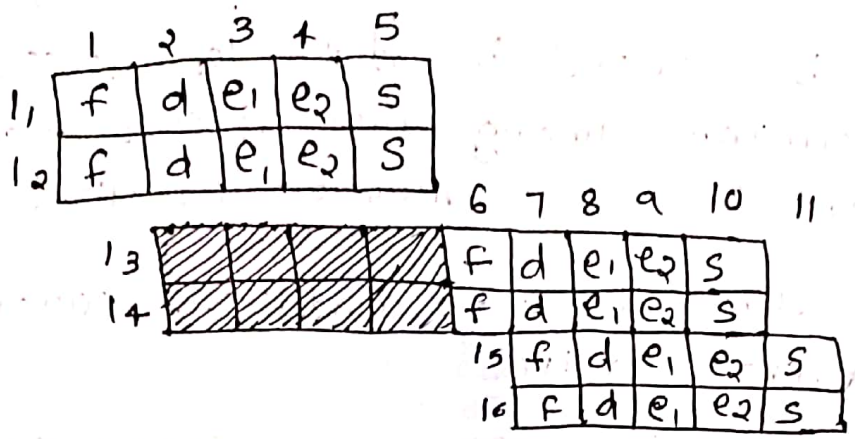
5.18 (a) Data dependence stalls the second pipeline in shaded cycles.

i. Figure 5.18(a) shows the case of no data dependence on the left and flow dependence ($I_1 \rightarrow I_2$) on the right.

ii. Without data dependence, all pipeline stages are utilized without idling.

iii. With dependence, instruction I2 entering the second pipeline must wait for two cycles (shaded time slots) before entering the execution stages.

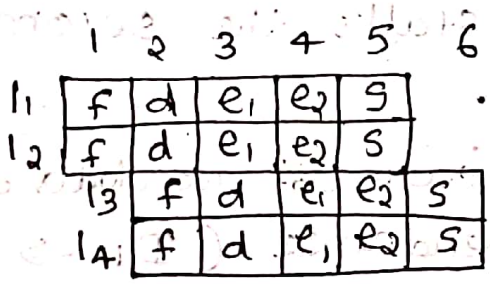
iv. This delay also pass to the next instruction I4 entering the pipeline.



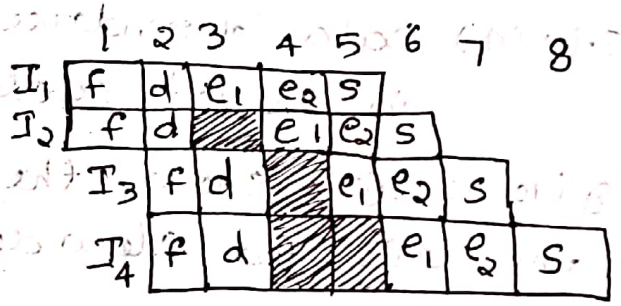
Captions:
f = fetch
d = decode
e1 = execute 1
e2 = execute 2
s = store (write-back)

Figure 5.18(b) Branch instruction I_2 causes a delay slot of length 4 in both pipelines.

In Figure 5.18(b),
(i) Shows the effect of branching (instruction I_2).
(ii) A delay slot of four cycles results from a branch taken by I2 at cycle 5.
(iii) Therefore, both pipelines must be flushed before the target instructions I_3 and I_4 can enter the pipelines from cycle 6.



(No resource conflicts)



(I1 and I_2 conflict in using the same functional unit, and I4 uses data generated by I2)

Figure 5.18 (c) Resource conflicts and data dependences cause the stalling of pipeline operations for some cycles.

Figure 5.18. Dependences and resource conflicts may stall one or two pipelines in a two-issue superscalar processor.

In Figure 5.18(c)

i. Shows a combined problem involving both resource conflicts and data dependence.

ii. Instructions I_1 and I_3 need to use the same functional unit, and $I_2 \rightarrow I_4$ exists.

iii. The net effect is that I_2 must be scheduled one cycle behind because the two pipeline stages (e_1 and e_2) or the same functional unit must be used by I_1 and I_2 in an overlapped fashion.

iv. The shaded boxes in all the timing charts correspond to idle stages.

c. Multipipeline scheduling

1. Instruction issue and completion policies are critical to superscalar processor performance.
2. There are three scheduling policies.
   i. In-order issue with in-order completion
   ii. In-order issue and out-of-order completion
   iii. Out-of-order issue and out-of-order completion

d) In-order Issue

1. When instructions are issued in program order, it is called as in-order issue.
2. If the instructions must be completed in program order then it is called in-order completion.
3. Figure 5.19a shows a schedule for the six instructions being issued in program order $I_1, I_2, \cdots, I_6$.
4. Pipeline 1 receives I_1, I_3 and I_5, and pipeline 2 receives instructions I_2, I_4, and I_6 in three consecutive cycles.

5. Due to $I1 \rightarrow I2$, I2 has to wait one cycle to use the data loaded in by I1.
6. I3 is delayed one cycle for the same adder used by I2.
7. I6 has to wait for the result of I5 before it can enter the multiplier stages.
8. In order to maintain in-order completion, I5 is forced to wait for two cycles to come out of Pipeline 1.
9. In total, nine cycles are needed and five idle cycles (shaded boxes) are observed.

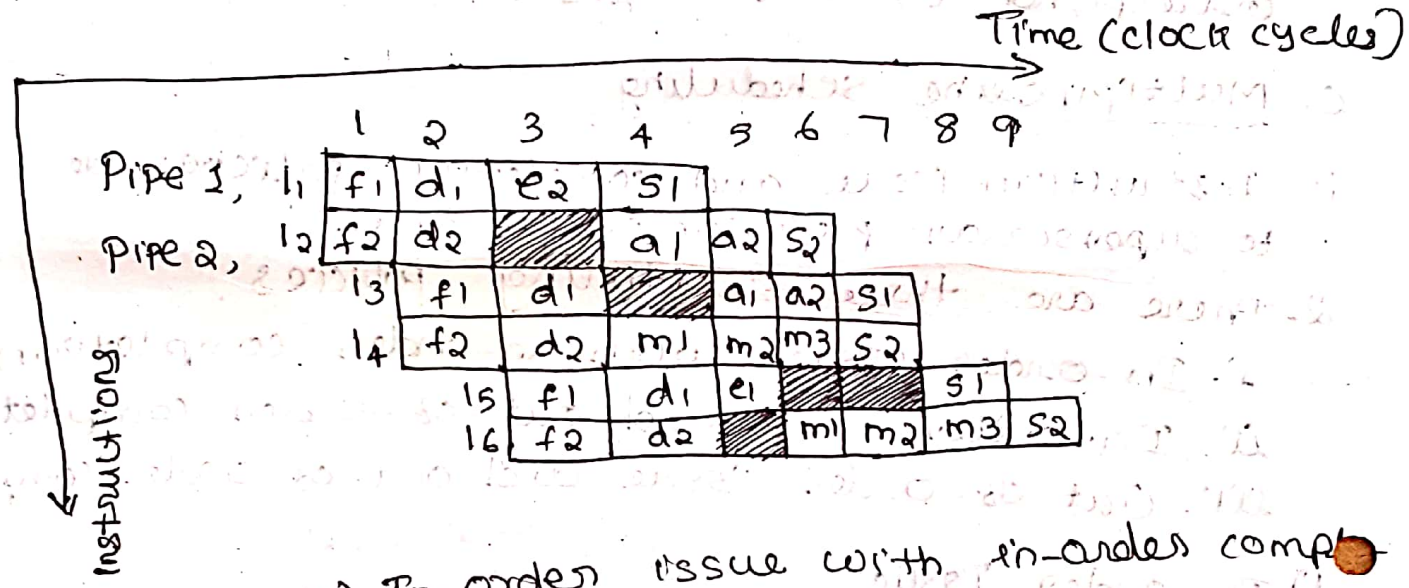


Figure 5.19 (a) In-order issue with in-order completion in nine cycles

9. In Figure 5.19b.
i. If the instructions completed out-of program order, it is called out-of-order completion.
ii. Out-of-order completion is allowed even if in-order issue is practiced.
iii. The only difference between this out-of-order schedule and the in-order schedule is that I5 is allowed to

complete ahead of I3 and I4, which are totally independent of I5.

iv. The total execution time does not improve.

e) Out-of-Order Issue (Figure 5.19(c))

1. When program order is violated, out-of-order issue is being practiced.

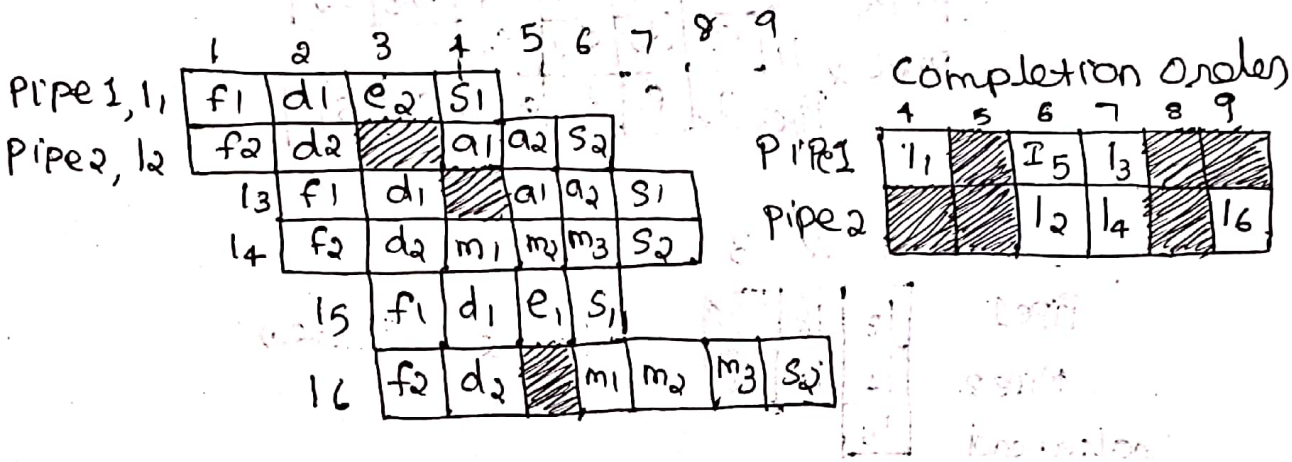


Figure 5.19(b) In-order issue and out-of-order completion in nine cycles.

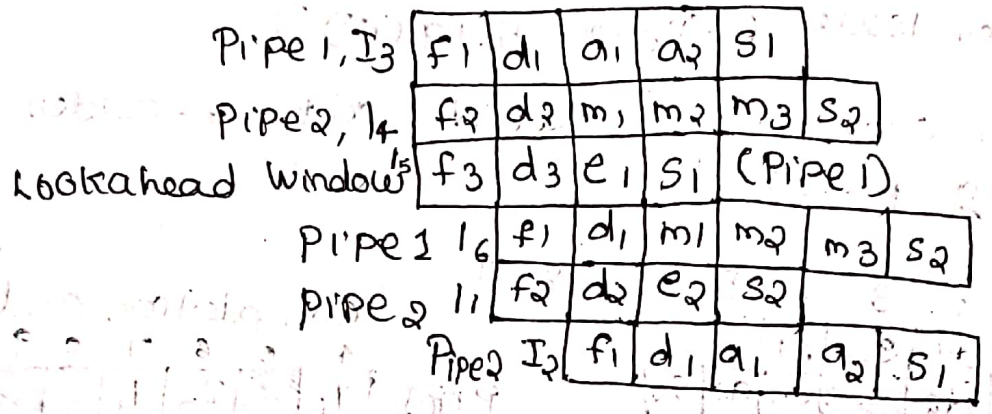
2. By using lookahead window, instruction I5 can be decoded in advance because it is independent of all the other instructions (Figure 5.19(c)).

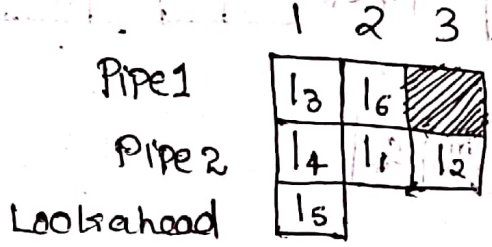
3. The six instructions are issued in three cycles as shown in Figure 5.19(c): I5 is fetched and decoded by the window, while I3 and I4 are decoded concurrently.

4. It is followed by issuing I6 and I1 at cycle 2 and I2 at cycle 3. Because the issue is out of order, the completion is also out of order as shown in Figure 5.19(c).

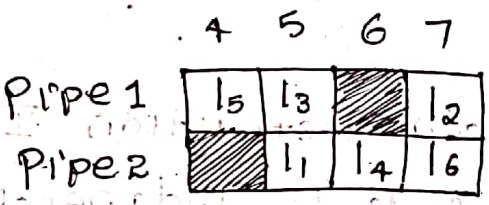
5. Now, the total execution time has been reduced to seven cycles with no idle stages during the execution

of these six instructions.





Issue order



Completion order

Figure 5.19 (c) out-of-order issue and out-of-order completion in seven cycles using an instruction lookahead window in the recoding process

Figure 5.19 Instruction issue and completion policies for a superscalar processor with and without instruction lookahead support.

f) Motorola 8810 Architecture

The Motorola 8810 is a superscalar RISC microprocessor.

i. Three-chip set,
ii. One CPU (88100) chip and two cache (88200) chips,
iii. It has advanced techniques for
a) Instruction-level parallelism, including
b) instruction issue, out-of-order instruction completion, speculative execution, dynamic instruction rescheduling, and two on-chip caches.
iv. It supports demanding graphics and digital signal processing applications.

g. Superscalar performance

1. The time required by the scalar base machine is
$T(1,1) = K + N - 1$ (base cycles)

where,
N is the number of independent instructions.

2. The ideal execution time required by an m-issue superscalar machine is
$T(m,1) = K + \frac{N-m}{m}$ (base cycles)

where,
K is the time required to execute the first m instructions through the m pipelines simultaneously,
$\frac{N-m}{m}$ is the time required to execute the remaining N-m instructions, m per cycle, through m pipelines.

3. The ideal speedup of the superscalar machine over the base machine is

the base machine is

$$S(m,1) = \frac{T(1,1)}{T(m,1)} = \frac{N+k-1}{N/m+k-1} = \frac{m(N+k-1)}{N+m(k-1)}$$

As $N \to \infty$, the speedup limit $S(m,1) \to m$, as expected.

2. Superpipelined Design

1. In a superpipelined processor of degree n, the pipeline cycle time is 1/n of the base cycle.
2. Figure 5.20(a) shows the execution of instructions with a superpipelined machine of degree $n = 3$.
3. Superpipelining is not possible without a high-speed clocking mechanism.

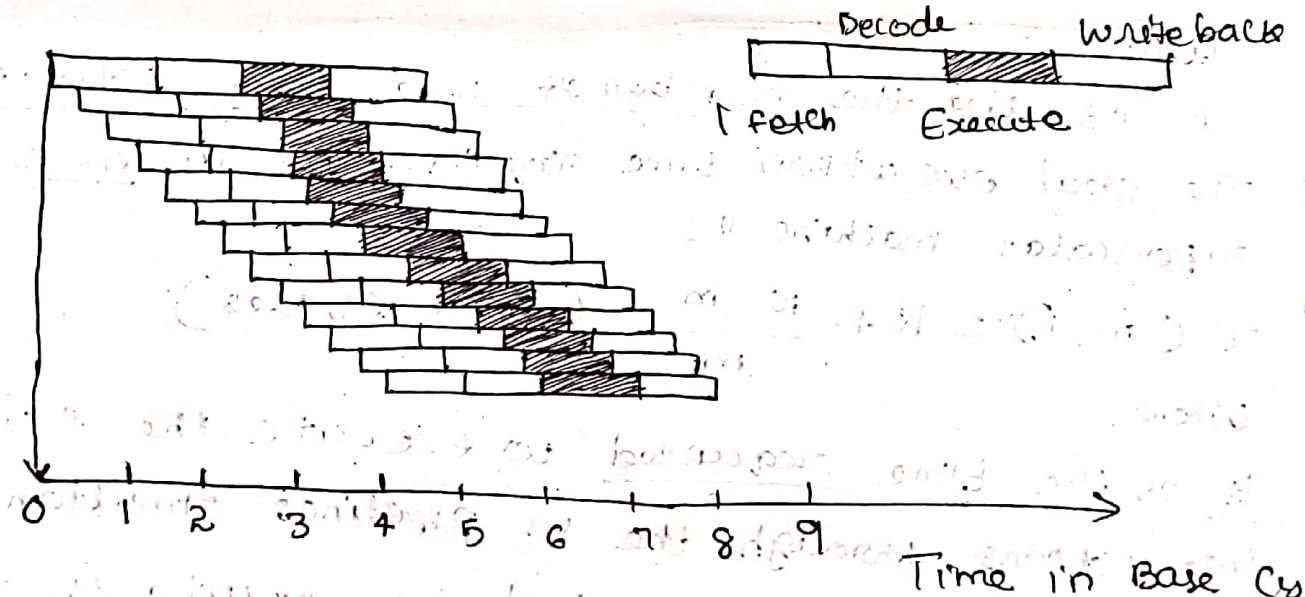


Figure 5.20(a) superpipelined execution with degree $n = 3$.

i. Super pipeline performance
ii. Superpipelined superscalar Design
iii. superpipelined superscalar performance

e. Superpipeline performance.

The minimum time required to execute N instructions for a superpipelined machine of degree n with k stages in the pipeline is

$$T(1,n) = k + \frac{1}{n}(N-1) \quad \text{(base cycles)}$$

Thus, the potential speedup of a superpipelined machine over the base machine is

$$S(1,n) = \frac{T(1,1)}{T(1,n)} = \frac{k+N-1}{k+(N-1)/n} = \frac{n(k+N-1)}{nk+N-1}$$

The speedup $S(1,n) \to n$, as $N \to \infty$.

The superpipeline and superscalar approaches can be combined in building so-called superpipelined superscalar processors.

f. Superpipelined Superscalar Design

A superpipelined superscalar processor of degree (m,n) is shown in Figure 5.20(b) with $(m,n) = (3,3)$.

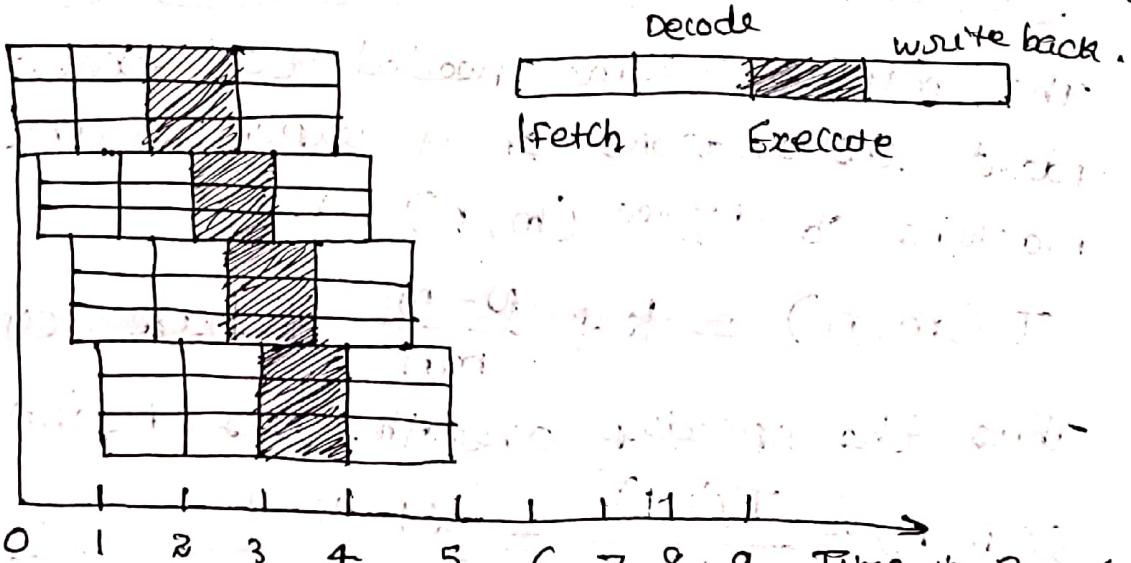


Figure 5.20(b) superpipelined superscalar execution with degree $m=n=3$

Figure 5.20 superpipelined processor architectures without and with multiple instruction issues respectively.

This machine executes m instructions every cycle with a pipeline cycle 1/n of the base cycle.
The level of parallelism required to fully utilize this machine is mn instructions.
Eg: DEC Alpha processor, R1000

Super scalar Design	Superpipelined design
1. More transistors required	Faster transistors required
2. Implemented in CMOS	Implemented in Cray, NEC
3. Rely on spatial parallelism	Emphasize temporal parallelism.
4. Duplicating hardware resources such as execution units and register file ports.	Overlapping multiple operations on a common piece of hardware

iii. Superpipelined Superscalar Performance

The minimum time needed to execute N independent instructions on a superpipelined superscalar machine of degree (m, n) as

$$T(m,n) = k + \frac{N-m}{mn} \text{ (base cycles)}$$

Thus the speedup over the base machine is

$$S(m,n) = \frac{T(1,1)}{T(m,n)} = \frac{k+N-1}{k+(N-m)/(mn)} = \frac{mn(k+N-1)}{mnk+N-m}$$

This speedup limit $S(m,n) \to mn$ as $N \to \infty$, as expected.

Example: The DEC 21064 superpipelined superscalar architecture

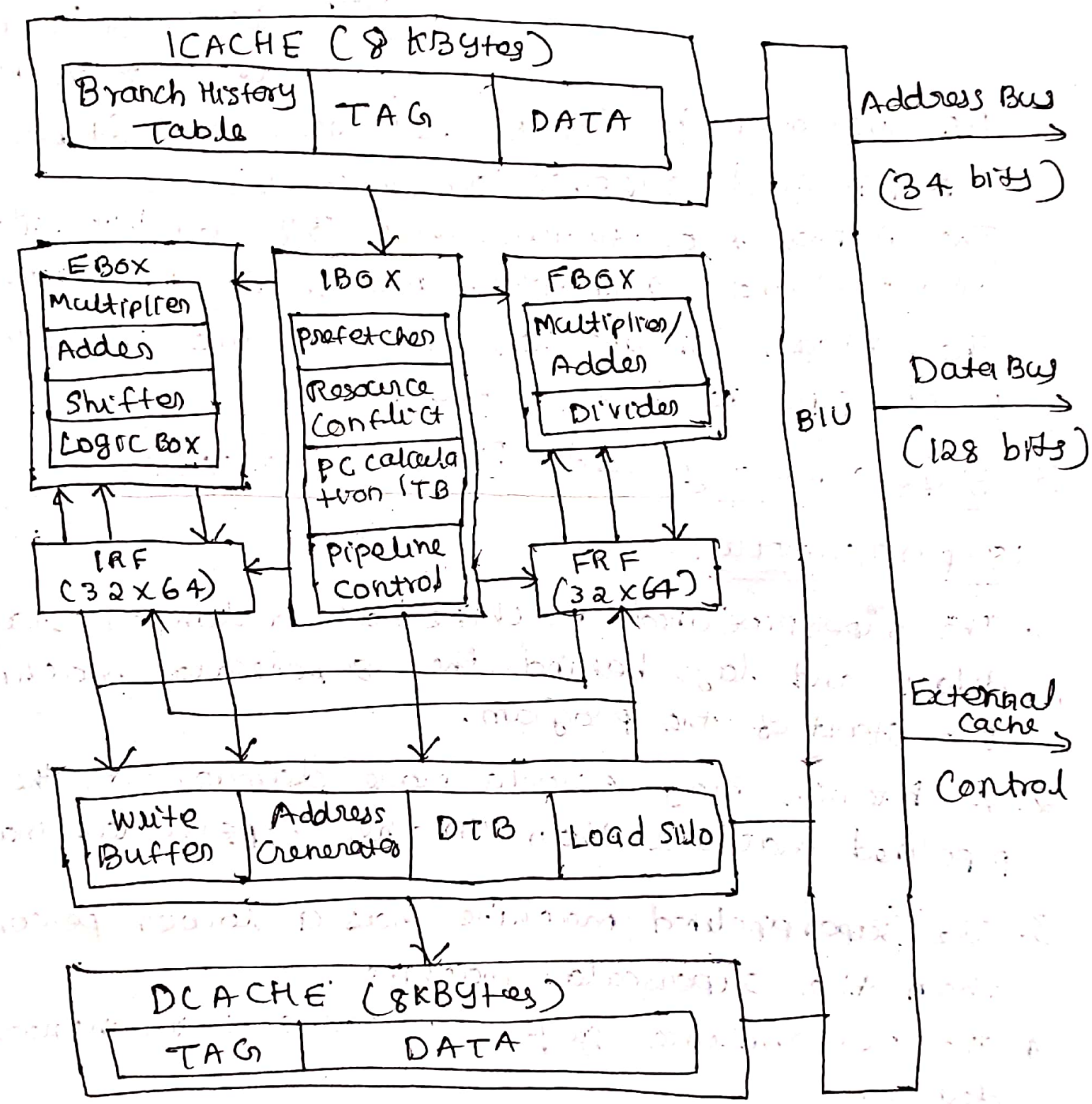


Figure 5-21 The architecture of the DEC 21064-A microprocessor.

EBOX = Integer Unit
FBOX = Floating-point unit
ABOX = Address Unit
IBOX = Central control

BIU = Bus interface unit
IRF = Integer register file
FRF = Floating-point register file
DTB = Data-stream translation buffer

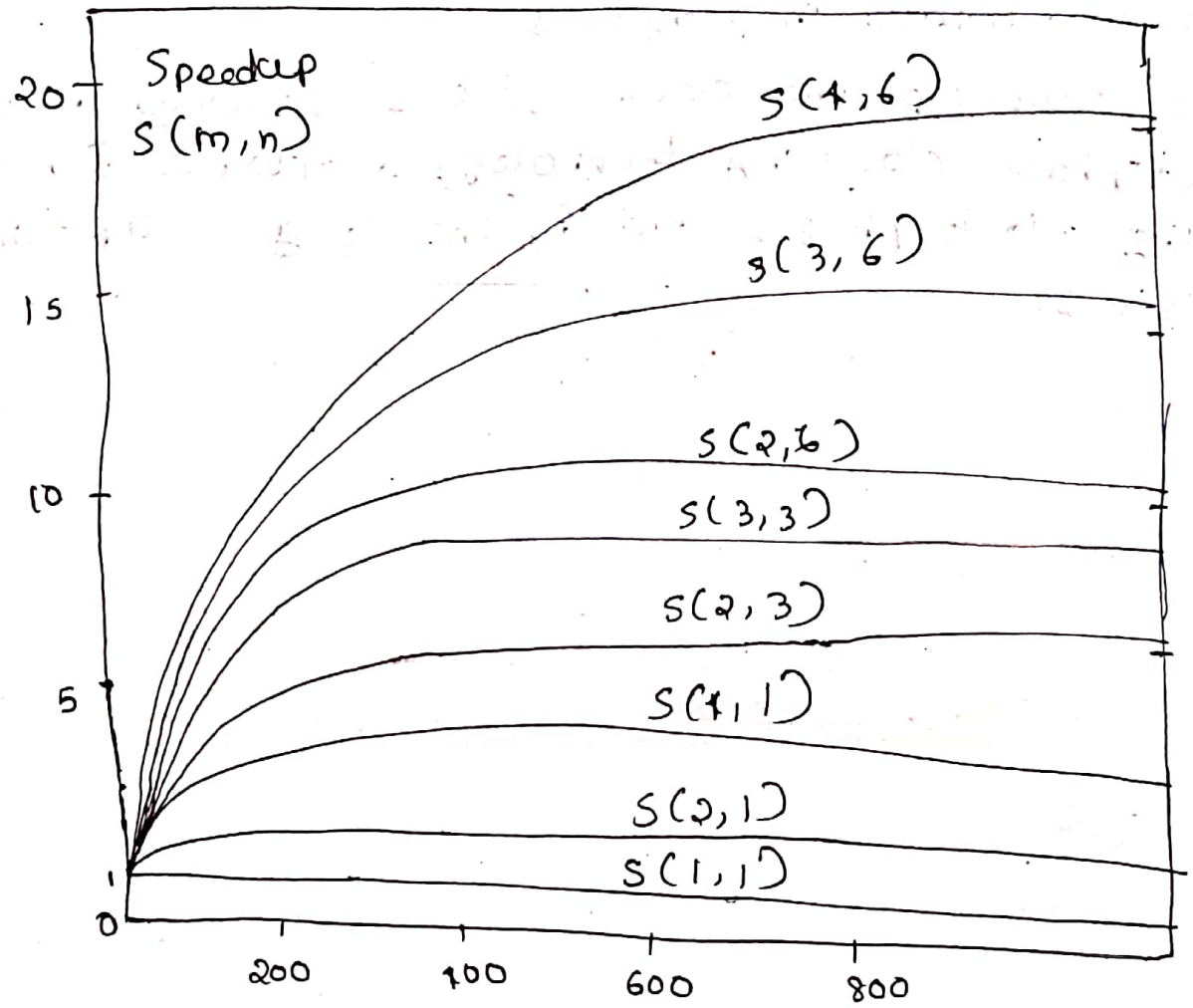
As illustrated in Figure 5.21, this is a 64-bit superpipelined superscalar processor.
The Alpha architecture has 32 64-bit integer registers and 32 64-bit floating-point registers.
The integer pipeline has 7 stages, and the floating-point pipeline has 10 stages.

3. Superysymmetry and Design Tradeoffs :

Superysymmetry.

1. The superpipelined machine has a longer startup delay and lags behind the superscalar machine at the start of the program.
2. A branch may create more damage on the superpipelined machine than on the superscalar machine.
3. The superpipelined machine has a lower performance than the superscalar machine.
4. The performance gap decreases with increasing degree.
5. The extra startup overhead and higher branching damages reported.

- Design Tradeoffs



In Figure 5.22 the speedup curve $S(m,n)$ in equation

$$S(m,n) = \frac{T(1,1)}{T(m,n)} = \frac{mn(k+n-1)}{mnk+n-m}$$

for $k=8$ Pipeline stages.

Four cases — single-issue, dual-issue, triple-issue, and quad-issue machines.

As the degree of superpipelining increases, the speedup curves show steady increases.

Two important limitations:

1. The superscalar degree m is limited by the small ILP encountered in programs.
2. The superpipeline clock cycle is limited by the multiphase clocking technology available for distributing clock phases and by the long setup time of registers.

Module VI

Multithreaded and data flow Architectures

Large-scale computers built with scalable, multi-threaded, or dataflow architectures are candidates for developing massively parallel processing (MPP) systems.

LATENCY-HIDING TECHNIQUES

1. The processor speed is increasing at a much faster rate than memory and interconnection networks.
2. Thus any scalable multiprocessor or large-scale multicomputer must rely on the use of latency-reducing, latency tolerating, or latency hiding mechanisms.
3. Latency hiding can be accomplished through
   i. Using prefetching techniques
   ii. using coherent caches
   iii. using relaxed memory consistency models
   iv. using multiple-contexts

a) Shared virtual memory
Single-address-space multiprocessors / multicomputers must use shared virtual memory.
i. The Architecture Environment
ii. The SVM Concept
iii. Page swapping

i. The Architecture Environment

1. The Dash architecture is a large-scale, cache-coherent, NUMA multiprocessor system, as depicted in Figure 6.1.
2. It consists of multiple multiprocessor clusters connected through scalable, low-latency interconnection network.

3. Physical memory is distributed among the processing nodes in various clusters.
4. The distributed memory forms a global address space.

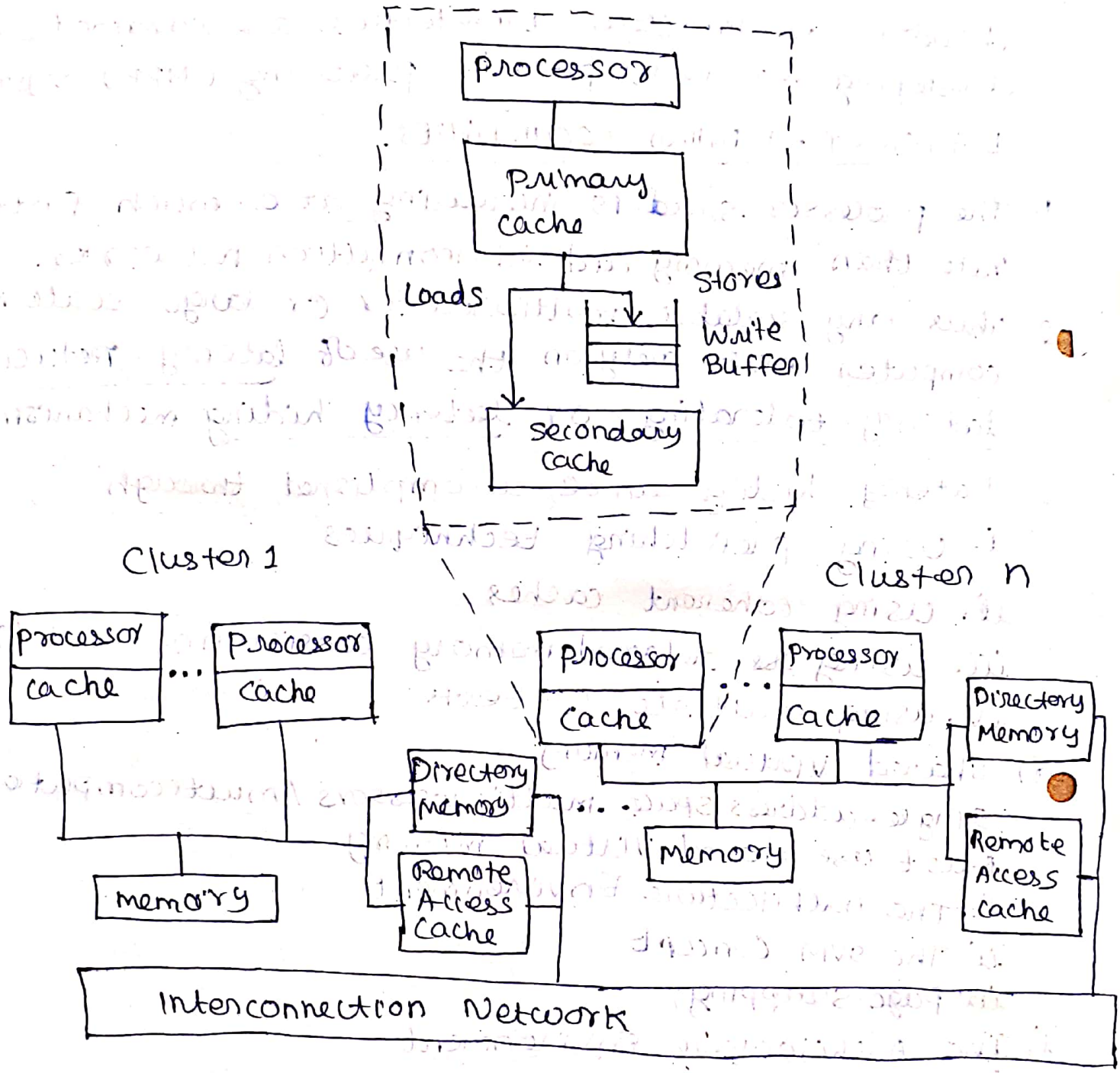


Figure 6.1 A scalable coherent cache multiprocessor with distributed shared memory modeled after the Stanford Dash.

5. Directory Memory
For each memory block, the directory keeps track of remote nodes caching it.

6. Local Cache
Two levels of local cache are used per processing node. i. one can assume a write-through primary cache and ii. a write-back secondary cache.

7. Write Buffer
Loads and writes can be seperated with the use of write buffers for implementing weaker memory consistency models.

8. Main memory.
The main memory is shared by all processing nodes in the same clusters.

9. Directory memory and Remote Access Cache
To facilitate prefetching and the directory-based coherence protocol, directory memory and remote-access caches are used for each cluster.

ii. The SVM Concept (Shared Virtual Memory)

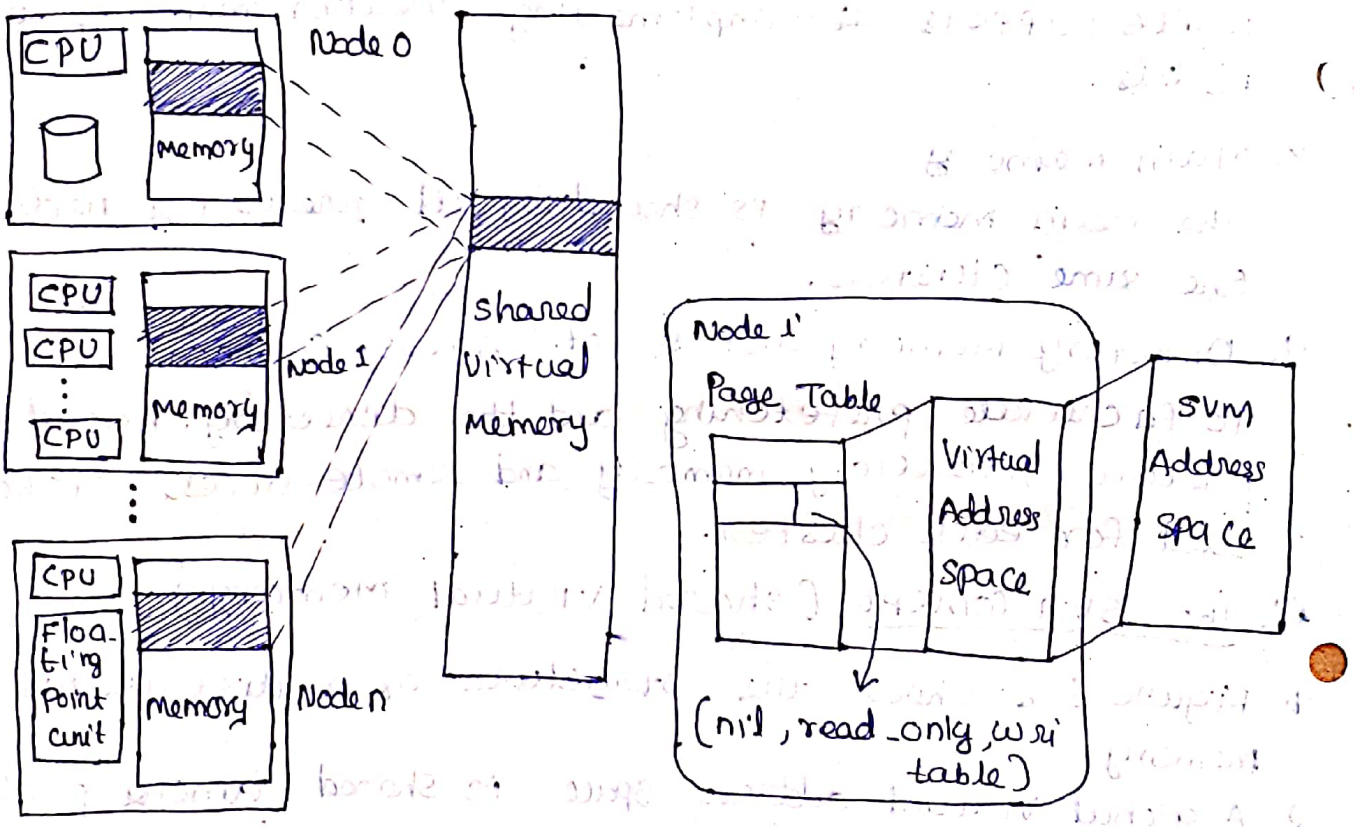
1. Figure 6.2 shows the structure of a distributed shared memory.
2. A global virtual address space is shared among processors residing at a large number of loosely coupled processing nodes.
3. Implementation and management issues of SVM are
a) First developed in a Ph.D thesis by Li (1986) at Yale University.
b) The idea is to implement coherent shared memory on a network of processors without physically shared memory.
c) The coherent mapping of SVM on a message-passing multicomputer architecture is shown in Figure 6.2b.
d) The system uses virtual addresses instead of physical addresses

for memory references.

d) Each virtual address space can be as large as a single node can provide and is shared by all nodes in the system.

e) First system - IVY implemented on a network of Apollo workstations by Li (1988).

f) A memory-mapping manager on each node views its local memory as a large cache of pages for its associated processor.



(a) Distributed shared memory (b) Shared virtual memory mapping

Figure 6.2 The concept of distributed shared memory with a global virtual address space shared among all processors on loosely coupled processing nodes in a massively parallel architecture.

iii. Page Swapping

1. When a processor writes a page that is also on other processors, it must update the page and then invalidate all copies on the other processors. Li described the page swapping as follows:
2. A memory reference causes a page fault when the page containing the memory location is not in a processor's local memory.
3. When a page fault occurs, the memory manager retrieves the missing page from the memory of another processor.
4. If there is a free page frame available on the receiving node, the page is moved in.
5. Otherwise, the SVM system uses page replacement policies to find an available page frame, swapping its contents to the sending node.
6. A hardware MMU can set the access rights (nil, read-only, writable) so that a memory access violating memory coherence will cause a page fault.

b) Prefetching Techniques.

i. Prefetching Techniques
ii. Benefits of prefetching
iii. Benchmark Results

i. Prefetching Techniques

1. Prefetching uses knowledge about the expected misses in a program to move the corresponding data close to the processor before it is actually needed.
2. Classification of prefetching:

a) binding prefetching,
b) nonbinding prefetching,
c) hardware controlled prefetching,
d) software controlled prefetching.

a) Binding prefetching:
The value of a later reference (e.g., a register load) is bound at the time when the prefetch completes. Since the value will become stale if another processor modifies the same location during the interval between prefetch and reference.

b) Nonbinding prefetching:
Brings the data close to the processor, but the data remains visible to the cache coherence protocol and is thus kept consistent until the processor actually reads the value.

c) Hardware controlled prefetching:
Long cache lines and instruction lookahead.
The effectiveness of long cache lines is limited by the reduced spatial locality in multiprocessor applications while instruction lookahead is limited by branches and the finite lookahead buffer size.

d) Software-controlled prefetching:
Explicit prefetch instructions are issued.
Software control allows the prefetching to be done selectively (reducing bandwidth requirements) and extends the possible interval between prefetch issue and actual reference, which is very important when latencies are large.

ii. Benefits of Prefetching

1. When a prefetch is issued early enough in the code

so that the line is already in the cache by the time it is referenced.

2. When the address of a data structure cannot be determined until immediately before it is referenced.
3. If multiple prefetches are issued back to back to fetch the data structure, the latency of all but the first prefetched references can be hidden due to the pipelining of the memory accesses.
4. Use an ownership-based cache coherence protocol.
5. Reduce the write latencies and the ensuing network traffic for obtaining ownership.
6. Network traffic is reduced in read-modify-write instructions, since prefetching with ownership avoids first fetching a read-shared copy.

iii. Benchmark Results

Stanford researchers (Gupta, Hennessy, Gharachorloo, Mowry, and Weber, 1991) have reported some benchmark results for evaluating various latency-hiding mechanisms. Bench-marks are:

1. A particle-based three-dimensional simulator used in aeronautics (MP3D),
2. an LU-decomposition program (LU), and
3. a digital logic simulation program (PTHOR).

c) Distributed Coherent Caches

Coherence problem is much more complicated for large-scale multiprocessors that use general interconnection networks. So

i. some existing large-scale multiprocessors do not provide caches (eg, BBN Butterfly),
ii. others provide caches that must be kept coherent by software (eg., IBM RP3), and

iii. Still others provide full hardware support for coherent caches (eg., Stanford Dash).

a Dash Experience
b Benefits of caching

a Dash Experience

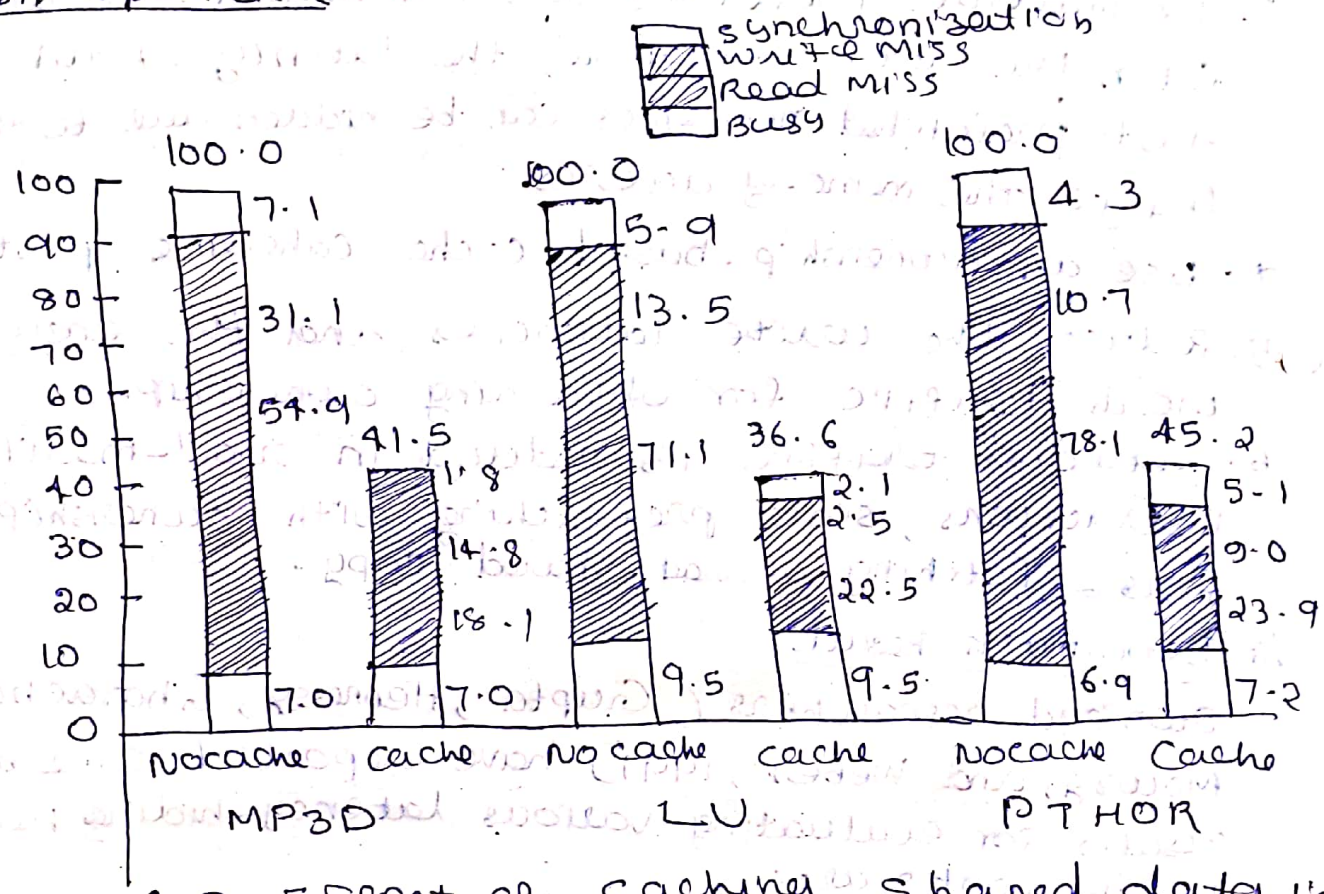


Figure 6.3 Effect of caching shared data in simulated Dash benchmark experiments.

Figure 6.3 presents a breakdown of the normalized execution times with and without caching of shared data for each of the applications. Private data are also cached in both caches.

1. The bottom section of each bar represents the busy time or useful cycles executed by the processor.
2. The section above it represents the time that the processor is stalled waiting for reads.
3. The section above that is the amount of time the processor is stalled waiting for writes to be completed.

4. The top section, labeled "synchronization", accounts for the processor is stalled due to locks and barriers.

b Benifits of Caching

1- The largest benefit comes from a reduction in the number of cycles wasted due to read misses
2. The cycles wasted due to write misses are also reduced.
3. Cache-hit ratios achied by

MP3D	LU	PTHOR	
80%	66%	77%	for shared-read references and
75%	97%	47%	for shared-write references.

d. Scalable Coherence Interface (SCI)

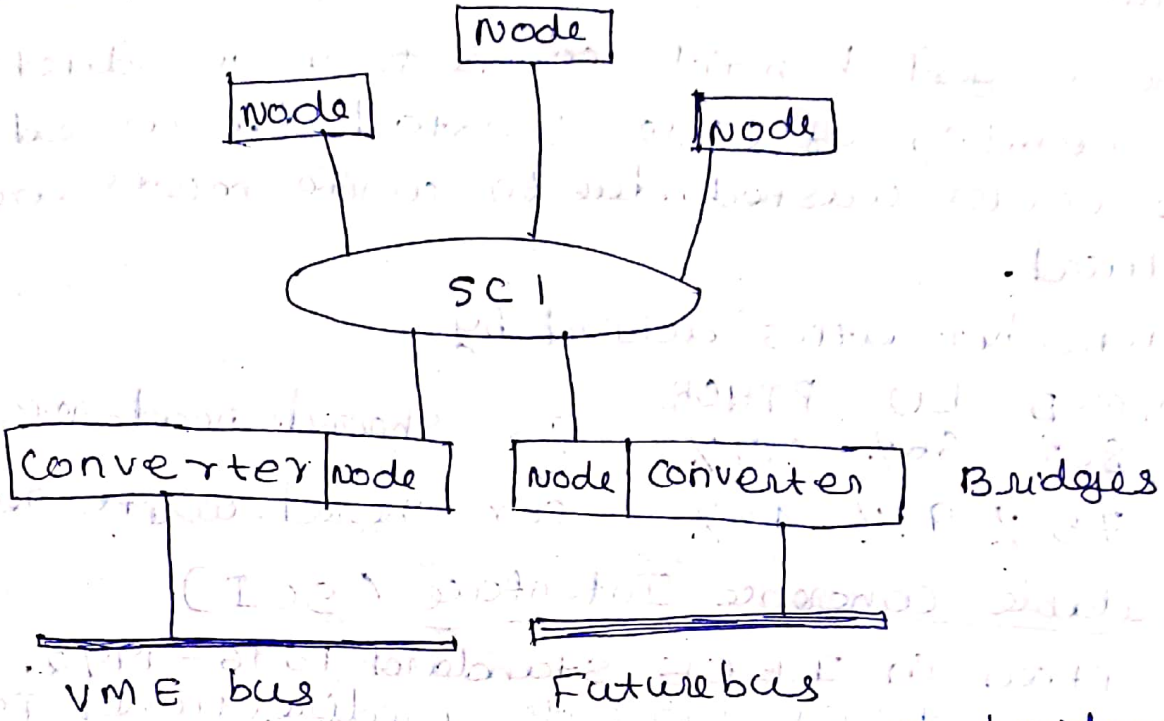
1. specified in IEEE standard 1596-1992.
2. The current SCI supports unidirectional point-to-point connections.
3. the cache coherence protocols ~~are~~ used in the SCI are directory-based.

i. SCI Interconnect models
ii. Sharing-List structures
iii. Sharing-List creation.
iv. Sharing-list Updates
v. Implementation Issues.

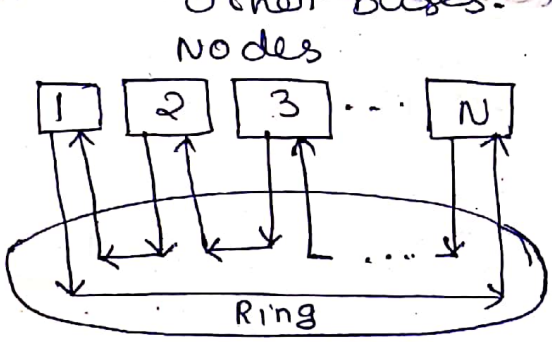
i. SCI Interconnect models

1. The SCI defines the interface between nodes and the external interconnect.
2. A typical SCI configuration is shown in Figure 6.4a.
3. Each SCI node can be a processor with attached memory and I/O devices.

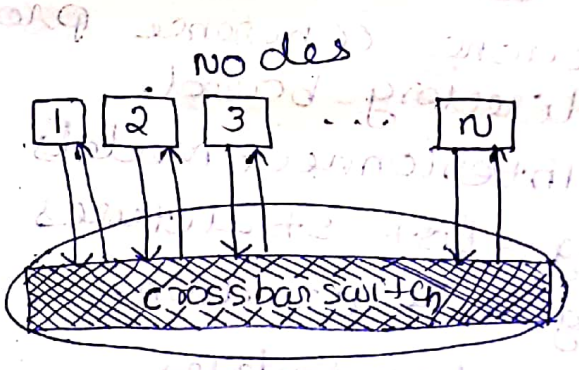
A. The SCI Interconnect can assume a ring structure or a crossbar switch as depicted in Figures 6.4b and 6.4c.



(a) Typical SCI configuration with bridges to other buses.



(b) A ring for point-to-point transactions



(c) A crossbar multiprocessor

Figure 6.4 SCI interconnection configurations.

5. The converter in Figure 6.4a are used to bridge the SCI ring to the VME or Future bus.
6. Each node has an input link and an output link which are connected from or to the SCI ring or crossbar.

Advantages:

1. The bandwidth, arbitration, and addressing mechanisms of a SCI ring are expected to significantly outperform backplane buses.
2. By eliminating the snoopy cache controllers, the SCI should also be cheaper.
3. It is scalable.

Disadvantages:

1. The performance of the SCI protocol does not scale.
2. When the sharing list is long, invalidations will take too much time.

ii) Sharing-List structures

1. Sharing lists are used in the SCI to build chained directories for cache coherence use.
2. Sharing lists length is unbounded.
3. Sharing lists are dynamically created, pruned, and destroyed.
4. By distributing the directories among the sharing processors, the SCI avoids scaling limitations imposed by using a central directory.
5. communications among sharing processors are supported by heavily shared memory controllers, as shown in Figure 6.5.
6. For every block address, the memory and cache entries have additional tag bits which are used to identify the first processor (head) in the sharing list and to link the previous and following nodes.
7. Doubly linked lists are maintained between processors in the sharing list, with forward and backward

pointers as shown by the double arrows in each link.

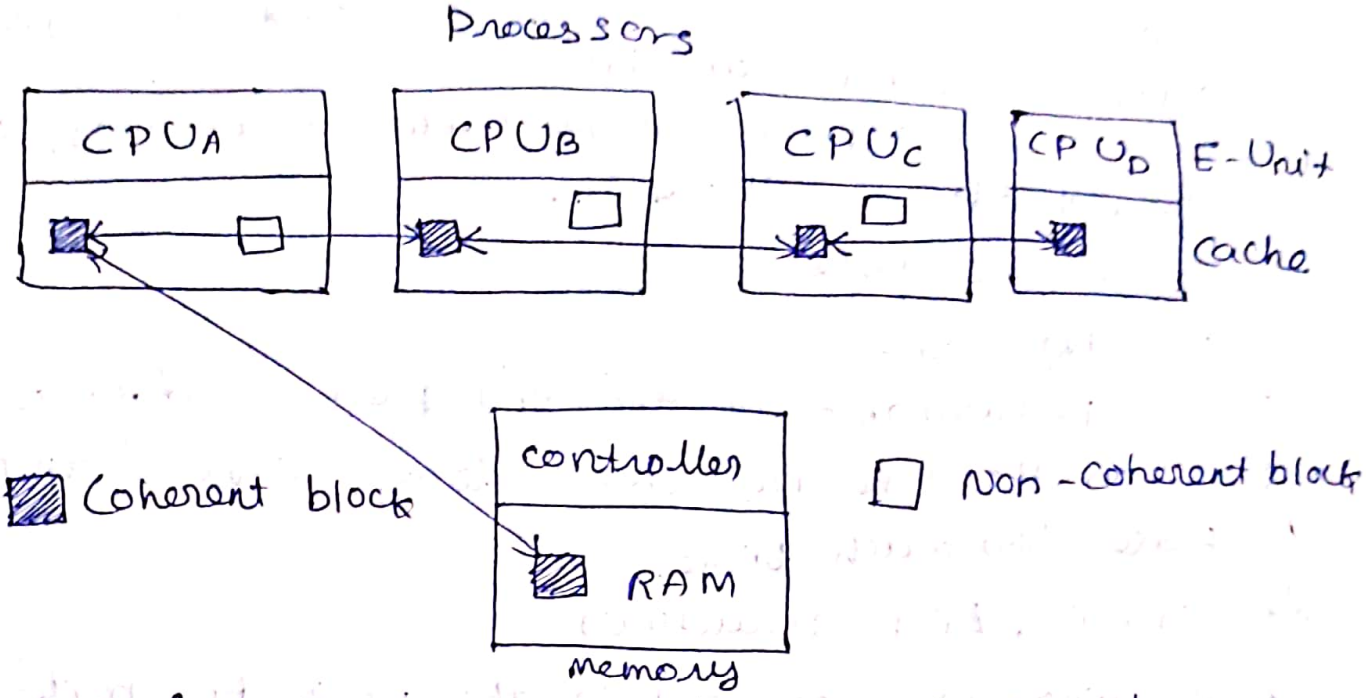
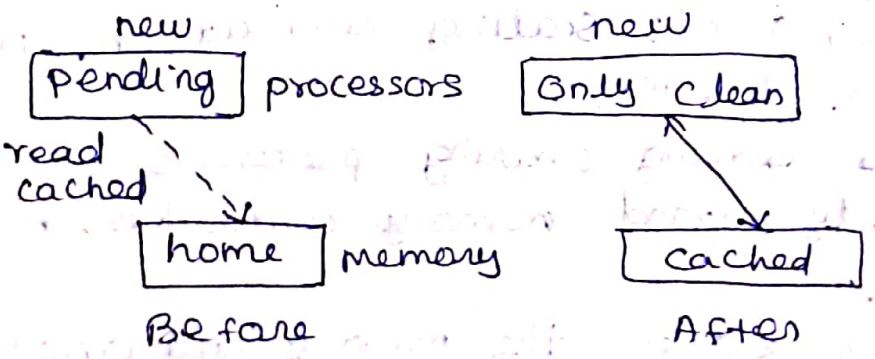


Figure 6.5 SCI cache coherence protocol with distributed directories.

iii. Sharing-List Creation

sharing-list creation is illustrated in Figure 6.6a.



6.6 (a) Creation of Sharing list.

1- The states of the sharing list are defined by the state of the memory and the states of list entries, such as clean, dirty, valid or stale

2. The head processor is always responsible for list management.
3. The stable and legal combinations of the memory and entry states can specify uncached data, clean or dirty data at various locations, and cached writable or stale data.
4. The memory is initially in the home state (uncached) and all cache copies are invalid.
5. Sharing-list creation begins at the cache where an entry is changed from an invalid to a pending state.
6. When a read-cache transaction is directed from a processor to the memory controller, the memory state is changed from uncached to cached and the requested data is returned.

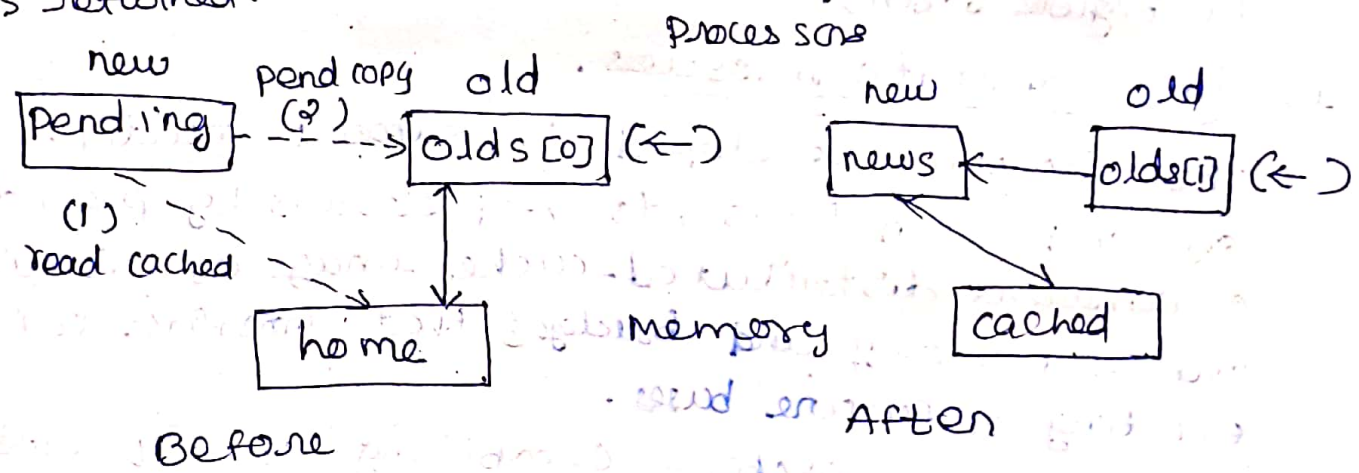


Figure 6.6(b) Addition of new nodes

Figure 6.6 Sharing-list creation and update examples.

iii Sharing-List Updates

1. For subsequent memory access, the memory state is cached, and the cache head of the sharing list has possibly dirty data.
2. As in Figure 6.6(b), a new requester (cache A)

first directs its read-cache transaction to memory but receives a pointer to cache B instead of the requested data.

3. A second cache-to-cache transaction, called prepend, is directed from cache A to cache B.
4. Cache B then sets its backward pointer to point to cache A and returns the requested data.
5. The dashed lines correspond to transactions between a processor and memory or another processor.
6. The solid lines are sharing-list pointers.
7. After the transaction, the inserted cache A becomes the new head, and the old head, cache B, is in the middle as shown by the new sharing list on the right in Figure 6.6(b).

iv. Implementation Issues.

1. The SCI was developed to support multiprocessor systems with thousands of processors by providing a coherent distributed-cache image of distributed shared memory and bridges that interface with existing or future buses.
2. Combining switches - combining of requests to overcome memory hot spots.
3. Differential emitter coupled logic (ECL) signaling. (250-MHz clock rate)
4. SCI fiber optic implementation useful for moving data out of detectors.

c) Relaxed Memory Consistency

Memory models for building scalable multiprocessors with distributed shared memory.

i. processor consistency
ii. Release consistency

i. processor consistency.

1. Introduced by Goodman (1989)
2. Writes issued by each individual processor are always in program order.
3. The order of writes from two different processors can be out of program order.
4. Two conditions related to other processors are required for ensuring Processor Consistency:
   (1) Before a read is allowed to perform with respect to any other processor, all previous read accesses must be performed.
   (2) Before a write is allowed to perform with respect to any other processor, all previous read or write accesses must be performed.

These conditions allow reads following a write to bypass the write.

ii. Release Consistency.

1. Introduced by Gharachorloo et al. (1990).
2. Release consistency requires that synchronization accesses in the program be identified and classified as either acquires (eg., locks) or releases (eg., unlocks).
3. An acquire is a read operation (which can be part of a read-modify-write) that gains permission to

access a set of data, while a release is a write operation that gives away such permission.

4. This facility used for synchronization.

5. The main advantage of the relaxed models is the potential for increased performance by hiding as much write latency as possible.

6. The main disadvantage is increased hardware complexity and a more complex programming model.

Release consistency.

1. Three conditions ensure release consistency:
   i) Before an ordinary read or write access is allowed to perform with respect to any other processer, all previous acquire accesses must be performed.
   ii) Before a release access is allowed to perform with respect to any other processor, all previous ordinary read and store accesses must be performed.
   iii) special accesses are processor-consistent with one another. The ordering restrictions imposed by weak consistency are not present in release consistency. Instead, release consistency requires processor consistency and not sequential consistency.

2. Release consistency can be satisfied by
   (i) stalling the processor on an acquire access until it completes, and
   ii) delaying the completion of release access until all previous memory accesses complete.

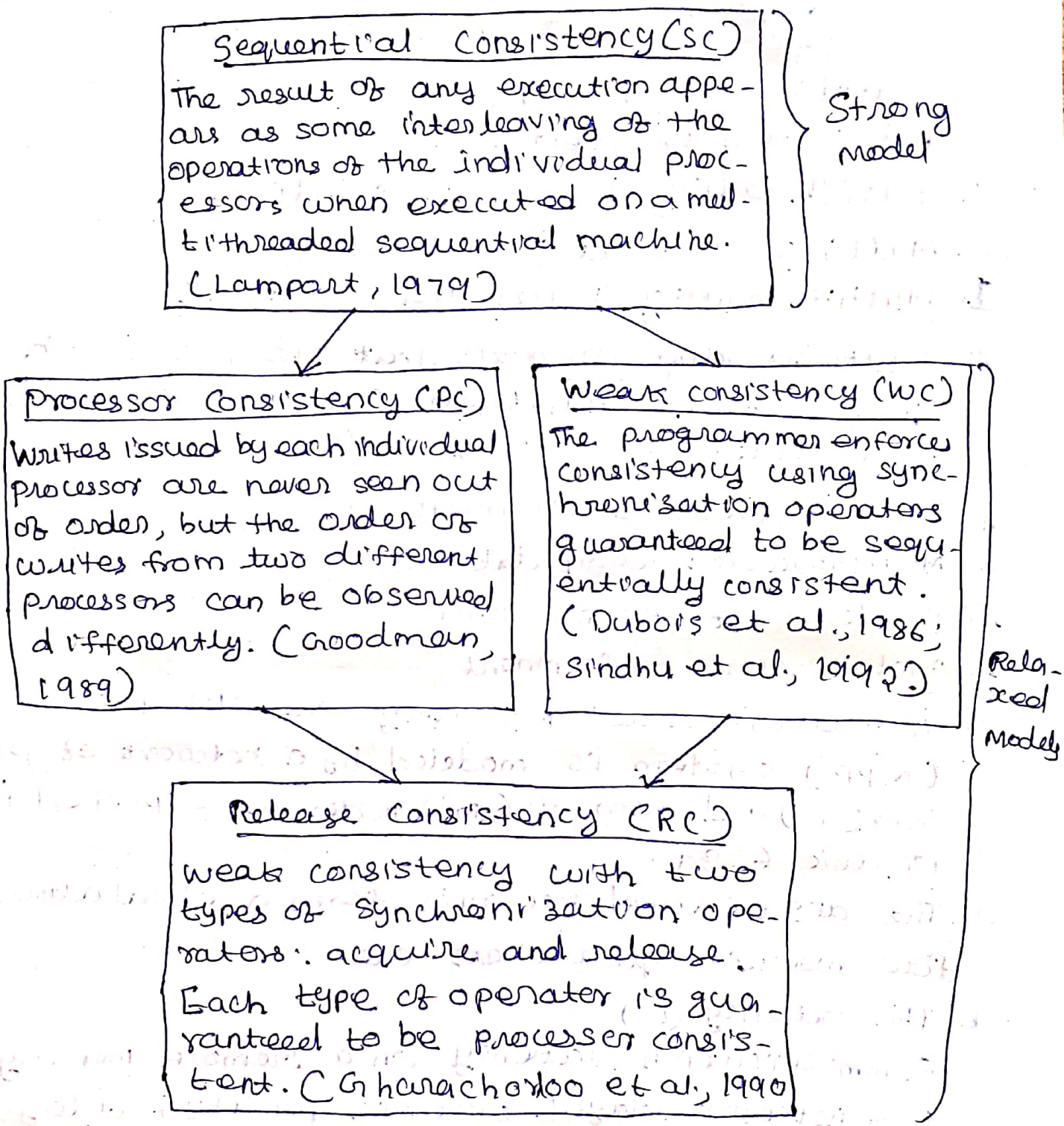


Figure 6.7 Intuitive definitions of four memory consistency models. The arrows point from strong to relaxed consistencies.

PRINCIPLES OF MULTITHREADING

Multidimensional and / or multithreaded processors and system architectures.

1. Multithreading Issues and solutions
2. Multiple - context processors

1. Multithreading Issues and Solutions

* Multithreading demands that the processor be designed to handle multiple contexts simultaneously on a context-switching basis.

i. Architecture Environment
ii. Multithreaded Computations
iii. Problems of Asynchrony

i. Architecture Environment

1. A multithreaded massively parallel processing (MPP) system is modeled by a network of processor (P) and memory (M) nodes as depicted in Figure 6.8a.
2. The distributed memories form a global address space. Four machine parameters are

i. The latency (L)
Communication latency on a remote memory access.
L = network delays + cache-miss penalty + delays caused by contentions in split transactions.

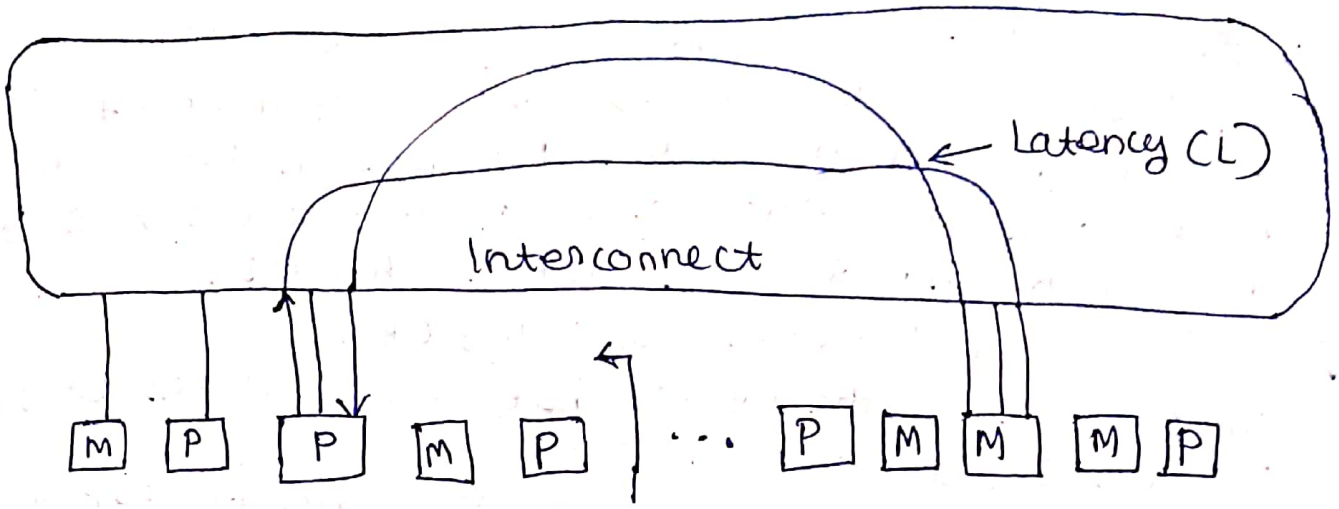
ii. The number of threads (N)
Number of threads that can be interleaved in each processor context of
Thread ⇒ (program counter, a register set, and the required context status words).

iii. The context-switching overhead (C).
This refers to the cycles lost in performing context switching in a processor. Depends on the switch mechanism and the amount of processor states devoted to maintaining active threads.

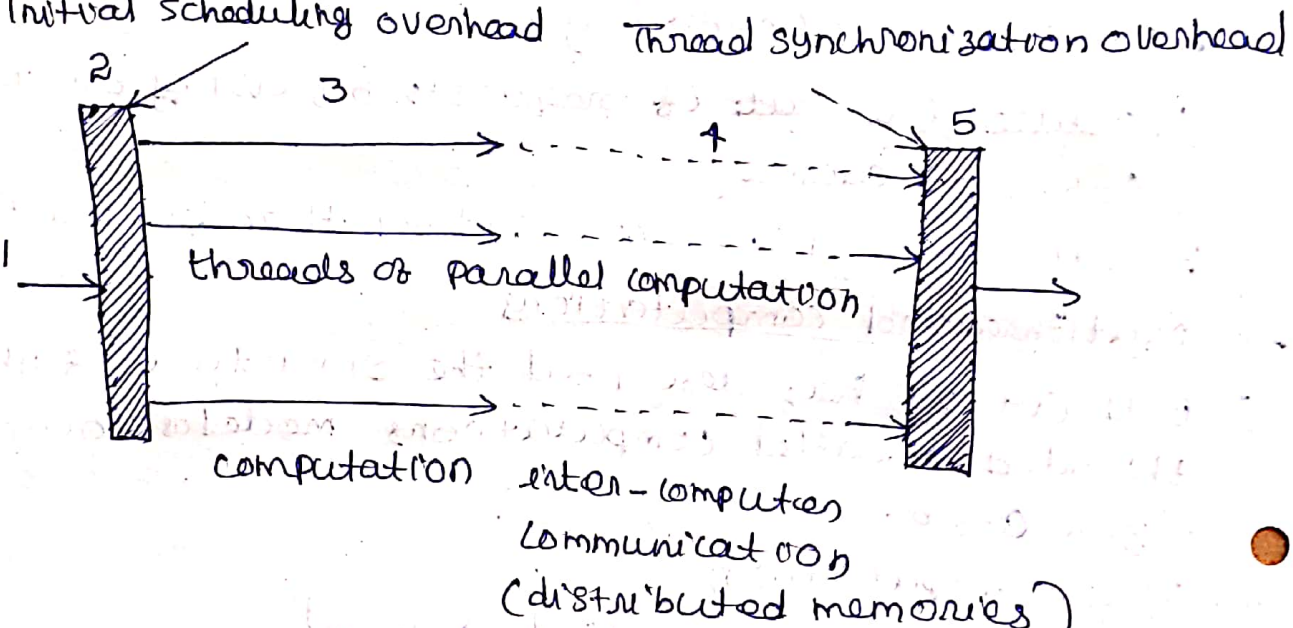
iv. The interval between switches (R).
This refers to the cycles between switches triggered by remote reference.
$p = 1/R$ is called the rate of requests for remote accesses.

To increase efficiency:
1. Reduce the rate of requests by using distributed coherent caches.
2. Eliminate processor waiting through multithreading.

d) Multithreaded computations
1. Bell (1992) has described the structure of the multithreaded parallel computations model as shown in Figure 6-8 b.
2. The computation
(1) starts a sequential thread,
(2) followed by supervisory scheduling
(3) begin threads of computation
(4) by intercomputer messages that update variables among the nodes when the computer has a distributed memory, and finally
(5) by synchronization prior to beginning the next unit of parallel work.



(a) The architecture environment.



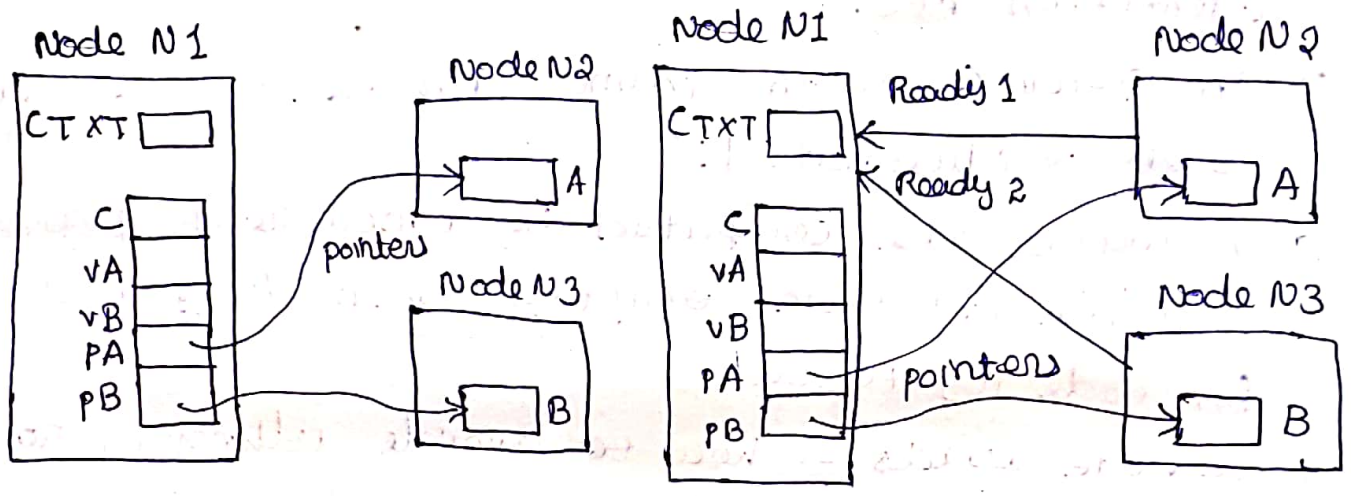
(b) Multithreaded computation model.

Figure 6-8 Multithreaded architecture and its computation model for a massively parallel processing system.

3. Multithreading is not capable of speedup in the execution of single threads, while weak ordering or relaxed consistency models are capable of doing this.

iii. Problems of Asynchrony

1. Massively parallel processors operate asynchronously in a network environment.
2. The asynchrony triggers two fundamental latency problems:
   i. remote loads and
   ii. synchronizing loads



ON Node N1, compute
C = A − B demands to execute:
vA = rload pA
vB = rload pB } ("remote" loads)
C = vA − vB

(a) The remote loads problem

On node N1, compute: C = A − B
A and B computed concurrently
Thread on N1 must be notified when A, B are ready.

(b) The synchronizing loads problem

Figure 6.9 Two common problems caused by asynchrony and communication latency in massively parallel processors.

3. The remote load situation is illustrated in figure 6.9a.
4. Variables A and B are located on nodes N2 and

N3 respectively. They need to be brought to node N1 to compute the difference A−B in variable C.

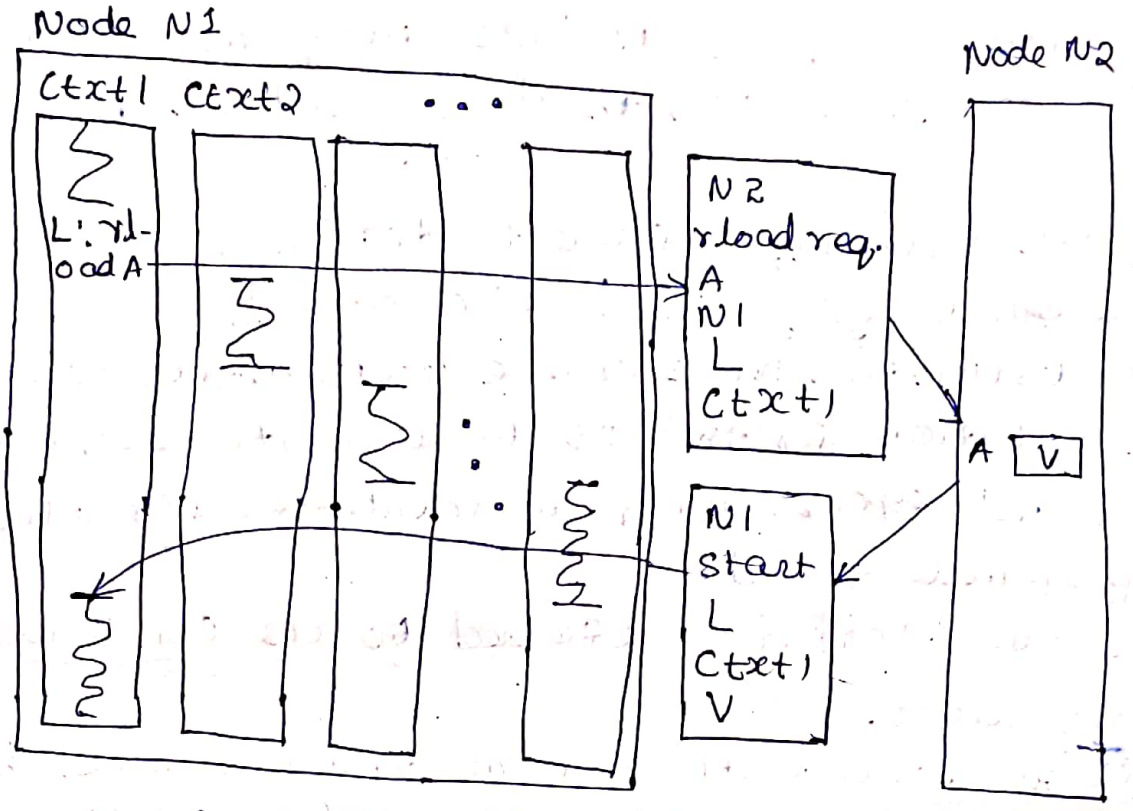
5. Let PA and PB be the pointers to A and B, respectively.
6. The context of the computation on N1 is represented by the variable CTXT. It can be a stack pointer, a frame pointer, a current-object pointer, a process identifier, etc.
7. In Figure 6.9b, the idling due to synchronizing loads is illustrated.
8. A and B are computed by concurrent processes, and we are not sure exactly when they will be ready for node N1 to read.
9. Remote loads :- how to avoide idling in node N1
   - latency caused - architectural
   - Predictable.

   Synchronizing load - depends on scheduling and the time it takes to compute A and B.
   - latency is unpredictable

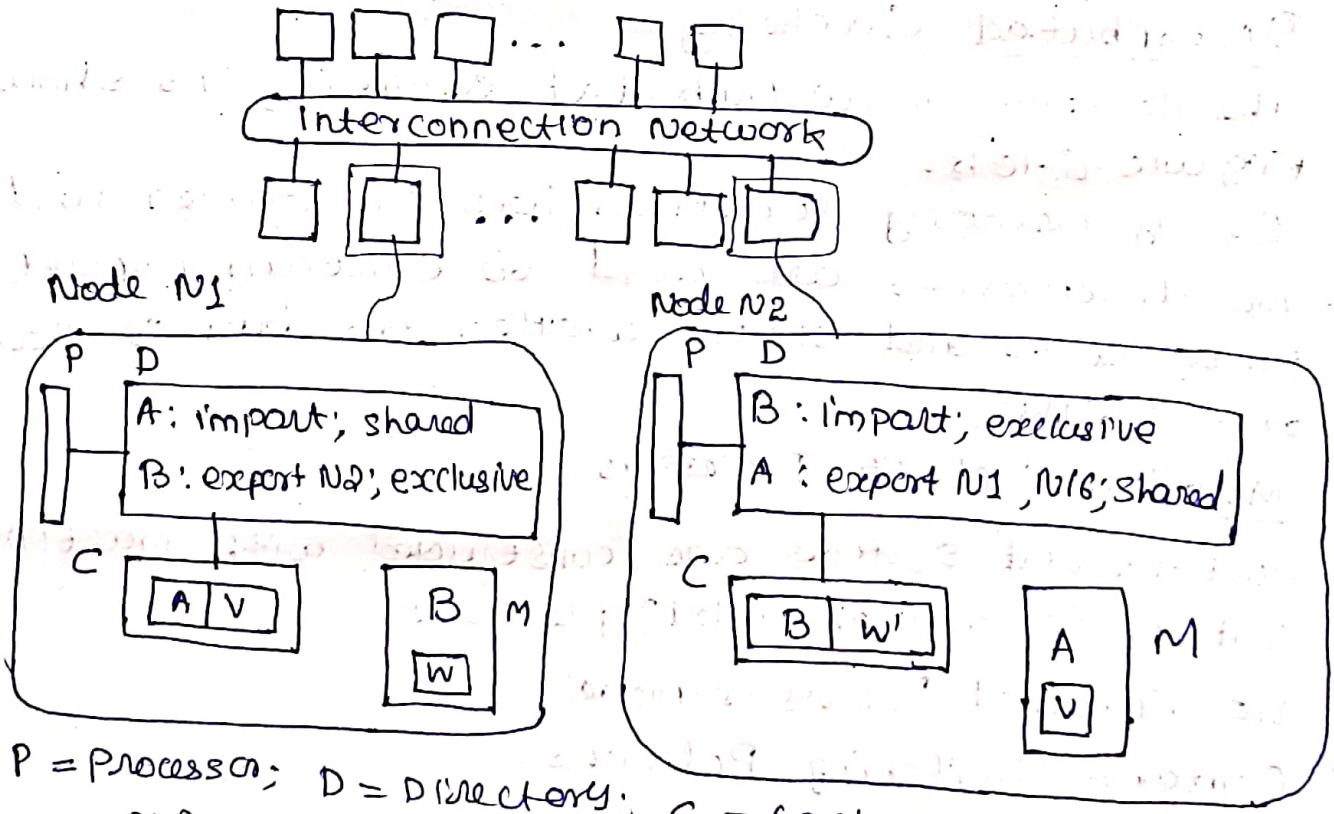
Multithreading solutions

1. Multiplex among many threads.
2. When one thread issues a remote-load request, the processor begins work on another thread, and so on (Figure 6.10a).
3. The cost of thread switching should be much smaller than that of the latency of the remote load.
4. After issuing a remote load from thread T_1 (Figure 6.10a), switch to thread T_2, which also

issues a remote load.



(a) Multithreading solution



P = Processor; D = Directory; C = Cache; M = Memory

(b) Distributed caching

Figure 6.10 Two (23) solutions for overcoming

the asynchrony problems.

5. The responses from the threads may not return in the order in which they were created, because of -
   i. Requests traveling different distances,
   ii. through varying degree of congestion,
   iii. to destination nodes whose loads differ greatly etc.
   iv. One solution for this is to associate each remote load and response with an identifier for the appropriate thread.
   v. These identifiers are referred to as continuations on messages.
   vi. A large continuation name space should be provided to name an adequate number of threads waiting for remote responses.

Distributed Caching

1. The concept of distributed caching is shown in Figure 6.10b.
2. Every memory location has an owner node.
3. The directories are used to contain import-export lists and state whether the data is shared or exclusive.

2. Multiple-Context Processors

Multithreaded systems are constructed with multiple-context (or multithreaded) processors.

i. The Enhanced Processor Model
ii. Context-Switching Policies
iii. Processor Efficiencies.

i) The Enhanced Processor Model

1. A conventional single-thread processor will wait during a remote reference, so it is idle for a period of time L.
2. A multithreaded processor, as modeled in Figure 6.11, will suspend the current context and switch to another, so after some fixed number of cycles it will again be busy doing useful work, even though the remote reference is outstanding.

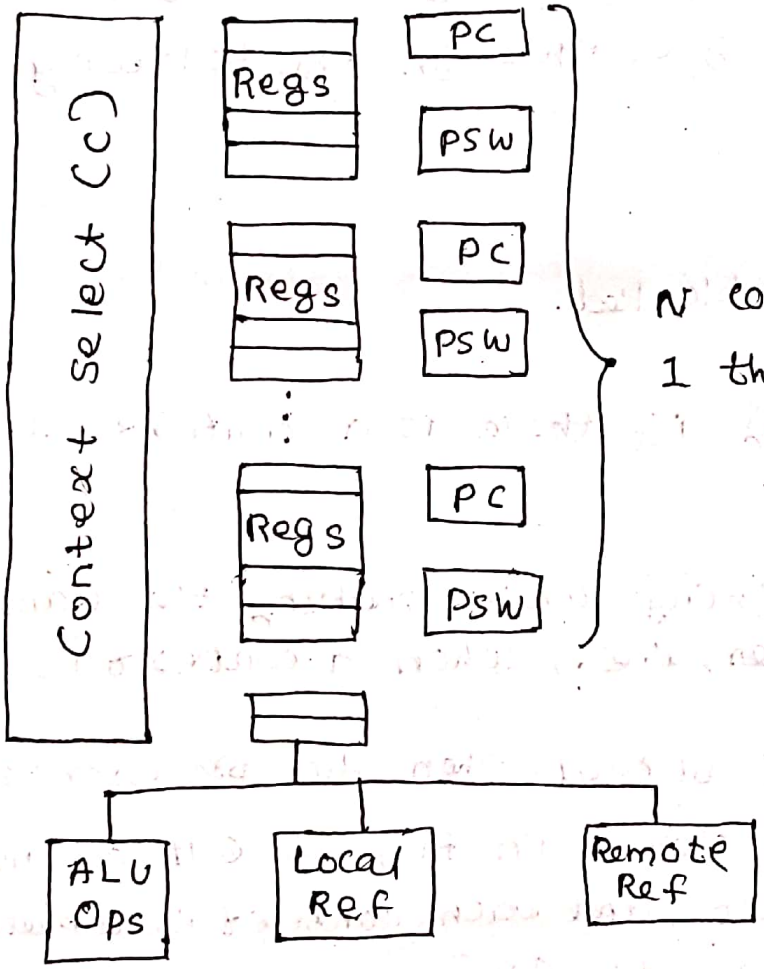


Figure 6.11 Multithreaded model.

3. The objective is to maximize the fraction of time that the processor is busy, efficiency of the processor is the performance index.

$$\text{Efficiency} = \frac{\text{busy}}{\text{busy} + \text{switching} + \text{idle}}$$

where busy, switching, and idle represent the amount of time, measured over some large interval, that the processor is in the corresponding state.

4. The basic idea behind a multithreaded machine is to interleave the execution of several contexts in order to dramatically reduce the value of idle, but without increasing the magnitude of switching.

5. A context cycles goes through the following states during its lifetime
   i. ready,
   ii. running,
   iii. leaving and iv. blocked.

6. Busy
   A processor is busy if there is a context in the running state.

7. Switching
   A processor is switching while making the transition one context to another, i.e., when a context is leaving.

8. Idle
   If all contexts are blocked then the processor is idle.

9. The abstract model shown in Figure 6.11 assumes one thread per context, and each context is represented by its own program counter (PC), register set, and process status word (PSW).

ii. Context-switching Policies

Different multithreaded architectures are distinguished by the context-switching policies adopted.

Four switching policies are:

1. Switch on cache miss

i. A context is preempted when it causes a cache miss.

ii. R is taken to be the average interval between misses (in cycles).

iii. L the time required to satisfy the miss.

iv. 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

i. Allows switching on every load, independent of whether it will cause a miss or not.

ii. R represents the average interval between loads.

iii. A general multithreading model assumes that a context is blocked for L cycles after every switch; but in the 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.

4. Switch on block of instruction

Blocks of instructions from different threads are interleaved. This will improve the cache-hit ratio due to locality. It will also benefit single-context performance.

iii. Processor Efficiencies.

1. A single-thread processor executes a context until a remote reference is issued (R cycles) and then is idle until the reference completes (L cycles).

The efficiency of a single-threaded machine is given by

$$E_1 = \frac{R}{R+L} = \frac{1}{1+L/R}$$

2. With multiple contexts, memory latency can be hidden by switching to a new context; but we assume that the switch takes C cycles of overhead.

The efficiency of a multiple context is analyzed under two different conditions.

i. Saturation region

The processor operates with maximum utilization. The cycle of the renewal process is R + C, and the efficiency is

$$E_{sat} = \frac{R}{R+C} = \frac{1}{1+C/R}$$

The efficiency in saturation is independent of the latency saturation is achieved when the time the processor spends servicing the other threads exceeds the time required to process a request, i.e., when $(N-1)(R+C) > L$.

N is the number of thread slots.

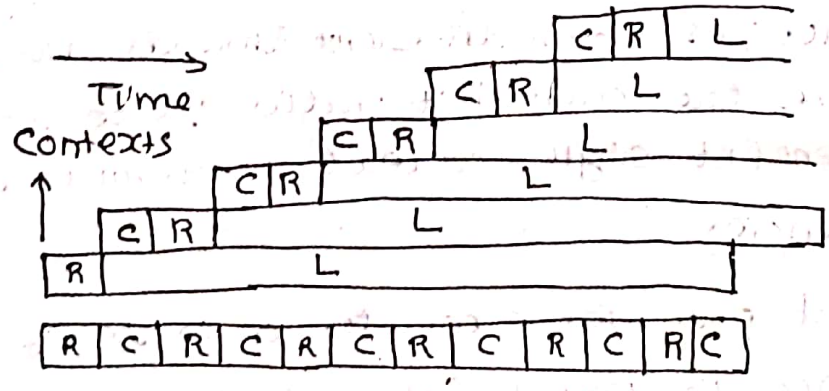


Figure 6.12 Snapshots of context switching in the saturation region.

ii. Linear region

1. When the number of contexts is below the saturation point, there may be no ready contexts after a context switch, so the processor will experience idle cycles.
2. The time required to switch to a ready context, execute it until a remote reference is issued, and process the reference is equal to $R+C+L$.
3. Assuming N is below the saturation point; during this time all the other contexts have a turn in the processor. Thus, the efficiency is given by

$$E_{lin} = \frac{NR}{R+C+L}$$

The efficiency increases linearly with the number of contexts until the saturation point is reached and beyond that remains constant.

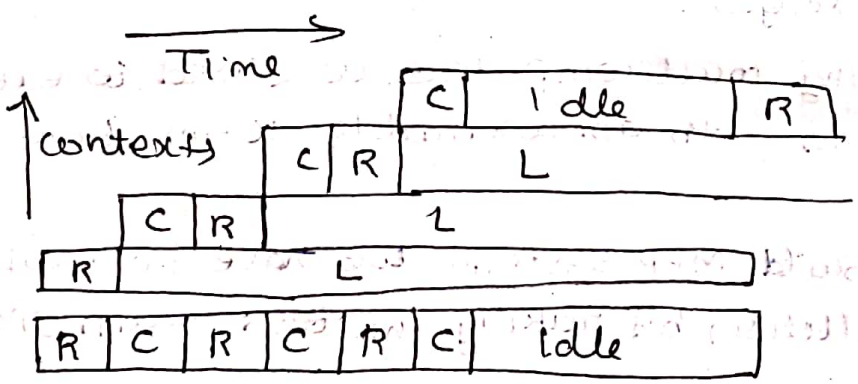


Figure 6.13 Snapshots of context switching in the linear region.

The processor efficiency is plotted as a function of the number of contexts in Figure 6.14.

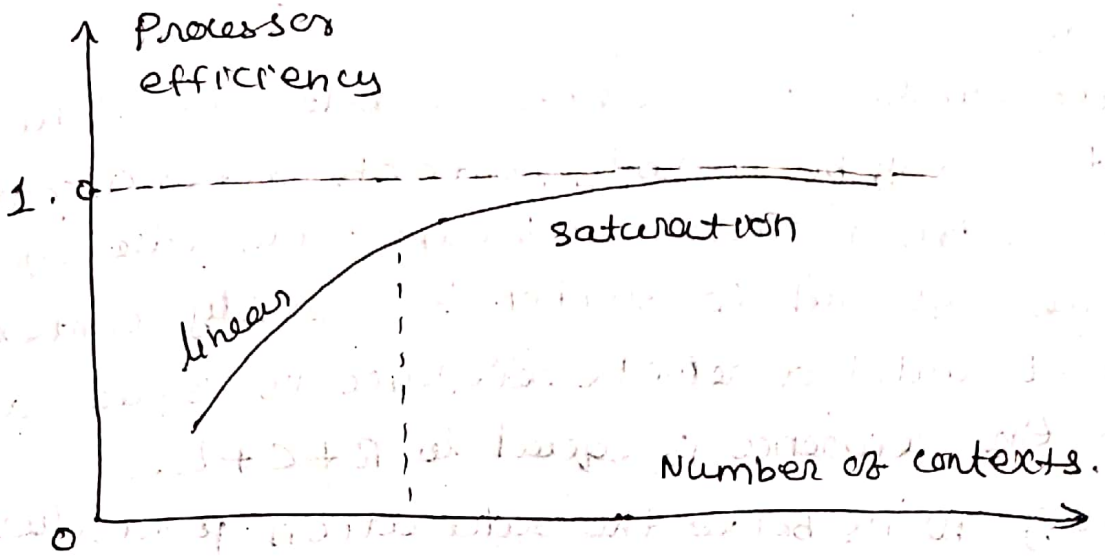


Figure 6.14 context switching and processor efficiency as a function of the number of contexts.

Fine-Grain Multicomputers

1. Shared-memory multiprocessors like the Cray Y-MP are used to perform coarse-grain computations in which each processor executes programs having a few tasks of 20s sec or longer.
2. Message-passing multicomputers are used to execute medium-grain programs with approximately 10-ms task size as in the iPSC/1.
3. In order to build MPP systems, we have to explore a higher degree of parallelism by making the task grain size even smaller.

Fine-Grain Parallelism

1. Latency Analysis
2. Fine-grain Parallelism

Latency Analysis

1. The communication latency T_c measures the data

Or message transfer time on a system interconnect.

2. The synchronization overhead T_s is the processing time required on a processor, or by a PE, or on a processing node of a multicomputer for the purpose of synchronization.

3. The sum $T_c + T_s$ gives the total time required for IPC.

4. The shared-memory Cray Y-MP has a short T_c but a long T_s.

5. The SIMD machine CM-2 has a short T_s but a long T_c.

6. The MIT J-Machine is designed to make a major improvement in both communication delays.

2. Fine-Grain Parallelism

1. The grain size T_g is measured by the execution time of a typical program, including both computing time and communication time involved.

2. Supercomputers handle large grain.

3. Both the CM-2 and the J-machine are designed as fine-grain machines.

4. Large grain implies lower concurrency or a lower DOP (degree of parallelism).

5. Fine grain leads to a much higher DOP and also to higher communication overhead.

6. Fine-grain multicomputers, like the J-Machine and Caltech Mosaic, are being designed to lower both the grain size and the communication overhead compared to those of traditional multicomputers.

The MIT J-Machine (based on the paper by Dally et al.)

1- The building block of MIT J-machine is the message-driven processor (MDP), which is a 36-bit microprocessor custom-designed for a fine-grain multicomputer.

i. The J-Machine Architecture
ii. The MDP Design
iii. Instruction-Set Architecture
iv. Communication Support, v. message Format and Routing
vi. The Router Design.
vii. Blocking Flow Control.
viii. Global naming
ix. ~~viii.~~ Synchronisation.

i- The J-Machine Architecture

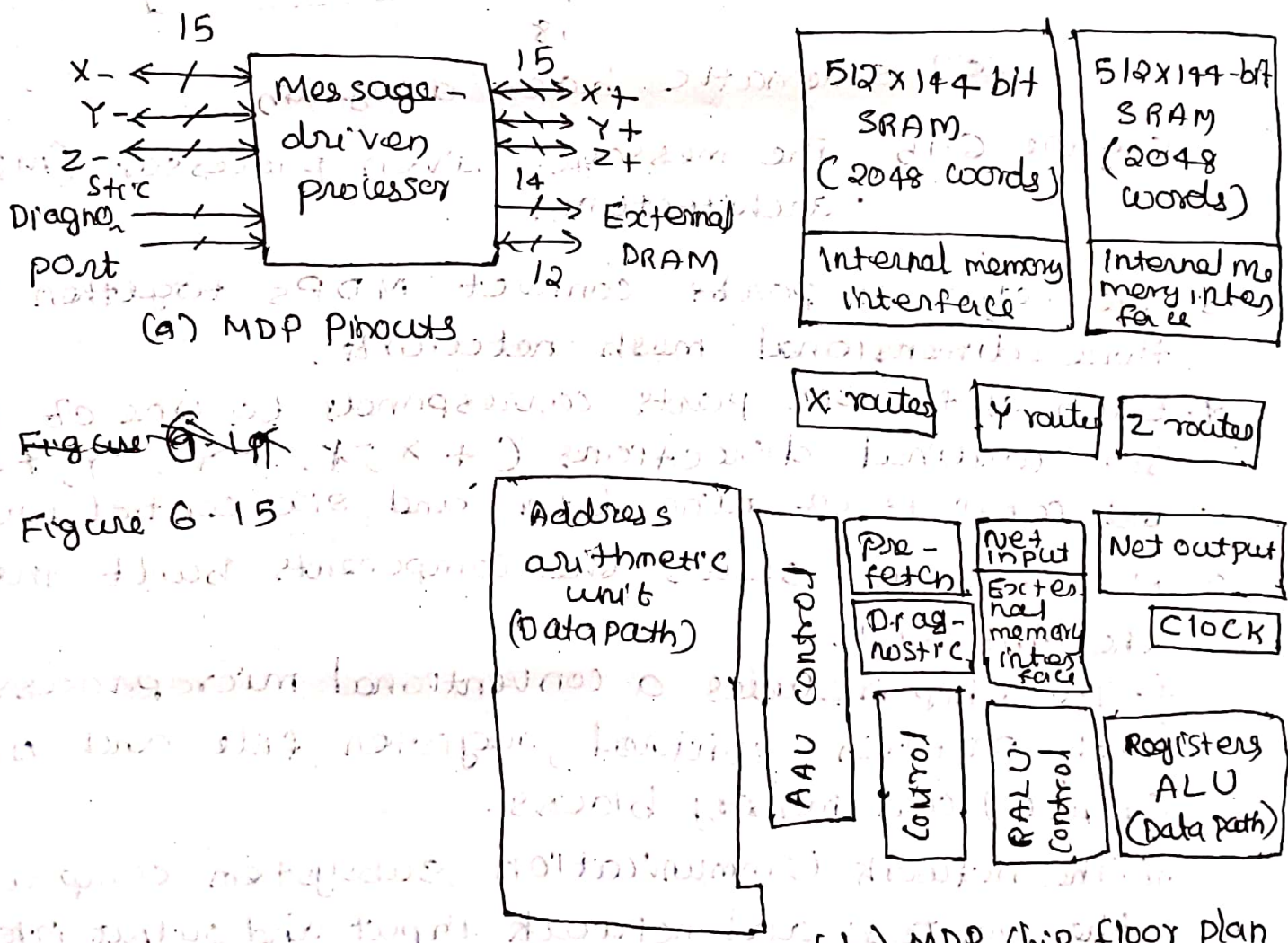
1. The k-ary n-cube networks have been applied in the MIT J-Machine.
2. The J-machine designers have called their network a three-dimensional mesh.

ii. The MDP-Design

1. The MDP chip includes a processor, a 4096-word by 36-bit memory; and a built-in-routes with network ports as shown in Figure 6.15.
2. An on-chip memory controller with error checking and correction (ECC) capability permit local memory to be expanded to 1 million words by adding external DRAM chips.
3. The processor is message-driven in the sense that it processes in response to messages, via the dispatch

mechanism.

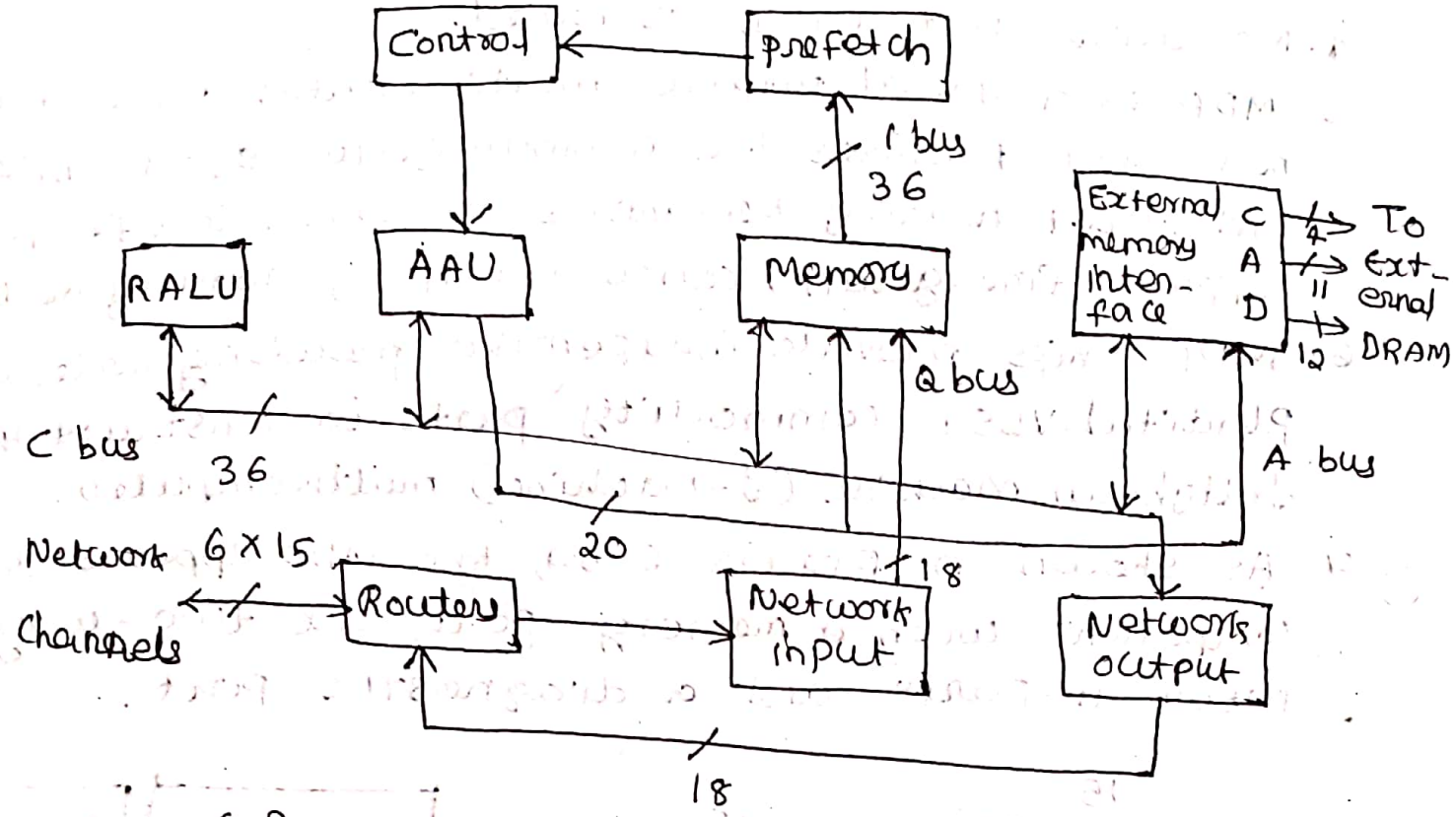
4. NO receive instruction is needed.
5. MDP is a general-purpose multicomputer processing node that provides the communication, synchronization and global naming mechanisms required to efficiently support fine-grain, concurrent programming models.
6. MDP chips provide inexpensive processing nodes with plentiful VLSI commodity parts to construct the Jellybean Machine (J-Machine) multicomputer.
7. As shown in Figure 6.16a, the MDP appears as a component with a memory port, six two-way network ports and a diagnostic port.



(a) MDP Pinouts

(b) MDP chip floor plan

Figure 6.15



(c) schematic block diagram

Figure 6.15 The message driven processor (MDP) architecture.

8. The network ports connect MDPs together in a three-dimensional mesh network.
9. Each of the six ports corresponds to one of the six cardinal directions (+x, -x, +y, -y, +z, -z) and consists of nine data and six control lines.
10. Figure 6.15(c) shows the components built inside the MDP chip.
    1- The chip includes a conventional microprocessor with prefetch, control, register file and ALU (RALU) and memory blocks.
    11. The network communication subsystem comprises the routers and network input and output interfaces.

iii. The address arithmetic unit (AAU) provides addressing functions.

ii. The MDP also includes a DRAM interface, control clock, and diagnostic interface.

iii Instruction-Set Architecture

1. The MDP extends a conventional microprocessor instruction-set architecture with instructions to support parallel processing.
2. Instruction set contains fixed-format, three-address instruction.
3. Seperate register sets are provided to support rapid switching among three execution levels:
   i. background,
   ii. priority 0 (P0), and
   iii. priority 1 (P1).
4. The P1 level has higher priority than the P0 level.
5. The register set at each priority level includes
   i. four GPRs,
   ii. four address registers,
   iii. four ID registers, and
   iv. one instruction pointer (IP).

iv. Communication support

The MDP provides hardware support for end-to-end message delivery including

1. formatting,
2. injection,
3. delivery,
4. buffer allocation,
5. buffering, and
6. task scheduling.

V. Message format and Routing

1. The J-Machine uses deterministic dimension-order E-cube routing.

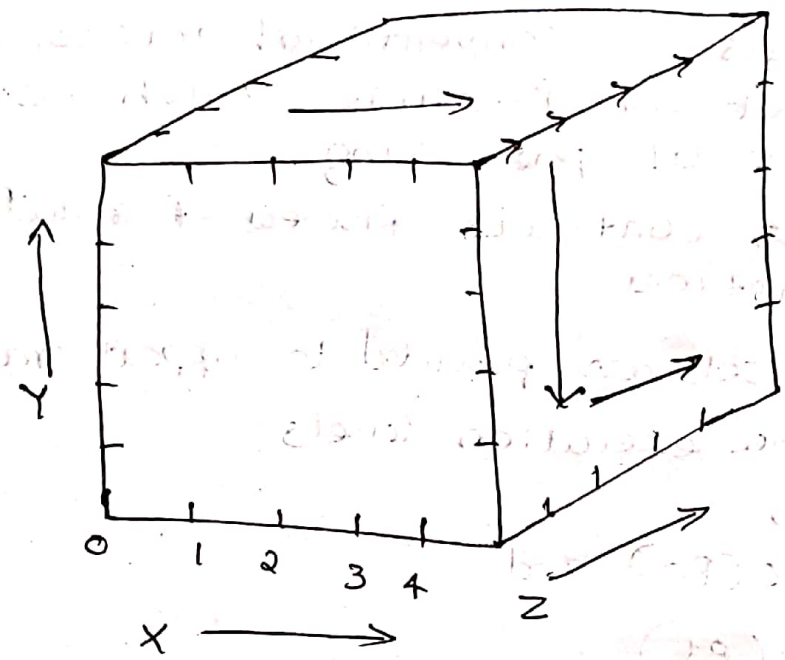


Figure 6.16 E-cube routing from node (1,5,2) to node (5,1,4) on a 6-ary 3-cube.

2. As shown in Figure 6.16, all messages route first in the X-dimension, then in the Y-direction, and then in the Z-direction.

3. Eg: A typical message in the MIT J-Machine

The following message consists of nine flits. The first three flits of the message contain the X-, Y-, and Z-addresses. Each node along the path compares the address in the head flit of the message.

If the two indices match, the node routes the rest to the next dimension. The final flit in the message is marked as the tail.

Flit	Contents		Remarks
1	5 : +		X - address
2	1 : −		Y - address
3	4 : +		Z - address
4	msg : 00		Method to call
5	0044 0		
6	INT : 00		Argument to method
7	0023		
8	INT : 00		Reply address
9	<1:5:2>	T	

4. The MDP supports a broad range of parallel programming models, including shared-memory, data-parallel, dataflow, actor, and explicit message passing, by providing a low-overhead primitive mechanism for communication, synchronization, and naming.

vi The Router Design

The routers form the switches in a J-Machine network and deliver messages to their destinations.
Each router contains two seperate virtual networks with different priorities that share the same physical channels.

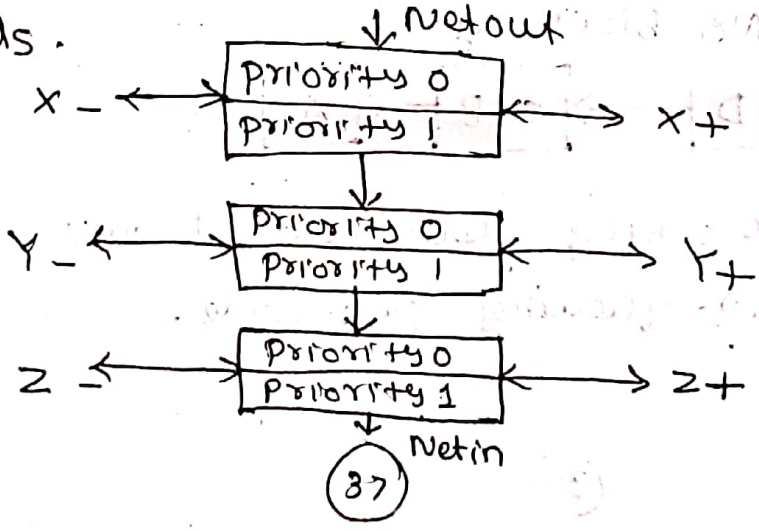


Figure 6.17 Dual-priority levels per dimension in the routes

A message entering the dimension competes with messages continuing in the dimension at a two-to-one switch. Once a message is granted this switch, all other input is locked out for the duration of the message. Once the head flit of the message has set up the route, subsequent flits follow directly behind it.

vii. Blocking Flow Control

1. The flow control in the J-Machine is illustrated in Figure 6.18. When a message arrives at a router path already in use by a message of the same priority, it is blocked.

2. The blocked message compresses into routers along its path, occupying one node per word (two flits) of the message.

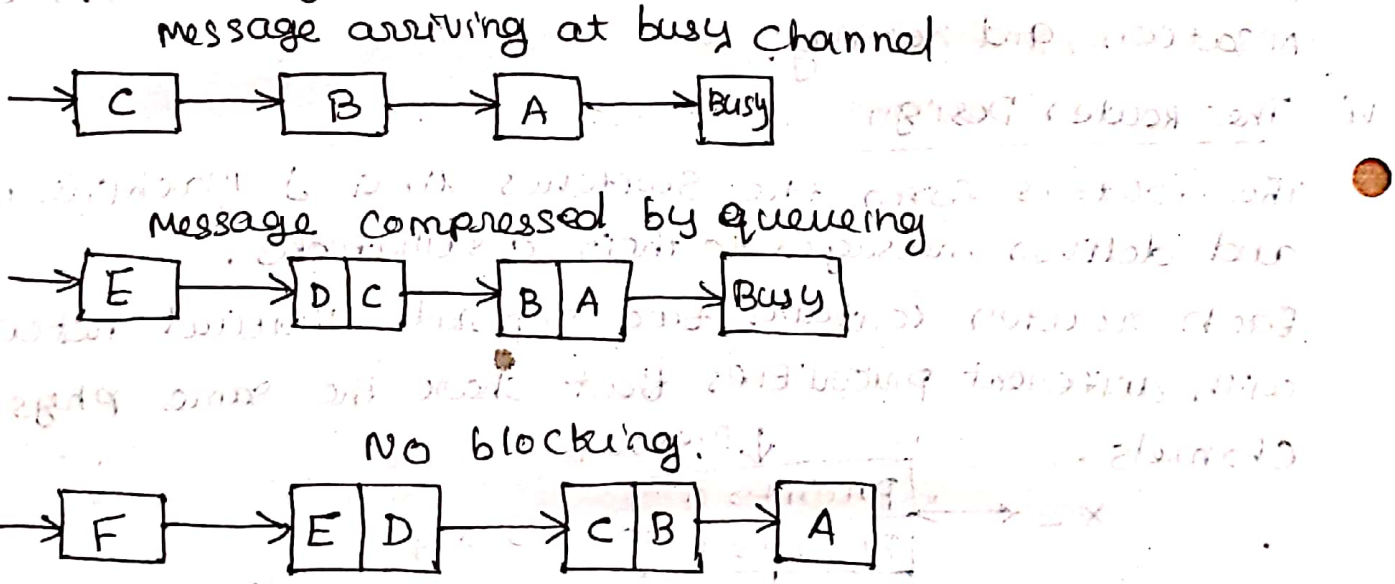


Figure 6.18. Blocking flow control with two stages of queuing per node.

viii. Global naming

1. The AAU, the largest logic block in the MDP, performs all functions associated with memory addressing.
2. To support naming and relocation, the AAU contains the address and ID registers.

ix. Synchronization

1. The MDP synchronizes using message dispatch and presence tags on all states.
2. Because each message arrival dispatches a process messages can signal events on remote nodes.

Dataflow and Hybrid Architectures.

Multithread architectures can be designed with a pure dataflow approach or with a hybrid approach combining von Neumann and data-driven mechanisms.

1. The Evolution of Dataflow Computers
2. The ETL/EM-4 in Japan
3. The MIT/Motorola *T prototype

1. The Evolution of Dataflow Computers.

1. Dataflow computers have the potential for exploiting all the parallelism available in a program.
2. Since execution is driven only by the availability of operands at the inputs to the functional units, there is no need for a program counter in this architecture, and its parallelism is limited only by the actual data dependencies in the application program.

3. Memory latency and synchronization overhead are two fundamental issues in multiprocessing.

4. Dataflow architectures represent a radical alternative to von Neumann architectures because they use dataflow graphs as their machine languages.

i. Dataflow Graphs
ii. Static versus Dynamic Dataflow
iii. Pure Dataflow Machines
iv. Explicit Token Store Machines
v. Hybrid and Unified Architectures

i. Dataflow Graphs

a) Dataflow graphs can be used as a machine language in dataflow computers.

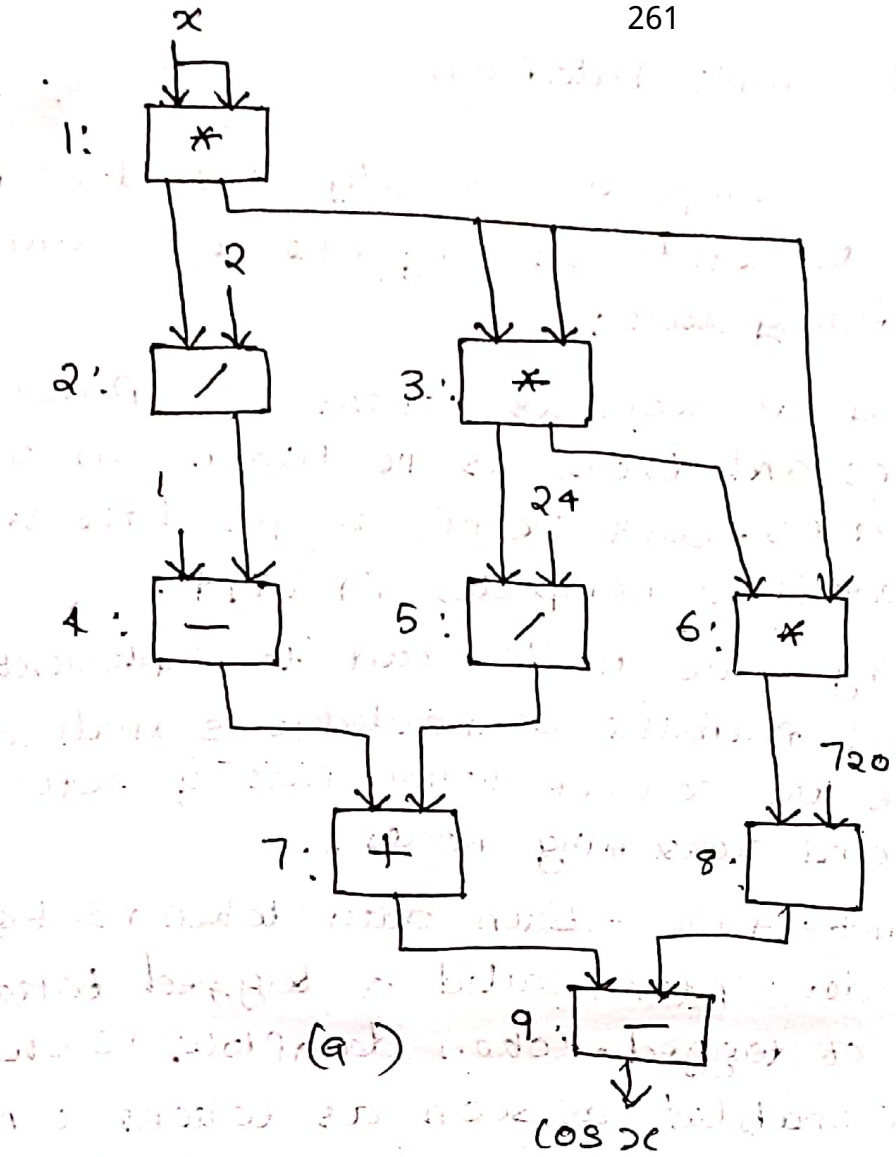
Eg: The dataflow graph for the calculation of $\cos x$.

This dataflow graph shows how to obtain an approximation of $\cos x$ by the following power series computation:

$$\cos x \simeq 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!}$$

$$= 1 - \frac{x^2}{2} + \frac{x^4}{24} - \frac{x^6}{720}$$

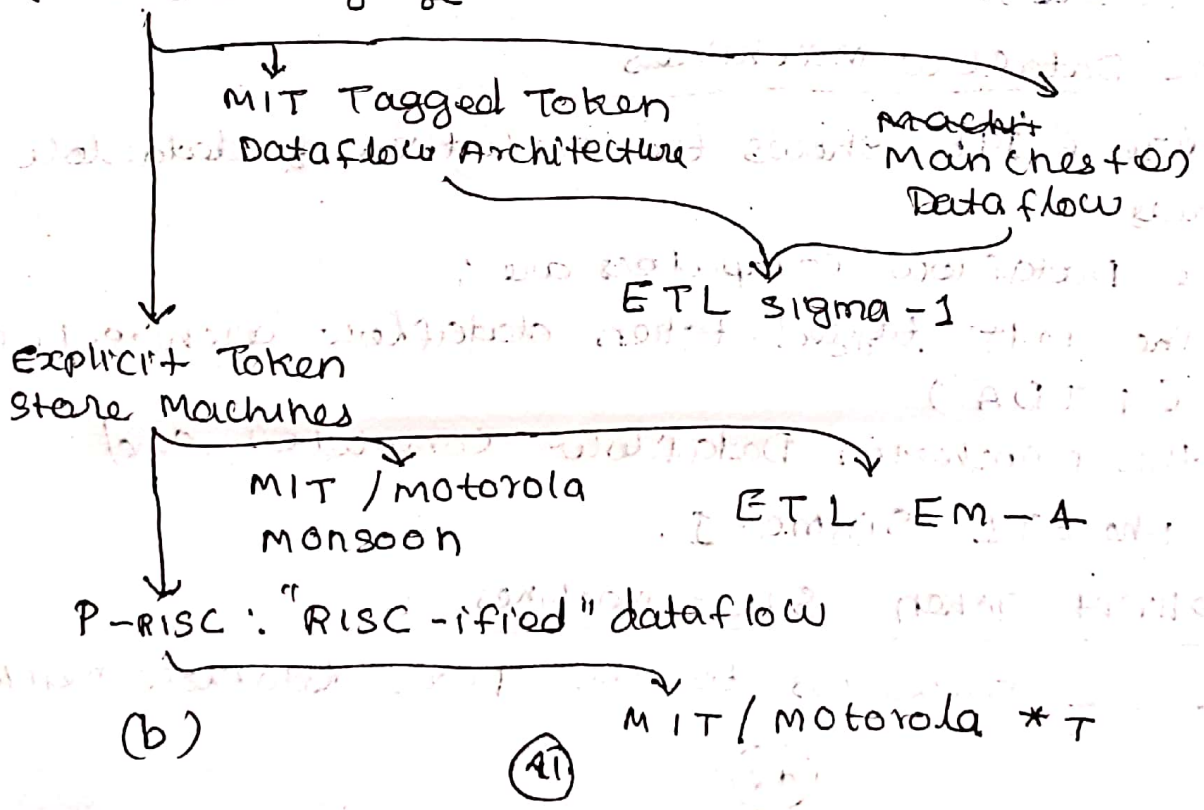
The corresponding dataflow graph consists of nine operators (actors or nodes). The edges in the graph interconnect the operator nodes. The successive powers of x are obtained by repeated multiplications. The constants (divisors) are fed into the nodes directly. All intermediate results are forwarded among the nodes.



(a)

Figure 6.19
(a) Dataflow graph for computing cos x.
(b) Evolution tree of dynamic data flow machines.

Dataflow graphs as a machine language



(b)

ii. Static versus Dynamic Dataflow

1. Static dataflow computers simply disallow more than one token to reside on any one arc, which is enforced by the firing rule:

   A node is enabled as soon as tokens are present on all input arcs and there is no token on any of its output arcs. Jack Dennis proposed the very first static dataflow computer in 1974.

2. The static firing rule is difficult to implement in hardware. Special feedback acknowledge signals are needed to secure the correct token passing between producing nodes and consuming nodes.

3. Dynamic architecture - Each data token is tagged with a context descriptor, called a tagged token. The firing rule of tagged-token dataflow is changed to: A node is enabled as soon as tokens with identical tags are present at each of its input arcs.

iii. Pure Dataflow Machines

Figure 6.19b shows the evolution of dataflow computers.

Pure Dataflow computers are:

a) The MIT tagged-token dataflow architecture (TTDA)

b) the Manchester Dataflow Computer, and

c) the ETL Sigma-I.

iv. Explicit Token Store Machines

1. These are successors to the pure dataflow machines.

2. The basic idea is to eliminate associative token matching.
3. The waiting token memory is directly addressed, plus the use of full/empty bits.
4. This idea was used in the MIT/Motorola Monsoon and in the ETL EM-4 system.

V. Hybrid and Unified Architectures

1. These are architectures combining positive features from the von Neumann and dataflow architectures.
   eg: i. MIT P-RISC,
   ii. the IBM Empire,
   iii. the MIT/Motorola *T.

2. The ETL / EM-4 in Japan:

1. The EM-4 has a global organisation as shown in Figure 6.20.
2. Each EMC-R node is a single-chip processor without floating-point hardware but including a switch of the network.
3. Each node plays the role of I-Structure memory and has 1-31 Mbytes of static RAM.
4. An omega network is built to provide the interconnections among the nodes.

i. The node Architecture
ii. Future Effort.

i. The Node Architecture

1. The internal design of the processor chip and of the

node memory are shown in Figure 6.20b.

2. The processor chip communicates with the network through a 3×3 crossbar switch unit.

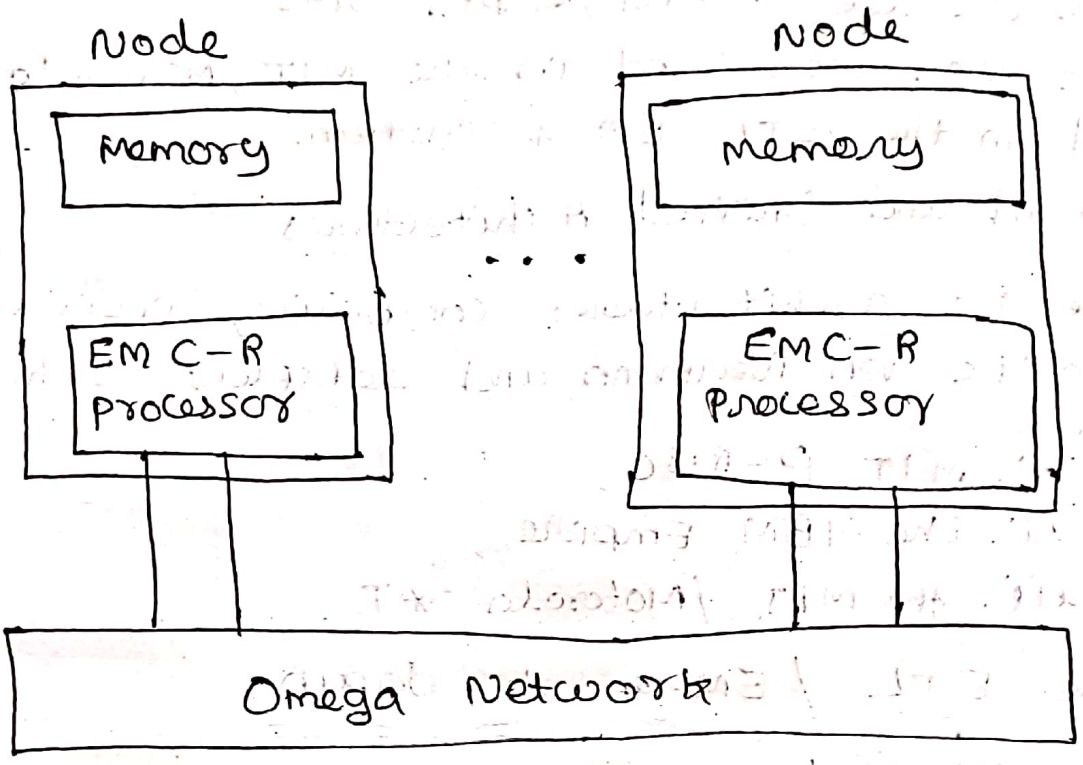
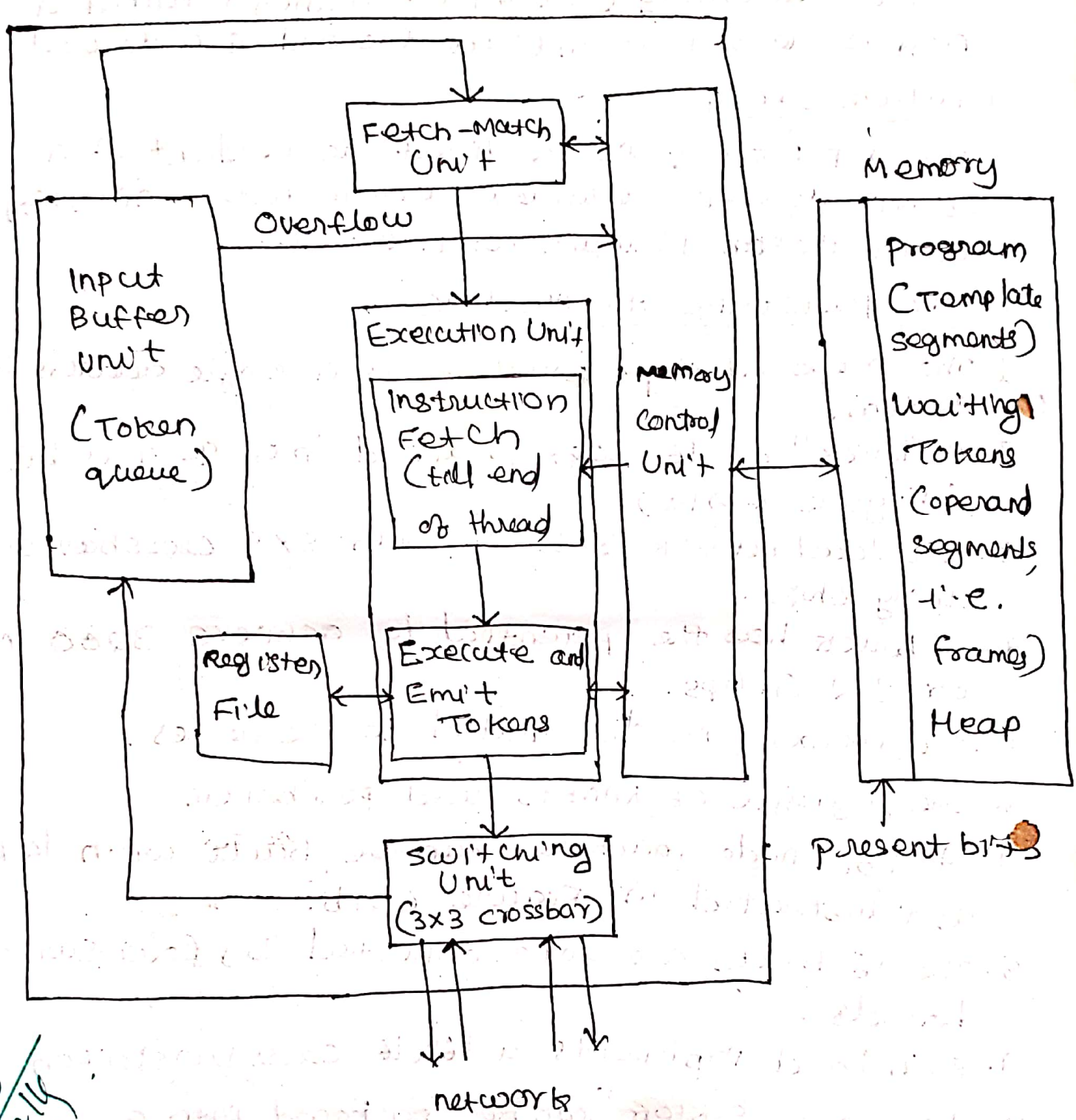


Figure 6.20 (a) Global organization

3. The processor and its memory are interfaced with a memory control unit.
4. The processor consists of six component units
5. The input buffer is used as a token store with a capacity of 32 words.
6. The fetch-match unit fetches tokens from the memory and performs the tag-matching operations among the tokens fetched in.
7. Instructions are directly fetched from the memory through the memory controller.
8. The heart of the processor is the execution unit

which fetches instructions until the end of a thread.



(b) The EMC-R processor design.

Figure 6.20 The ETL EM-4 dataflow architecture.

9. Instructions are fetched continually using traditional sequencing (PC+1 or branch) until a "stop" flag is raised to indicate the end of a thread.

iii Future Effort

The *T project is a direct descendant of a series of MIT dynamic dataflow architectures unifying with the von Neumann architectures.

i: The Prototype Architecture

1. The proposed *T prototype is a single-address-space system.
2. A "brick" of 16 nodes is packed in a 4-in cube (Figure 6.21a).
3. The local network is built with 8x8 crossbar switching chips.
4. A brick has the potential to achieve 3200 MIPS or 3.2 Gflops.
5. The memory is distributed to the nodes.
6. One gigabyte of RAM is used per brick.
7. A 256-node machine can be built with 16 bricks as illustrated in Figure 6.21b.
8. The 16 bricks are interconnected by four switching boards.
9. Each board implements a 16x16 crossbar switch.
10. The entire system can be packaged into a 1.5-m cube.
11. No cables are used between the boards.

APPENDIX 1

CONTENT BEYOND THE SYLLABUS

1. Computer Hardware and Software for the Generation of Virtual Environments

The computer technology that allows us to develop three-dimensional virtual environments (VEs) consists of both hardware and software. The current popular, technical, and scientific interest in VEs is inspired, in large part, by the advent and availability of increasingly powerful and affordable visually oriented, interactive, graphical display systems and techniques. Graphical image generation and display capabilities that were not previously widely available are now found on the desktops of many professionals and are finding their way into the home.

One possible organization of the computer technology for VEs is to decompose it into functional blocks. In Figure 1, three distinct classes of blocks are shown: (1) rendering hardware and software for driving modality-specific display devices; (2) hardware and software for modality-specific aspects of models and the generation of corresponding display representations; (3) the core hardware and software in which modality-independent aspects of models as well as consistency and registration among multimodal models are taken into consideration. Beginning from left to right, human sensorimotor systems, such as eyes, ears, touch, and speech, are connected to the computer through human-machine interface devices. These devices generate output to, or receive input from, the human as a function of sensory modal drivers or renderers.

The auditory display driver, for example, generates an appropriate waveform based on an acoustic simulation of the VE. To generate the sensory output, a computer must simulate the VE for that particular sensory mode. For example, a haptic display may require a physical simulation that includes compliance and texture.

An acoustic display may require sound models based on impact, vibration, friction, fluid flow, etc. Each sensory modality requires a simulation tailored to its particular output. Next, a unified representation is necessary to coordinate individual sensory models and to synchronize output for each sensory driver. This representation must account for all human participants in the VE, as well as all autonomous internal entities. Finally, gathered and computed information must be summarized and broadcast over the network in order to maintain a consistent distributed simulated environment.

2. The computer technology for the generation of VEs.

The computer hardware used to develop three-dimensional VEs includes high-performance workstations with special components for multisensory displays, parallel processors for the rapid computation of world models, and high-speed computer networks for transferring information among participants in the VE. The implementation of the virtual world is accomplished with software for interaction, navigation, modeling (geometric, physical, and behavioral), communication, and hypermedia integration. Control devices and head-mounted displays are covered elsewhere in this report.

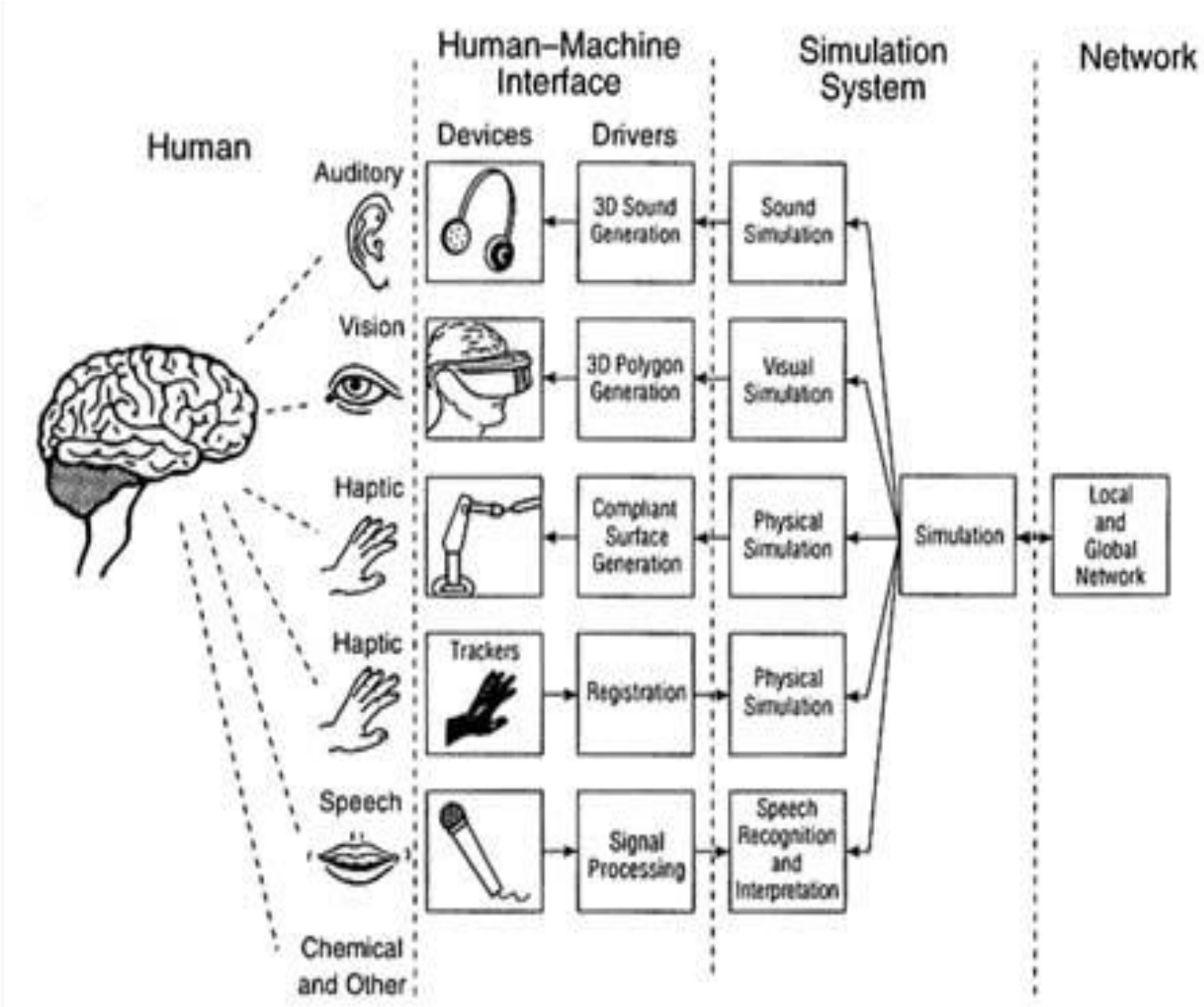


FIGURE 1 Organization of the computer technology for virtual reality.

3. Graphics Architectures for VE Rendering

The high-level computer architecture issues that determine the applicability of a graphics system to VE rendering. Two assumptions are made about the systems included in our discussion. First, they use a *z-buffer* (or depth buffer), for hidden surface elimination. A z-buffer stores the depth—or distance from the eye point—of the closest surface "seen" at that pixel. When a new surface is scan converted, the depth at each pixel is computed. If the new depth at a given pixel is closer to the eye point than the depth currently stored in the z-buffer at that pixel, then the new depth and intensity information are written into both the z-buffer and the frame buffer. Otherwise, the new information is discarded and the next pixel is examined. In this way, nearer objects always overwrite more distant objects, and when every object has been scan converted, all surfaces have been correctly ordered in depth. The second assumption for these graphic systems is that they use an application-programmable, general-purpose processor to cull the database. The result is to provide the rendering hardware with only the graphics primitives that are within the viewing volume (a perspective pyramid or parallel piped for perspective and parallel projections respectively). Both of these assumptions are valid for commercial graphics workstations and for the systems that have been designed by researchers at the University of North Carolina at Chapel Hill.

The rendering operation is composed of three stages: per-primitive,

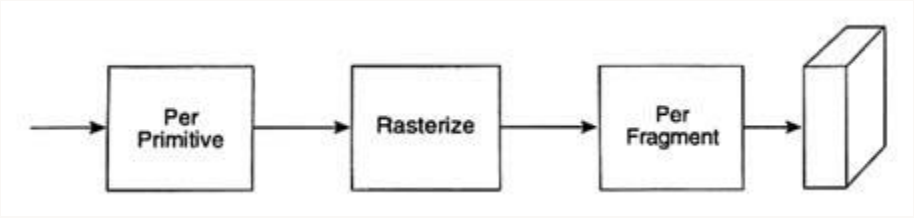


FIGURE 2 The graphics pipeline.

rasterization, and per-fragment (as shown in Figure 2). *Per-primitive* operations are those that are performed on the points, lines, and triangles that are presented to the rendering system. These include transformation of vertices from object coordinates to world, eye, view volume, and eventually to window coordinates, lighting calculations at each vertex, and clipping to the visible viewing volume. *Rasterization* is the process of converting the window-coordinate primitives to fragments corresponding to the pixels held in the frame buffer. The frame buffer is a dedicated block of memory that holds intensity and other information for every pixel on the display surface. The frame buffer is scanned repeatedly by the display hardware to generate visual imagery. Each of the fragments includes *x* and *y* window coordinates, a color, and a depth for use with the z-buffer for hidden surface elimination. Finally, *per-fragment* operations include comparing the fragment's depth value to the value stored in the z-buffer and, if the comparison is successful, replacing the color and depth values in the frame buffer with the fragment's values.

The performance demanded of such a system can be substantial: 1 million triangles per second or hundreds of millions of fragments per second. The calculations involved in performing this work easily require billions of operations per second. Since none of today's fastest general purpose processors can satisfy these demands, all modern high-performance graphics systems are run on parallel architectures. Figure 3 is a general representation of a parallel architecture, in which the rendering operation of Figure 2 is simply replicated. Whereas such an architecture is attractively simple to implement, it fails to solve the rendering problem, because primitives in object coordinates cannot be easily separated into groups corresponding to different subregions of the frame buffer. There is in general a many-to-many mapping between the primitives in object coordinates and the partitions of the frame buffer.

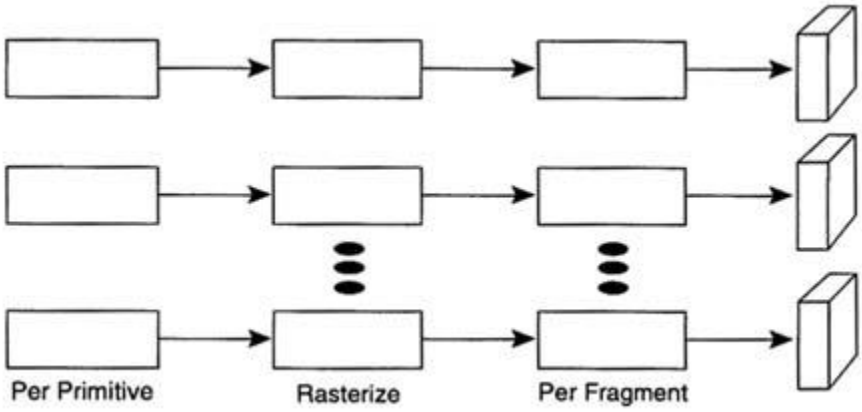


FIGURE 3 Parallel graphics pipelines.

To allow for this many-to-many mapping, disjoint parallel rendering pipes must be combined at a minimum of one point along their paths, and this point must come after the per-primitive operations are completed. The point or crossbar can be located prior to the rasterization (the primitive crossbar), between rasterization and per-fragment (the fragment crossbar), and following pixel merge (the pixel merge crossbar). A detailed discussion of these architectures is provided in the technical appendix to this chapter. There are four major graphics systems that represent different architectures based on crossbar location. Silicon Graphics RealityEngine is a flow-through architecture with a primitive crossbar; the Freedom series from Evans & Sutherland is a flow-through architecture with a fragment crossbar; Pixel Planes 5 uses a tiled primitive crossbar; and PixelFlow is a tiled, pixel merge machine.