ELECTIVE I

ADVANCED COMPUTER ARCHITECTURE

OBJECTIVES:

- To study parallel computer architecture, design and micro-operations
- To understand the interconnection networks and synchronization mechanism

UNIT-I

Evolution of Computer systems – Parallelism in Uniprocessor Systems: Architecture, Mechanisms – Parallel Computer Structures: Pipeline , Array, Multiprocessor.

UNIT – II

Linear Pipeline processors: Asynchronous and Synchronous Models – Non-linear PipelineProcessors: Reservation and Latency Analysis –Collision-free scheduling – Instruction Pipeline Design: Instruction Execution Phases – Mechanisms f or Instruction Pipelining – Arithmetic Pipeline Design: Computer Arithmetic Principles – Static Arithmetic Pipelines – Multifunctional Arithmetic Pipelines - Superscalar Pipeline Design.

UNIT- III

SIMD Array Processor – SIMD Interconnection Network: Static vs Dynamic Network – Mesh connection Iliac Network- Tube interconnection Network. Associative Array Processing: Associative memory organization.

UNIT – IV

Multiprocessor System Interconnects: Hierarchical Bus System - Crossbar Switch and Multiport Memory - Multistage and Combining Networks – Cache Coherence and Synchronization Mechanisms: The Cache Coherence Problem – Snoopy Bus Protocols – Directory-Based Protocols – Hardware Synchronization Mechanisms – Message-Passing Mechanisms: Message-Routing Schemes – Deadlock and Virtual Channels – Flow Control Strategies – Multicast Routing Algorithms.

UNIT – V

Multiprocessor Operating Systems- Interprocessor Communication Mechanisms - Multiprocessor Scheduling Strategies.

TEXT BOOKS

1. Kai Hwang, Faye A.Briggs, “Computer Architecture and Parallel Processing,” McGrawHill, 1985.

2. Kai Hwang, “Advanced Computer Architecture,” McGraw -Hill International Editions, 2001.

REFERENCES

1. Grama, "An Introduction to Parallel Computing: Design and Analysis of Algorithms," 2 nd Edition, Pearson, 2004.

2. Gita Alaghband, Harry Frederick Jordan, "Fundamentals of Parallel Processing," Prentice Hall, 2003.

3. Seyed H Roosta, "Parallel Processing and Parallel Algorithms: Theory and Computation," Springer Science & Business Madia, 1999

OUTCOMES:

On completion of the course the student can understand

- Parallel computer architecture, design and micro-operations
- Interconnection of networks and synchronization mechanism
- Develop design skills of Instruction Sets
- Know how to design a pipelined data path

CHAPTER

ONE

INTRODUCTION TO PARALLEL PROCESSING

Basic concepts of parallel processing on high-performance computers are introduced in this chapter. We will review the architectural evolution, examine various forms of concurrent activities in modern computer systems, and assess advanced applications of parallel processing computers. Parallel computer structures will be characterized as *pipelined computers*, *array processors*, and *multiprocessor systems*. Several new computing concepts, including data flow and VLSI approaches, will be introduced. The material presented in this introductory chapter will provide an overview of the field and pave the way to studying in subsequent chapters the details of theories of parallel computing, machine architectures, system controls, fast algorithms, and programming requirements.

1.1 EVOLUTION OF COMPUTER SYSTEMS

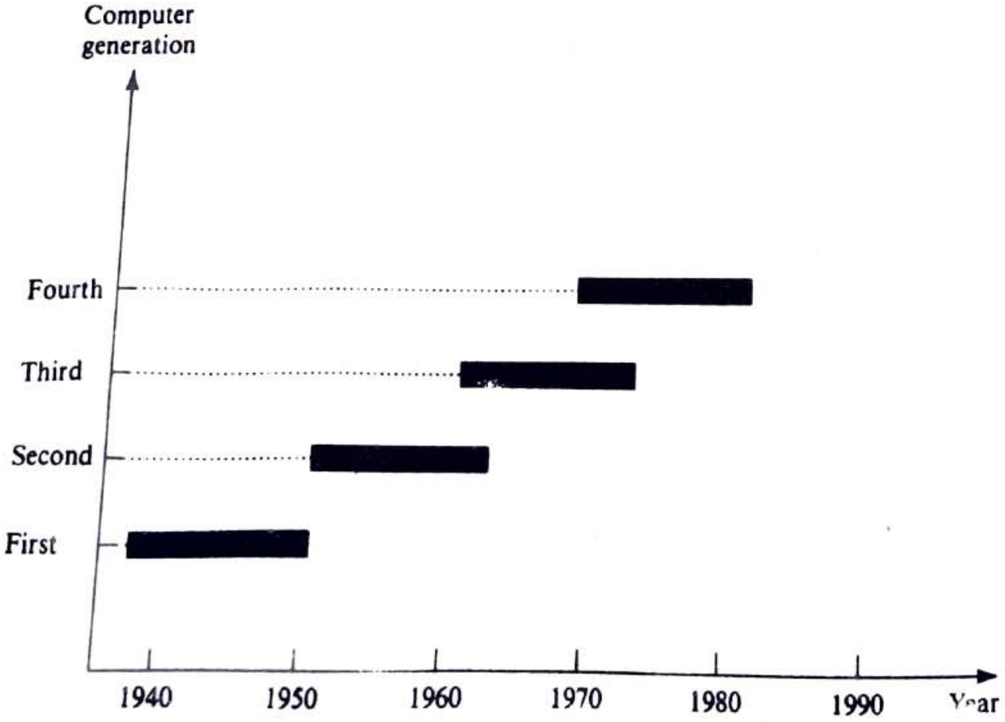
Over the past four decades the computer industry has experienced four generations of development, physically marked by the rapid changing of building blocks from relays and vacuum tubes (1940–1950s) to discrete diodes and transistors (1950–1960s), to small- and medium-scale integrated (SSI/MSI) circuits (1960–1970s), and to large- and very-large-scale integrated (LSI/VLSI) devices (1970s and beyond). Increases in device speed and reliability and reductions in hardware cost and physical size have greatly enhanced computer performance. However, better devices are not the sole factor contributing to high performance. Ever since the stored-program concept of von Neumann, the computer has been recognized as more than just a hardware organization problem. A modern computer system is really a composite of such items as processors, memories, functional units, interconnection networks, compilers, operating systems, peripheral devices, communication channels, and database banks.

To design a powerful and cost-effective computer system and to devise efficient programs to solve a computational problem, one must understand the underlying hardware and software system structures and the computing algorithms to be implemented on the machine with some user-oriented programming languages. These disciplines constitute the technical scope of *computer architecture*. Computer architecture is really a system concept integrating hardware, software, algorithms, and languages to perform large computations. A good computer architect should master all these disciplines. It is the revolutionary advances in integrated circuits and system architecture that have contributed most to the significant improvement of computer performance during the past 40 years. In this section, we review the generations of computer systems and indicate the general trends in the development of high performance computers.

1.1.1 Generations of Computer Systems

The division of computer systems into generations is determined by the device technology, system architecture, processing mode, and languages used. We consider each generation to have a time span of about 10 years. Adjacent generations may overlap in several years as demonstrated in Figure 1.1. The long time span is intended to cover both development and use of the machines in various parts of the world. We are currently in the fourth generation, while the fifth generation is not materialized yet.

The first generation (1938–1953) The introduction of the first electronic analog computer in 1938 and the first electronic digital computer, ENIAC (Electronic Numerical Integrator and Computer), in 1946 marked the beginning of the first generation of computers. Electromechanical relays were used as switching devices



igure 1.1 The evolution of computer systems.

in the 1940s, and vacuum tubes were used in the 1950s. These devices were interconnected by insulated wires. Hardware components were expensive then, which forced the CPU structure to be *bit-serial*: arithmetic is done on a bit-by-bit fixed-point basis, as in a ripple-carry addition which uses a single full adder and one bit of carry flag.

Only binary-coded machine language was used in early computers. In 1950, the first stored-program computer, EDVAC (Electronic Discrete Variable Automatic Computer), was developed. This marked the beginning of the use of system software to relieve the user's burden in low-level programming. However, it is not difficult to imagine that hardware costs predominated and software-language features were rather primitive in the early computers. By 1952, IBM had announced its 701 electronic calculator. The system used Williams' tube memory, magnetic drums, and magnetic tape.

The second generation (1952–1963) Transistors were invented in 1948. The first *tra*nsistorized *di*gital *c*omputer, TRADIC, was built by Bell Laboratories in 1954. Discrete transistors and diodes were the building blocks: 800 transistors were used in TRADIC. Printed circuits appeared. By this time, coincident current magnetic *core* memory was developed and subsequently appeared in many machines. Assembly languages were used until the development of high-level languages, Fortran (*for*mula *tran*slation) in 1956 and Algol (*algo*rithmic *l*anguage) in 1960.

In 1959, Sperry Rand built the Larc system and IBM started the Stretch projeet. These were the first two computers attributable to architectural improvement. The Larc had an independent I/O processor which operated in parallel with one or two processing units. Stretch featured instruction lookahead and error correction, to be discussed in Section 1.2. The first IBM scientific, transistorized computer, IBM 1620, became available in 1960. Cobol (*co*mmon *b*usiness *o*riented *l*anguage) was developed in 1959. Interchangeable disk packs were introduced in 1963. Batch processing was popular, providing sequential execution of user programs, one at a time until done.

The third generation (1962–1975) This generation was marked by the use of small-scale integrated (SSI) and medium-scale integrated (MSI) circuits as the basic building blocks. Multilayered printed circuits were used. Core memory was still used in CDC-6600 and other machines but, by 1968, many fast computers, like CDC-7600, began to replace cores with solid-state memories. High-level languages were greatly enhanced with intelligent compilers during this period.

Multiprogramming was well developed to allow the simultaneous execution of many program segments interleaved with I/O operations. Many high-performance computers, like IBM 360/91, Illiac IV, TI-ASC, Cyber-175, STAR-100, and C.mmp, and several vector processors were developed in the early seventies. Time-sharing operating systems became available in the late 1960s. Virtual memory was developed by using hierarchically structured memory systems.

The fourth generation (1972–present) The present generation computers emphasize the use of large-scale integrated (LSI) circuits for both logic and memory sections. High-density packaging has appeared. High-level languages are being extended to handle both scalar and vector data, like the extended Fortran in many vector processors. Most operating systems are time-sharing, using virtual memories. Vectorizing compilers have appeared in the second generation of vector machines, like the Cray-1 (1976) and the Cyber-205 (1982). High-speed mainframes and supers appear in multiprocessor systems, like the Univac 1100/80 (1976), Fujitsu M 382 (1981), the IBM 370/168 MP, the IBM 3081 (1980), the Burroughs B-7800 (1978), and the Cray X-MP (1983). A high degree of pipelining and multiprocessing is greatly emphasized in commercial supercomputers. A massively parallel processor (MPP) was custom-designed in 1982. This MPP, consisting of 16,384 bit-slice microprocessors, is under the control of one array controller for satellite image processing.

The future Computers to be used in the 1990s may be the next generation. Very-large-scale integrated (VLSI) chips will be used along with high-density modular design. Multiprocessors like the 16 processors in the S-1 project at Lawrence Livermore National Laboratory and in the Denelcor's HEP will be required. Cray-2 is expected to have four processors, to be delivered in 1985. More than 1000 *mega float-point operations per second* (megaflops) are expected in these future supercomputers. We will study major existing systems and discuss possible future machines in subsequent chapters.

1.2 PARALLELISM IN UNIPROCESSOR SYSTEMS

Most general-purpose uniprocessor systems have the same basic structure. In this section, we will briefly review the architecture of uniprocessor systems. The development of parallelism in uniprocessors will then be introduced categorically. It is assumed that readers have had at least one basic course in the past on conventional computer organization. Therefore, we will provide only concise specifications of the architectural features of two popular commercial computers. Parallel-processing mechanisms and methods to balance subsystem bandwidths will then be described for a typical uniprocessor system. Details of these structures, mechanisms, and methods can be found in references suggested in the bibliographic notes.

1.2.1 Basic Uniprocessor Architecture

A typical uniprocessor computer consists of three major components: the *main memory*, the *central processing unit* (CPU), and the *input-output* (I/O) *subsystem*. The architectures of two commercially available uniprocessor computers are given below to show the possible interconnection of structures among the three subsystems. We will examine major components in the CPU and in the I/O subsystem.

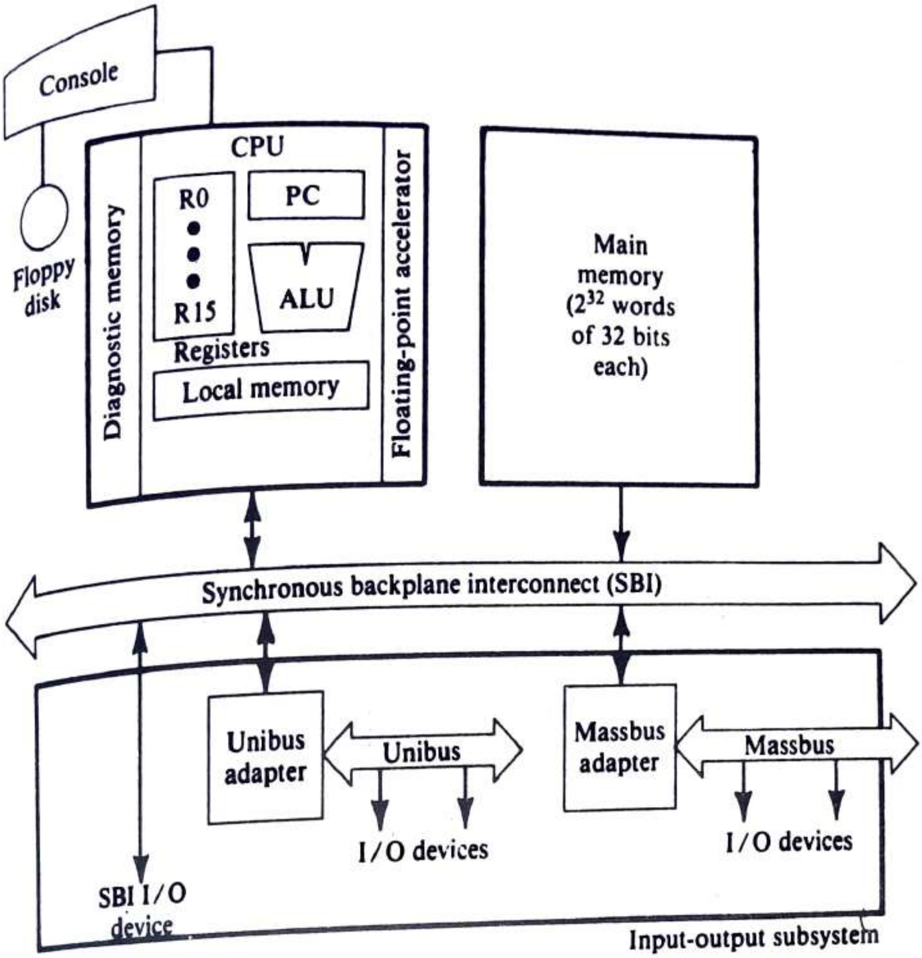


Figure 1.3 The system architecture of the supermini VAX-11/780 uniprocessor system (Courtesy of Digital Equipment Corporation).

Figure 1.3 shows the architectural components of the super minicomputer VAX-11/780, manufactured by Digital Equipment Company. The CPU contains the *master controller* of the VAX system. There are sixteen 32-bit general-purpose registers, one of which serves as the *program counter* (PC). There is also a special *CPU status register* containing information about the current state of the processor and of the program being executed. The CPU contains an *arithmetic and logic unit* (ALU) with an optional *floating-point accelerator*, and some local *cache memory* with an optional *diagnostic memory*. The CPU can be intervened by the operator through the console connected to a floppy disk.

The CPU, the main memory (2^{32} words of 32 bits each), and the I/O subsystems are all connected to a common bus, the *synchronous backplane interconnect* (SBI). Through this bus, all I/O devices can communicate with each other, with the CPU, or with the memory. Peripheral storage or I/O devices can be connected directly to the SBI through the *unibus* and its controller (which can be connected to PDP-11 series minicomputers), or through a *massbus* and its controller.

Another representative commercial system is the mainframe computer IBM System 370/Model 168 uniprocessor, shown in Figure 1.4. The CPU contains the

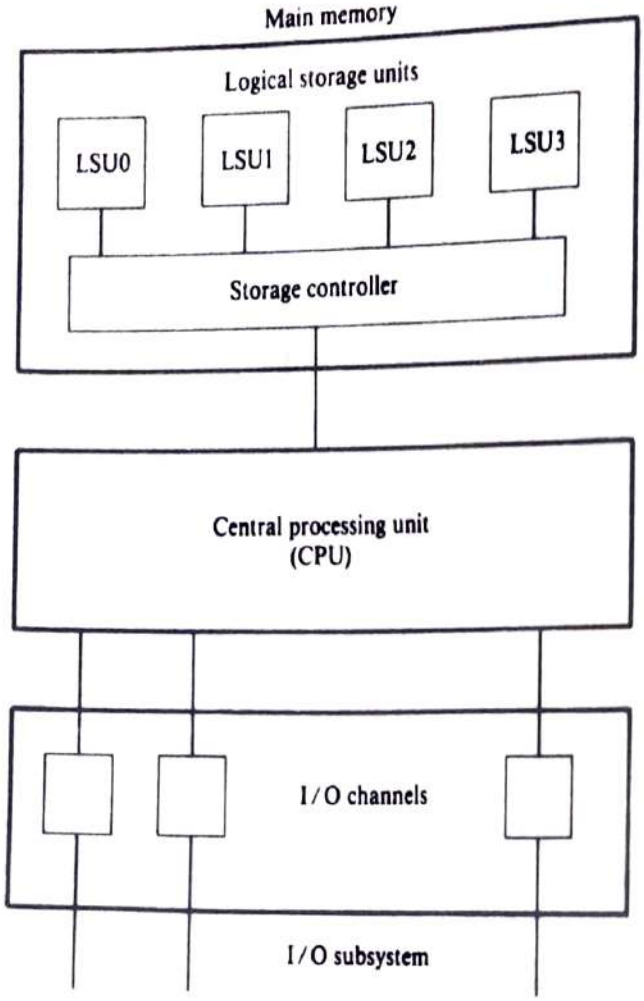


Figure 1.4 The system architecture of the mainframe IBM System 370/Model 168 uniprocessor computer (Courtesy of International Business Machines Corp.).

instruction decoding and execution units as well as a cache. Main memory is divided into four units, referred to as *logical storage units* (LSU), that are four-way interleaved. The *storage controller* provides multiport connections between the CPU and the four LSUs. Peripherals are connected to the system via high-speed I/O *channels* which operate asynchronously with the CPU. In Chapter 9, we will show that this uniprocessor can be modified to assume some multiprocessor configurations.

Hardware and software means to promote parallelism in uniprocessor systems are introduced in the next three subsections. We begin with hardware approaches which emphasize resource multiplicity and time overlapping. It is necessary to balance the processing rates of various subsystems in order to avoid bottlenecks and to increase total *system throughput*, which is the number of instructions (or basic computations) performed per unit time. Finally, we study operating system software approaches to achieve parallel processing with better utilization of the system resources.

1.2.2 Parallel Processing Mechanisms

A number of parallel processing mechanisms have been developed in uniprocessor computers. We identify them in the following six categories:

- Multiplicity of functional units
- Parallelism and pipelining within the CPU
- Overlapped CPU and I/O operations
- Use of a hierarchical memory system
- Balancing of subsystem bandwidths
- Multiprogramming and time sharing

We will describe below the first four techniques and discuss the remaining two approaches in the subsections to follow.

Multiplicity of functional units The early computer had only one arithmetic and logic unit in its CPU. Furthermore, the ALU could only perform one function at a time, a rather slow process for executing a long sequence of arithmetic logic instructions. In practice, many of the functions of the ALU can be distributed to multiple and specialized functional units which can operate in parallel. The CDC-6600 (designed in 1964) has 10 functional units built into its CPU (Figure 1.5). These 10 units are independent of each other and may operate simultaneously. A *scoreboard* is used to keep track of the availability of the functional units and registers being demanded. With 10 functional units and 24 registers available, the instruction issue rate can be significantly increased.

Another good example of a multifunction uniprocessor is the IBM 360/91 (1968), which has two parallel *execution units* (E units): one for fixed-point

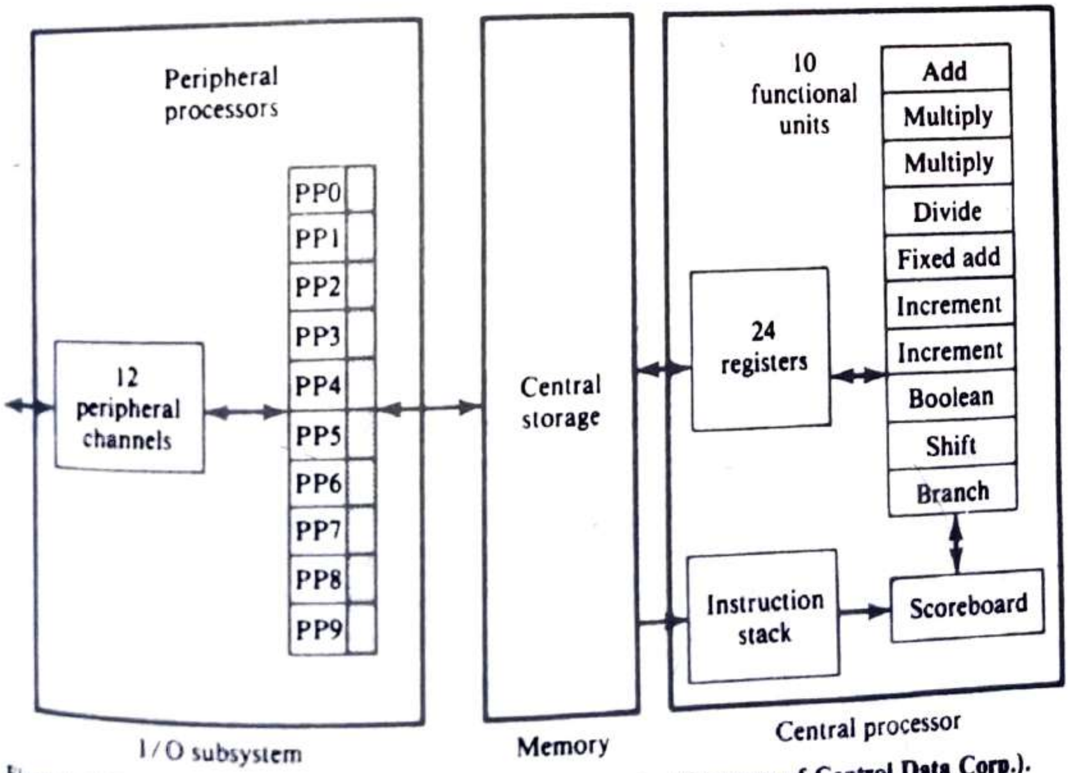


Figure 1.5 The system architecture of the CDC-6600 computer (Courtesy of Control Data Corp.).

arithmetic, and the other for floating-point arithmetic. Within the floating-point E unit are two functional units: one for floating-point add-subtract and the other for floating-point multiply-divide. IBM 360/91 is a highly pipelined, multifunction, scientific uniprocessor. We will study 360/91 in detail in Chapter 3. Almost all modern computers and attached processors are equipped with multiple functional units to perform parallel or simultaneous arithmetic logic operations. This practice of functional specialization and distribution can be extended to array processors and multiprocessors, to be discussed in subsequent chapters.

Parallelism and pipelining within the CPU Parallel adders, using such techniques as carry-lookahead and carry-save, are now built into almost all ALUs. This is in contrast to the bit-serial adders used in the first-generation machines. High-speed multiplier recoding and convergence division are techniques for exploring parallelism and the sharing of hardware resources for the functions of multiply and divide (to be described in Section 3.2.2). The use of multiple functional units is a form of parallelism with the CPU.

Various phases of instruction executions are now pipelined, including instruction fetch, decode, operand fetch, arithmetic logic execution, and store result. To facilitate overlapped instruction executions through the pipe, instruction prefetch and data buffering techniques have been developed. Instruction and arithmetic pipeline designs will be covered in Chapters 3 and 4. Most commercial uniprocessor systems are now pipelined in their CPU with a clock rate between 10 and 500 ns.

Overlapped CPU and I/O operations I/O operations can be performed simultaneously with the CPU computations by using separate I/O controllers, channels, or I/O processors. The direct-memory-access (DMA) channel can be used to provide direct information transfer between the I/O devices and the main memory. The DMA is conducted on a *cycle-stealing* basis, which is apparent to the CPU. Furthermore, I/O multiprocessing, such as the use of the 10 I/O processors in CDC-6600 (Figure 1.5), can speed up data transfer between the CPU (or memory) and the outside world. I/O subsystems for supporting parallel processing will be described in Section 2.5. Back-end database machines can be used to manage large databases stored on disks.

Use of hierarchical memory system Usually, the CPU is about 1000 times faster than memory access. A hierarchical memory system can be used to close up the speed gap. Computer memory hierarchy is conceptually illustrated in Figure 1.6. The innermost level is the register files directly addressable by ALU. Cache memory can be used to serve as a buffer between the CPU and the main memory. Block access of the main memory can be achieved through multiway interleaving across parallel memory modules (see Figure 1.4). Virtual memory space can be established with the use of disks and tape units at the outer levels.

Details of memory subsystems for both uniprocessor and multiprocessor computers are given in Chapter 2. Various interleaved memory organizations are given in Section 3.1.4. Parallel memories for array processors are treated in

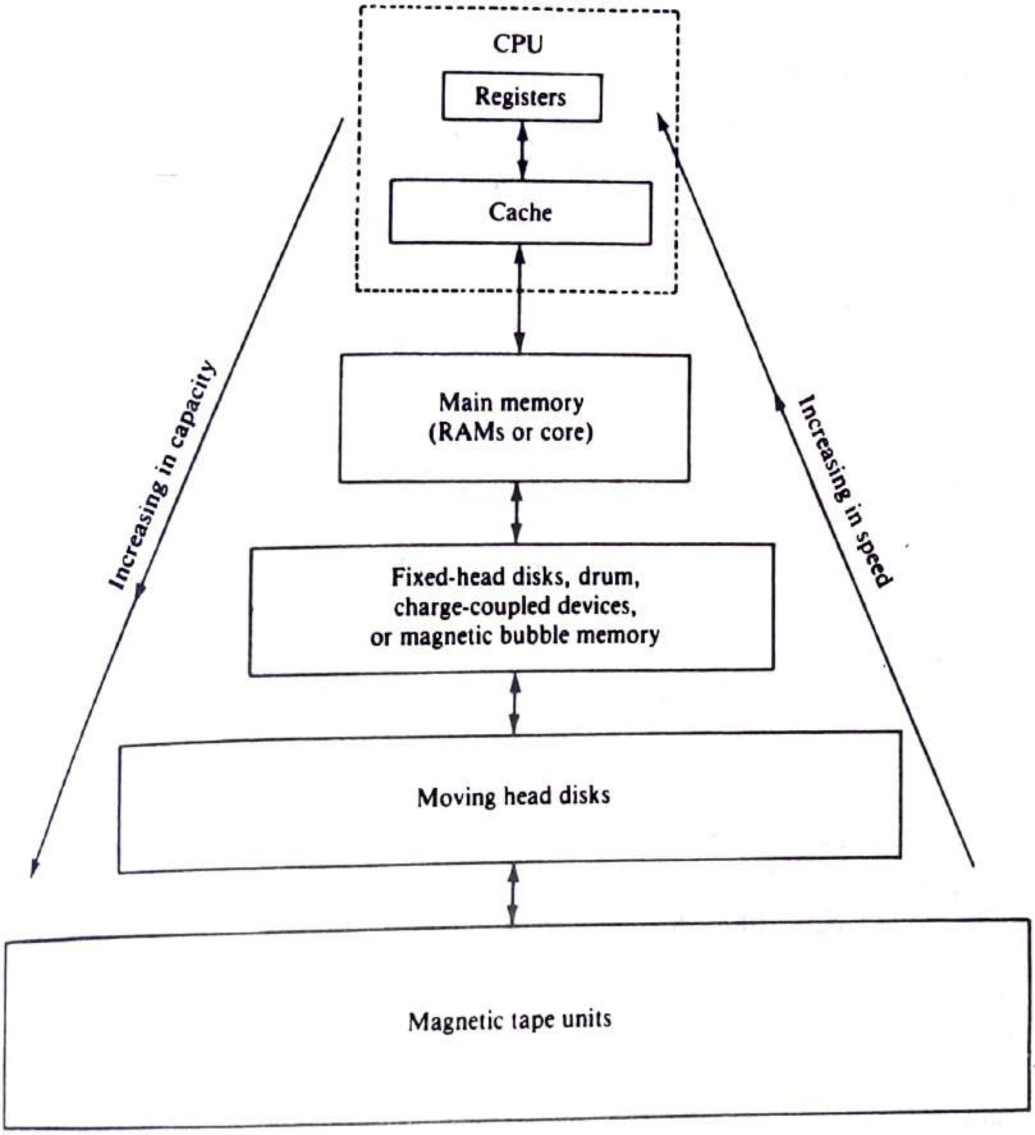


Figure 1.6 The classical memory hierarchy.

Section 6.2.4, along with the description of the Burroughs Scientific Processor (1978). Multiprocessor memory and cache coherence problems will be treated in Section 7.3. All these techniques are intended to broaden the memory bandwidth to match that of the CPU.

1.2.3 Balancing of Subsystem Bandwidth

In general, the CPU is the fastest unit in a computer, with a processor cycle t_p of tens of nanoseconds; the main memory has a cycle time t_m of hundreds of nanoseconds; and the I/O devices are the slowest with an average access time t_d of a few milliseconds. It is thus observed that

$$t_d > t_m > t_p \tag{1.1}$$

For example, the IBM 370/168 has $t_d = 5$ ms (disk), $t_m = 320$ ns, and $t_p = 80$ ns. With these speed gaps between the subsystems, we need to match their processing bandwidths in order to avoid a system bottleneck problem.

The *bandwidth* of a system is defined as the number of operations performed per unit time. In the case of a main memory system, the memory bandwidth is measured by the number of memory words that can be accessed (either fetch or store) per unit time. Let W be the number of words delivered per memory cycle t_m. Then the maximum memory bandwidth B_m is equal to

$$B_m = \frac{W}{t_m} \qquad \text{(words/s or bytes/s)} \tag{1.2}$$

For example, the IBM 3033 uniprocessor has a processor cycle $t_p = 57$ ns. Eight double words (8 bytes each) can be requested from an eight-way interleaved memory system (with eight LSEs in Figure 1.7) per each memory cycle $t_m = 456$ ns. Thus, the maximum memory bandwidth of the 3033 is $B_m = 8 \times 8$ bytes/456 ns $= 140$ megabytes/s. Memory access conflicts may cause delayed access of some of the processor requests. In practice the utilized memory bandwidth B_m^u is usually lower than B_m; that is, $B_m^u \leq B_m$. A rough measure of B_m^u has been suggested as

$$B_m^u \doteq \frac{B_m}{\sqrt{M}} \tag{1.3}$$

where M is the number of interleaved memory modules in the memory system (to be described in Section 3.1.4). For the IBM 3033 uniprocessor, we thus have an approximate $B_m^u = 140/\sqrt{8} = 49.5$ megabytes/s.

For external memory and I/O devices, the concept of bandwidth is more involved because of the sequential-access nature of magnetic disks and tape units. Considering the latencies and rotational delays, the data transfer rate may vary. In general, we refer to the average data transfer rate B_d as the bandwidth of a disk unit. A typical modern disk may have a data rate of 1 megabyte/s. With multiple disk drives, the data rate can increase to 10 megabytes/s, say for 10 drives per channel controller. A modern magnetic tape unit has a data transfer rate around 1.5 megabytes/s. Other peripheral devices, like line printers, readers/punch, and CRT terminals, are much slower due to mechanical motions.

The bandwidth of a processor is measured as the maximum CPU computation rate B_p, as in 160 megaflops for the Cray-1 and 12.5 *million instructions per second* (MIPS) for IBM 370/168. These are all peak values obtained by $1/t_p = 1/12.5$ ns and 1/80 ns respectively. In practice, the utilized CPU rate is $B_p^u \leq B_p$. The utilized CPU rate B_p^u is based on measuring the number of output results (in words) per second:

$$B_p^u = \frac{R_w}{T_p} \qquad \text{(words/s)} \tag{1.4}$$

where R_w is the number of word results and T_p is the total CPU time required to generate the R_w results. For a machine with variable word length, the rate will vary. For example, the CDC Cyber-205 has a peak CPU rate of 200 megaflops for

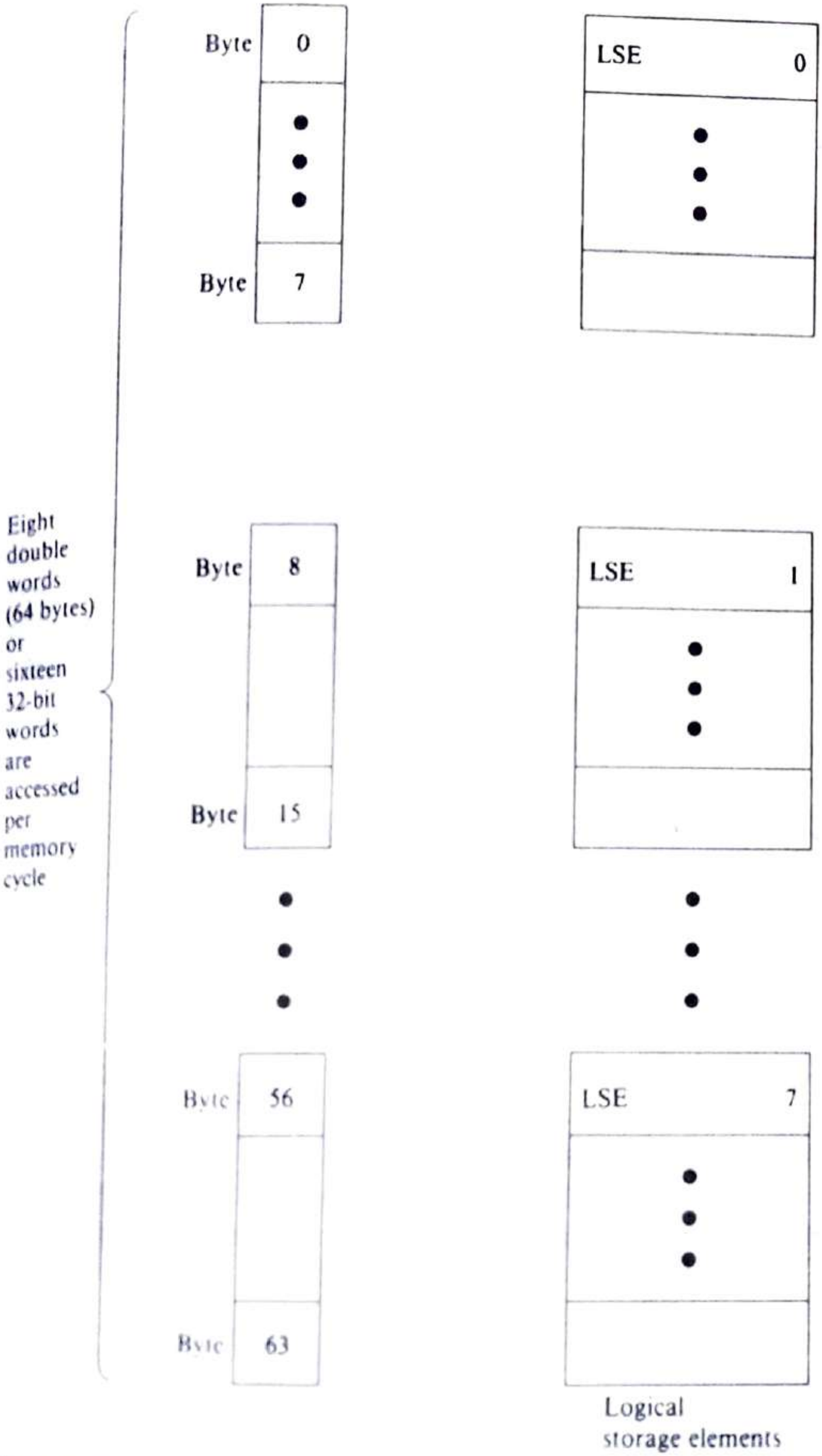

Byte 0
Byte 7
LSE 0
Eight double words (64 bytes) or sixteen 32-bit words are accessed per memory cycle
Byte 8
Byte 15
LSE 1
Byte 56
Byte 63
LSE 7
Logical storage elements

Figure 1.7 The interlea ed memory structure in IBM 3033 uniprocessor.

32-bit results and only 100 megaflops for 64-bit results (one vector processor is assumed).

Based on current technology (1983), the following relationships have been observed between the bandwidths of the major subsystems in a high-performance uniprocessor:

$$B_m \geq B_m^u \geq B_p \geq B_p^u > B_d \tag{1.5}$$

This implies that the main memory has the highest bandwidth, since it must be updated by both the CPU and the I/O devices, as illustrated in Figure 1.8. Due to the unbalanced speeds (Eq. 1.1), we need to match the processing power of the three subsystems. Two major approaches are described below.

Bandwidth balancing between CPU and memory The speed gap between the CPU and the main memory can be closed up by using fast cache memory between them. The cache should have an access time $t_c = t_p$. A block of memory words is moved from the main memory into the cache (such as 16 words/block for the IBM 3033) so that immediate instructions/data can be available most of the time from the cache. The cache serves as a data/instruction buffer. Detailed descriptions of cache memories will be given in Sections 2.4 and 7.3

Bandwidth balancing between memory and I/O devices Input-output channels with different speeds can be used between the slow I/O devices and the main memory. These I/O channels perform buffering and multiplexing functions to transfer the data from multiple disks into the main memory by stealing cycles from the CPU. Furthermore, *intelligent disk controllers* or *database machines* can be used to filter out the irrelevant data just off the tracks of the disk. This filtering will alleviate the I/O channel saturation problem. The combined buffering, multiplexing, and filtering operations thus can provide a faster, more effective data transfer rate, matching that of the memory.

In the ideal case, we wish to achieve a totally balanced system, in which the entire memory bandwidth matches the bandwidth sum of the processor and I/O devices; that is,

$$B_p^u + B_d = B_m^u \tag{1.6}$$

where $B_p^u = B_p$ and $B_m^u = B_m$ are both maximized. Achieving this total balance requires tremendous hardware and software supports beyond any of the existing systems.

1.2.4 Multiprogramming and Time Sharing

Even when there is only one CPU in a uniprocessor system, we can still achieve a high degree of resource sharing among many user programs. We will briefly review the concepts of *multiprogramming* and *time sharing* in this subsection. These are software approaches to achieve concurrency in a uniprocessor system. The

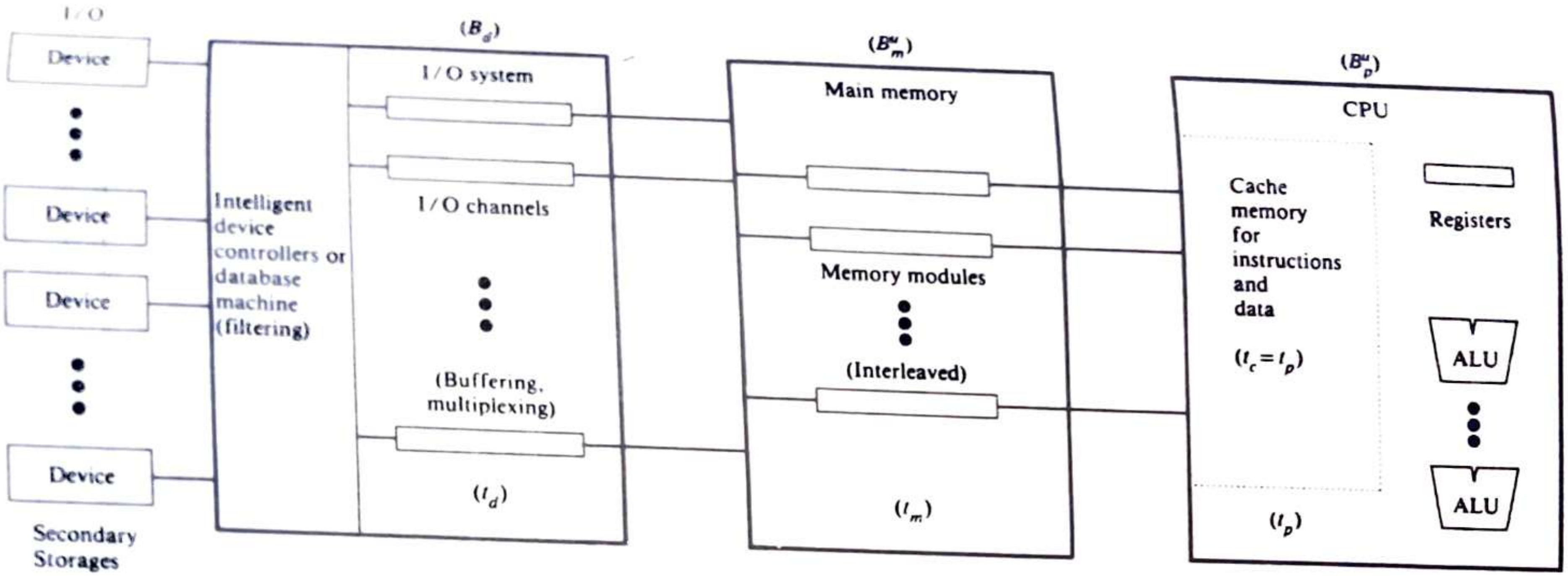


Figure 1.8 **Bandwidth balancing mechanisms between CPU, memory, and I/O subsystem in a uniprocessor computer.**

conventional batch processing is illustrated by the sequential execution in Figure 1.9*a*. We use the notation *i*, *c*, and *o* to represent the *input*, *compute*, and *output* operations, respectively.

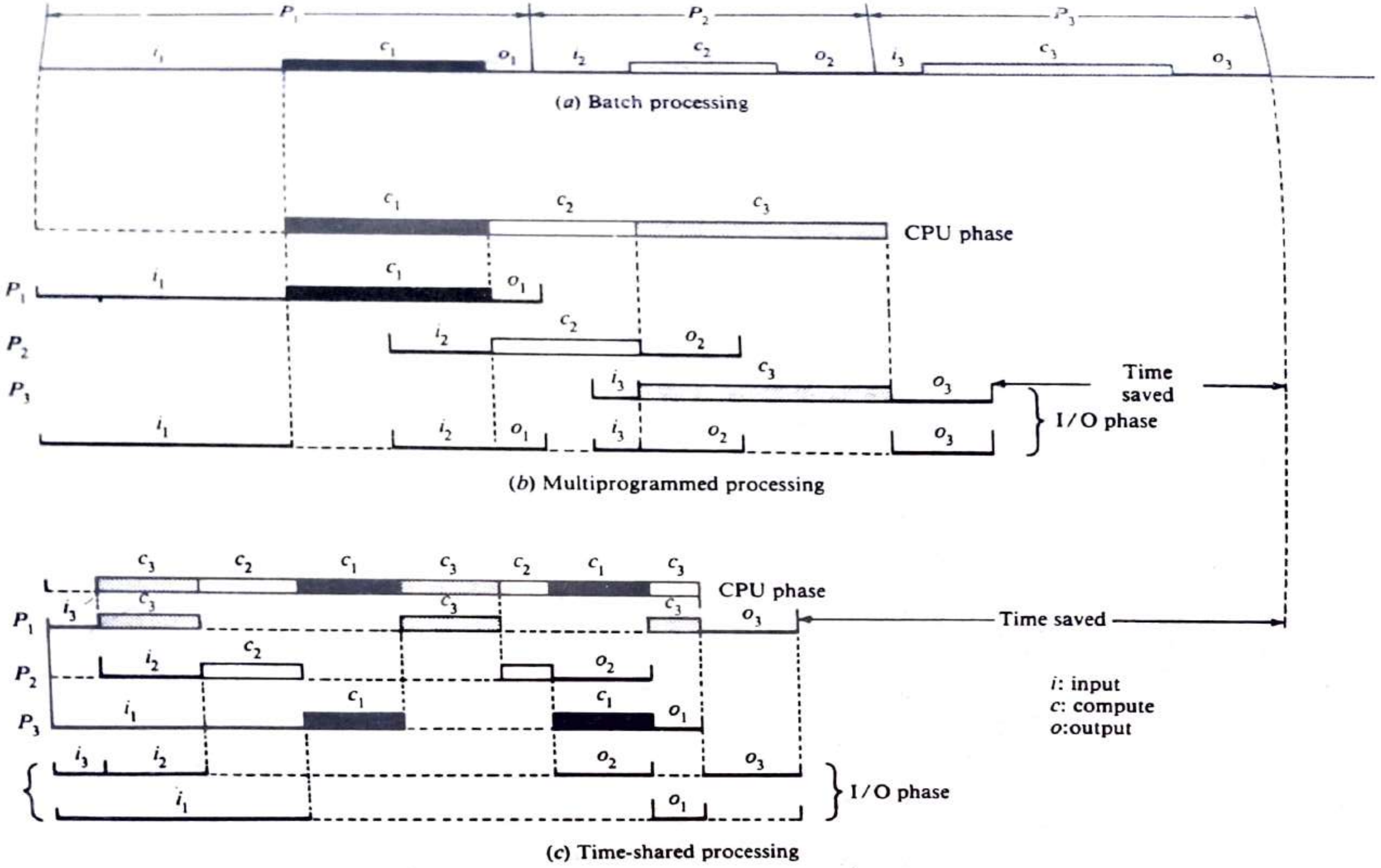
Multiprogramming Within the same time interval, there may be multiple processes active in a computer, competing for memory, I/O, and CPU resources. We are aware of the fact that some computer programs are *CPU-bound* (computation intensive), and some are *I/O-bound* (input-output intensive). We can mix the execution of various types of programs in the computer to balance bandwidths among the various functional units. The program interleaving is intended to promote better resource utilization through overlapping I/O and CPU operations.

As illustrated in Figure 1.9*b*, whenever a process P_1 is tied up with I/O operations, the system scheduler can switch the CPU to process P_2. This allows the simultaneous execution of several programs in the system. When P_2 is done, the CPU can be switched to P_3. Note the overlapped I/O and CPU operations and thé CPU wait time are greatly reduced. This interleaving of CPU and I/O operations among several programs is called *multiprogramming*. The programs can be mixed across the boundary of user tasks and system processes, in either a monoprogramming or a multiprogramming environment. The total execution time is reduced with multiprogramming. The processes $P_1, P_2, \ldots$ may belong to the same or different programs.

Time sharing Multiprogramming on a uniprocessor is centered around the sharing of the CPU by many programs. Sometimes a high-priority program may occupy the CPU for too long to allow others to share. This problem can be overcome by using a *time-sharing* operating system. The concept extends from multiprogramming by assigning fixed or variable *time slices* to multiple programs. In other words, equal opportunities are given to all programs competing for the use of the CPU. This concept is illustrated in Figure 1.9*c*. The execution time saved with time sharing may be greater than with either batch or multiprogram processing modes.

The time-sharing use of the CPU by multiple programs in a uniprocessor computer creates the concept of *virtual processors*. Time sharing is particularly effective when applied to a computer system connected to many interactive terminals. Each user at a terminal can interact with the computer on an instantaneous basis. Each user thinks that he or she is the sole user of the system, because the response is so fast (waiting time between time slices is not recognizable by humans). Time sharing is indispensable to the development of real-time computer systems.

Time sharing.was first developed for a uniprocessor system. The concept can be extended to designing interactive time-sharing multiprocessor systems. Of course, the time sharing on multiprocessors is much more complicated. We will discuss the operating system design considerations for multiprocessor systems in Chapters 7, 8, and 9. The performance of either a uniprocessor or a multiprocessor system depends heavily on the capability of the operating system. After all, the major function of an operating system is to optimize the resource allocation and management, which often leads to high performance.



(a) Batch processing

(b) Multiprogrammed processing

(c) Time-shared processing

Figure 1.9 Operating system approaches to achieve parallel processing in a uniprocessor computer.

1.3 PARALLEL COMPUTER STRUCTURES

Parallel computers are those systems that emphasize parallel processing. The basic architectural features of parallel computers are introduced below. We divide parallel computers into three architectural configurations:

- Pipeline computers
- Array processors
- Multiprocessor systems

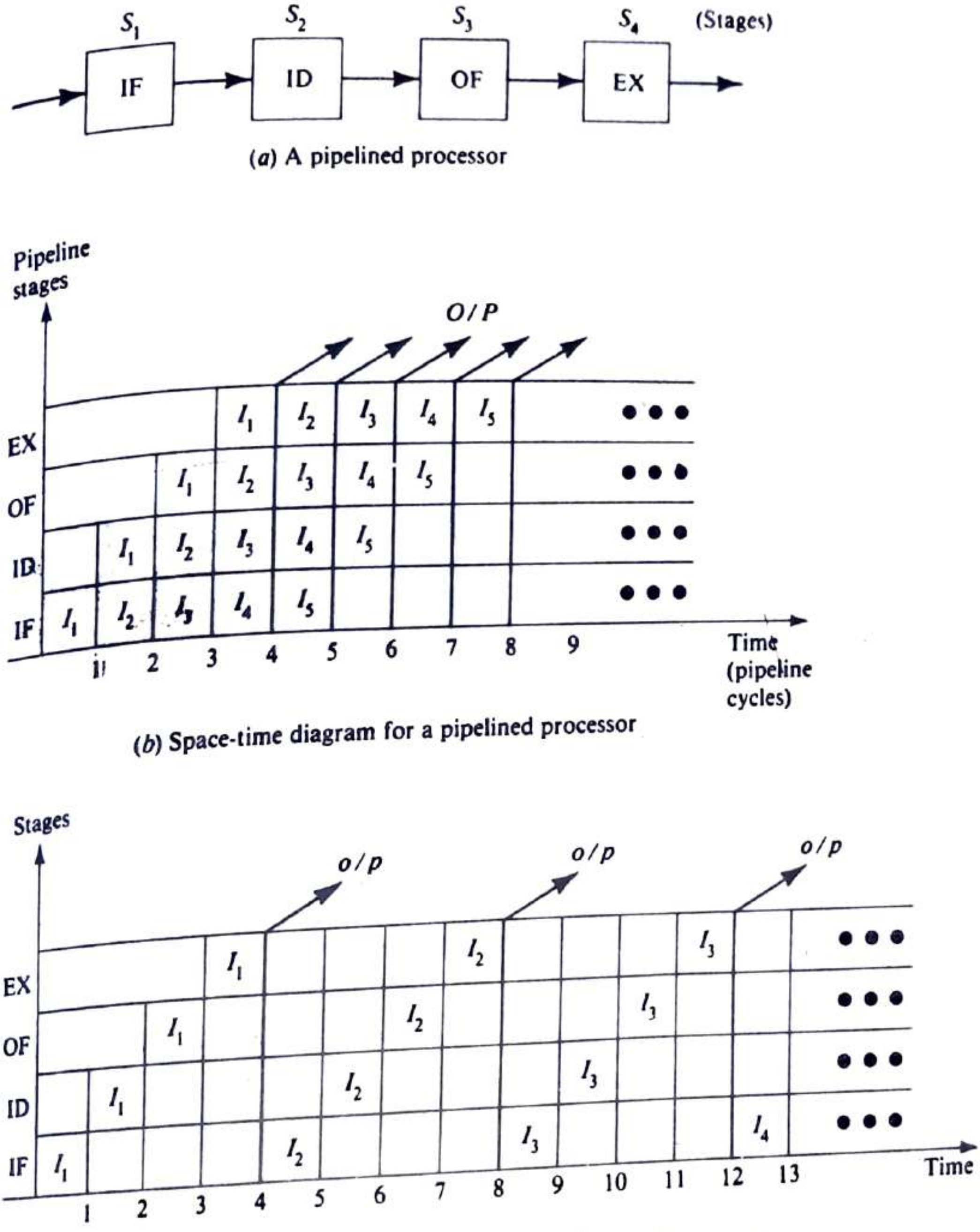
A pipeline computer performs overlapped computations to exploit *temporal parallelism.* An array processor uses multiple synchronized arithmetic logic units to achieve *spatial parallelism.* A multiprocessor system achieves *asynchronous parallelism* through a set of interactive processors with shared resources (memories, database, etc.). These three parallel approaches to computer system design are not mutually exclusive. In fact, most existing computers are now pipelined, and some of them assume also an "array" or a "multiprocessor" structure. The fundamental difference between an array processor and a multiprocessor system is that the processing elements in an array processor operate synchronously but processors in a multiprocessor system may operate asynchronously.

New computing concepts to be introduced in this section include the *data flow computers* and some *VLSI algorithmic processors.* All these new approaches demand extensive hardware to achieve parallelism. The rapid progress in the VLSI technology has made these new approaches possible.

1.3.1 Pipeline Computers

Normally, the process of executing an instruction in a digital computer involves four major steps: *instruction fetch* (IF) from the main memory; *instruction decoding* (ID), identifying the operation to be performed; *operand fetch* (OF), if needed in the execution; and then *execution* (EX) of the decoded arithmetic logic operation. In a nonpipelined computer, these four steps must be completed before the next instruction can be issued. In a pipelined computer, successive instructions are executed in an overlapped fashion, as illustrated in Figure 1.10. Four pipeline stages, IF, ID, OF, and EX, are arranged into a linear cascade. The two space-time diagrams show the difference between overlapped instruction execution and sequentially nonoverlapped execution.

An *instruction cycle* consists of multiple pipeline cycles. A pipeline cycle can be set equal to the delay of the slowest stage. The flow of data (input operands, intermediate results, and output results) from stage to stage is triggered by a common clock of the pipeline. In other words, the operation of all stages is synchronized under a common clock control. Interface latches are used between adjacent segments to hold the intermediate results. For the nonpipelined (non-overlapped) computer, it takes four pipeline cycles to complete one instruction. Once a pipeline is filled up, an output result is produced from the pipeline on each



(*a*) A pipelined processor

(*b*) Space-time diagram for a pipelined processor

(*c*) Space-time diagram for a nonpipelined processor

Figure 1.10 Basic concepts of pipelined processor and overlapped instruction execution.

cycle. The instruction cycle has been effectively reduced to one-fourth of the original cycle time by such overlapped execution.

Theoretically, a *k*-stage linear pipeline processor could be at most *k* times faster. We will prove this in Chapter 3. However, due to memory conflicts, data dependency, branch and interrupts, this ideal speedup may not be achieved for out-of-sequence computations. What has been described so far is the *instruction pipeline*. For some CPU-bound instructions, the execution phase can be further partitioned into a multiple-stage arithmetic logic pipeline, as for sophisticated floating-point operations. Some main issues in designing a pipeline computer include job sequencing, collision prevention, congestion control, branch handling, reconfiguration, and hazard resolution. We will learn how to cope with each of these problems later.

Due to the overlapped instruction and arithmetic execution, it is obvious that pipeline machines are better tuned to perform the same operations repeatedly through the pipeline. Whenever there is a change of operation, say from *add* to *multiply*, the arithmetic pipeline must be drained and reconfigured, which will cause extra time delays. Therefore, pipeline computers are more attractive for vector processing, where component operations may be repeated many times. Most existing pipeline computers emphasize vector processing. We will study basic vector processing requirements in Chapter 3. Various vectorization methods will be presented in Chapter 4, after learning the structure and capability of commercially available pipeline supercomputers and attached processors.

A typical pipeline computer is conceptually depicted in Figure 1.11. This architecture is very similar to several commercial machines like Cray-1 and VP-200, to be described in Chapter 4. Both scalar arithmetic pipelines and vector arithmetic pipelines are provided. The instruction preprocessing unit is itself pipelined with three stages shown. The OF stage consists of two independent stages, one for fetching scalar operands and the other for vector operand fetch. The scalar registers are fewer in quantity than the vector registers because each vector register implies a whole set of component registers. For example, a vector register in Cray-1 contains 64 component registers, each of which is 64 bits wide. Each vector register in Cray-1 requires 4096 flip-flops. Both scalar and vector data could appear in fixed-point or floating-point format. This means different pipelines can be dedicated to different arithmetic logic functions with different data formats. The scalar arithmetic pipelines differ from the vector arithmetic pipelines in structure and control strategies. Modern vector processors are usually augmented with a powerful scalar processor to handle a mixture of vector and scalar instructions.

Pipelined computers to be studied in Chapter 4 include the early vector processors, Control Data's Star-100 and Texas Instruments' Advanced Scientific Computer (ASC); the attached pipeline processors, AP-120B and FPS-164 by Floating Point Systems, Datawest MATP, and IBM 3838; and recent vector processors, Cray-1, Cyber-205, and Fujitsu VP-200. Vectorization methods to be studied include resource reservation, pipeline chaining, vector segmentation, vectorizing compiler design, and optimization of compilers for vector processing. A performance evaluation model for pipeline processors will also be presented.

1.3.2 Array Computers

An *array processor* is a synchronous parallel computer with multiple arithmetic logic units, called *processing elements* (PE), that can operate in parallel in a lockstep fashion. By replication of ALUs, one can achieve the spatial parallelism. The PEs are synchronized to perform the same function at the same time. An appropriate data-routing mechanism must be established among the PEs. A typical

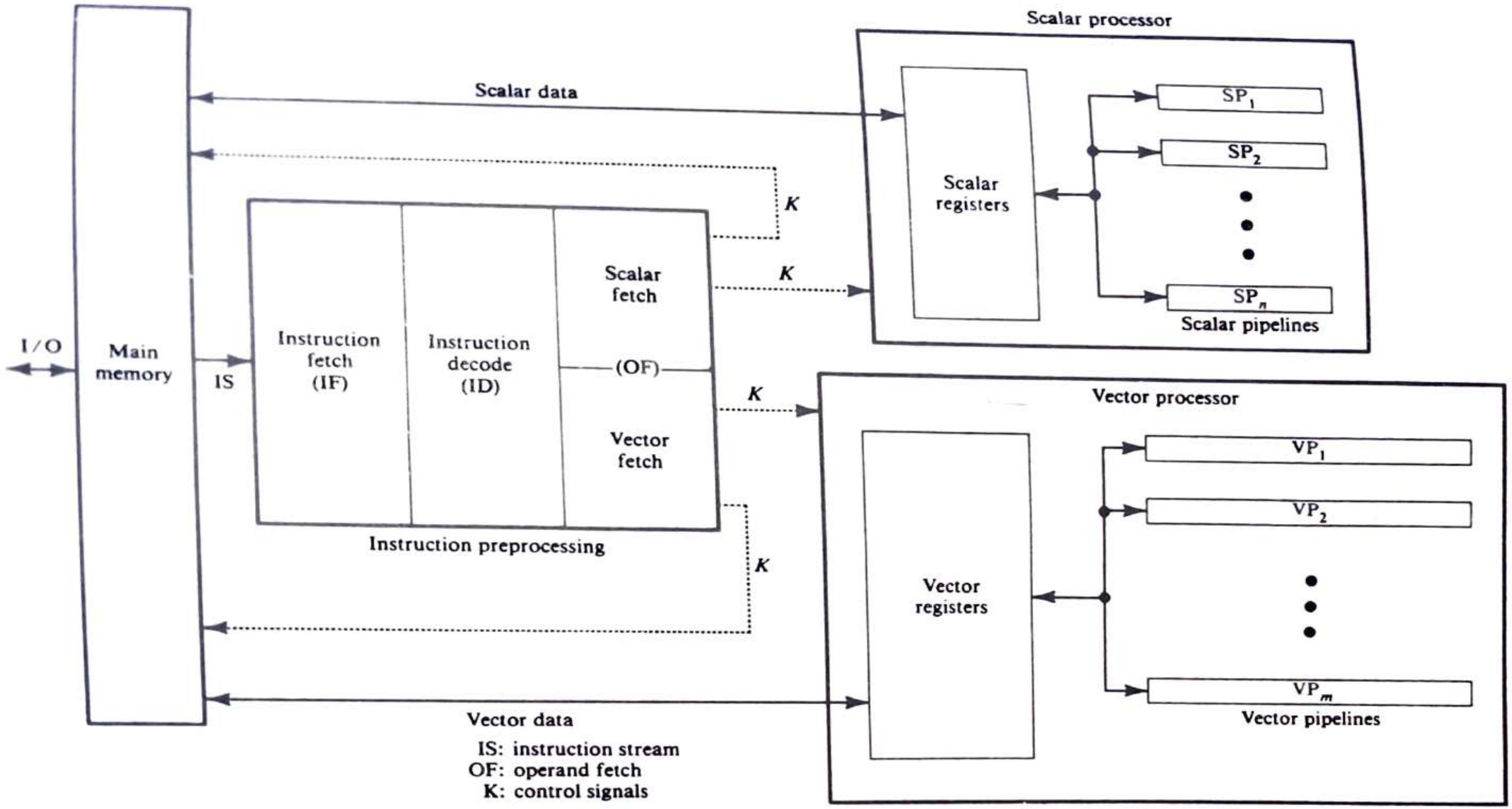


Figure 1.11 Functional structure of a modern pipeline computer with scalar and vector capabilities.

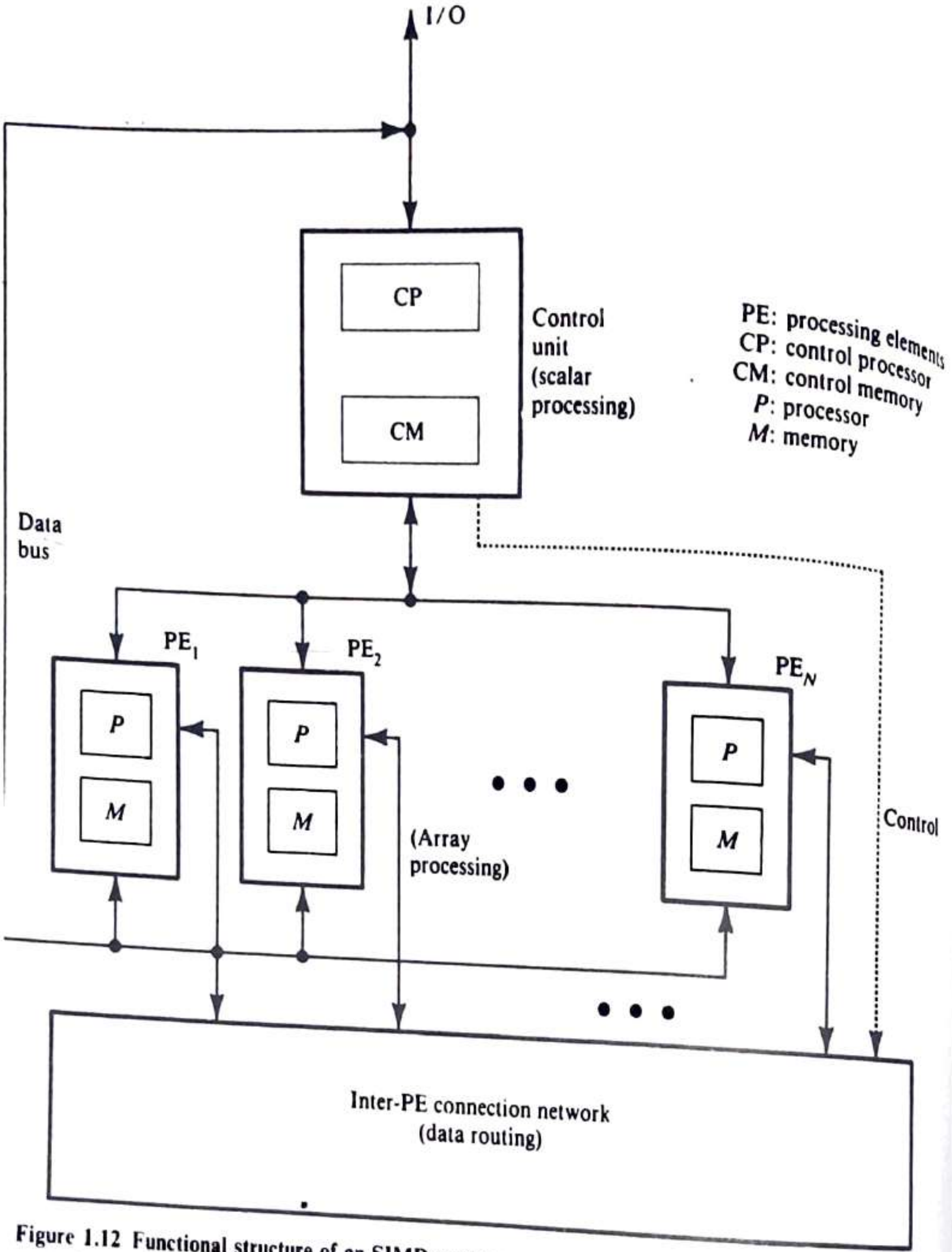


Figure 1.12 Functional structure of an SIMD array processor with concurrent scalar processing in the control unit.

array processor is depicted in Figure 1.12. Scalar and control-type instructions are directly executed in the *control unit* (CU). Each PE consists of an ALU with registers and a local memory. The PEs are interconnected by a data-routing network. The interconnection pattern to be established for specific computation is under program control from the CU. Vector instructions are broadcast to the PEs for distributed execution over different component operands fetched directly from the local memories. Instruction fetch (from local memories or from the control memory) and decode is done by the control unit. The PEs are passive devices without instruction decoding capabilities.

Various interconnection structures for a set of PEs will be studied in Chapter 5. Both recirculating networks and multistage networks will be covered. *Associative memory*, which is content addressable, will also be treated there in the context of parallel processing. Array processors designed with associative memories are called *associative processors*. Parallel algorithms on array processors will be given for matrix multiplication, merge sort, and fast Fourier transform (FFT). A performance evaluation of the array processor will be presented, with emphasis on resource optimization.

Modern array processors will be described in Chapter 6. Different array processors may use different interconnection networks among the PEs. For example, Illiac-IV uses a mesh-structured network and Burroughs Scientific Processor (BSP) uses a crossbar network. In addition to Illiac-IV and BSP, we will study a bit-slice array processor called a *massively parallel processor* (MPP). Array processors are much more difficult to program than pipeline machines. We will study various performance enhancement methods for array processors, including the use of skewed memory allocation, language extensions for vector-array processing, and possible future architectural improvements.

1.3.3 Multiprocessor Systems

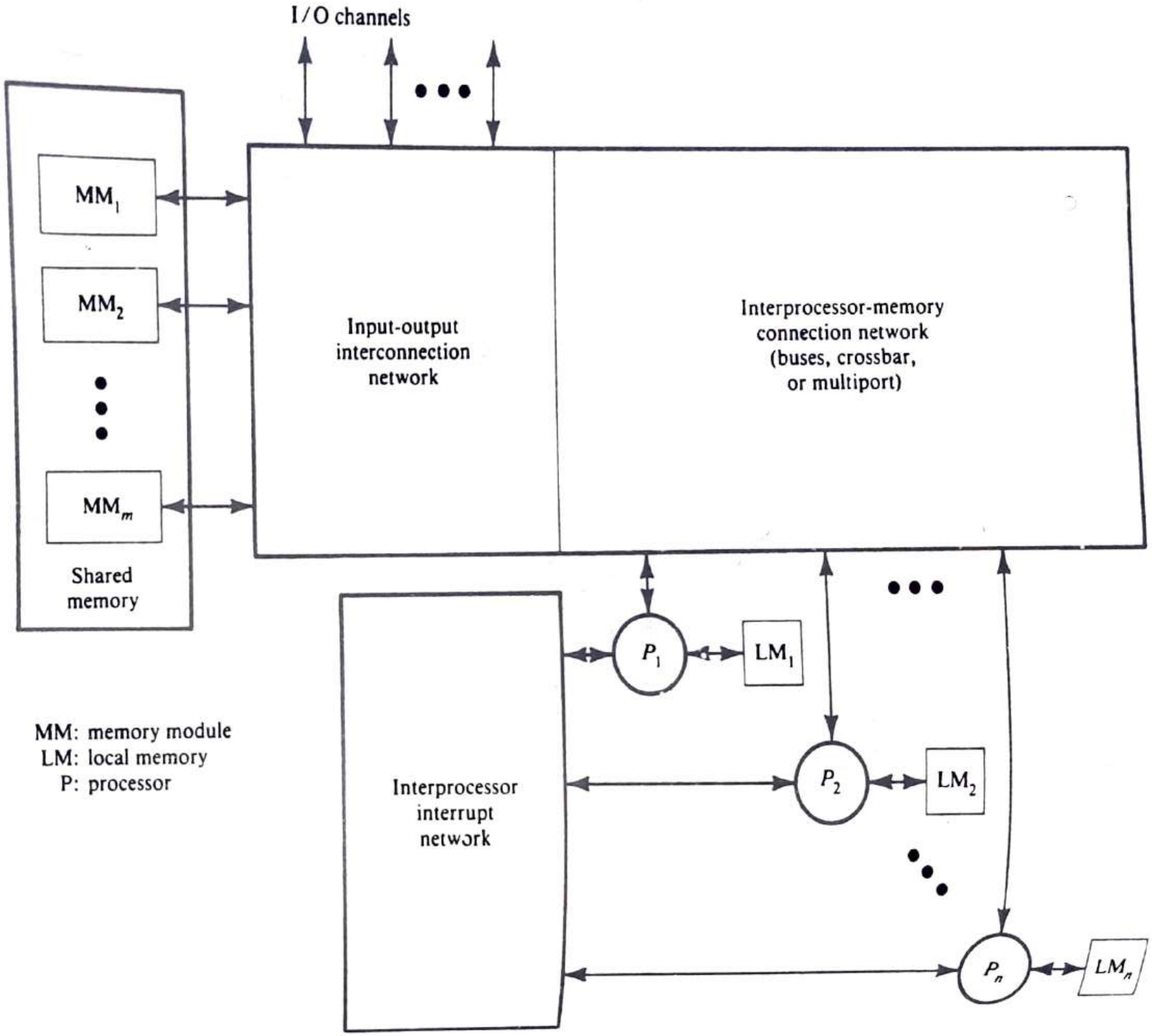
Research and development of multiprocessor systems are aimed at improving throughput, reliability, flexibility, and availability. A basic multiprocessor organization is conceptually depicted in Figure 1.13. The system contains two or more processors of approximately comparable capabilities. All processors share access to common sets of memory modules, I/O channels, and peripheral devices. Most importantly, the entire system must be controlled by a single integrated operating system providing interactions between processors and their programs at various levels. Besides the shared memories and I/O devices, each processor has its own local memory and private devices. Interprocessor communications can be done through the shared memories or through an interrupt network.

Multiprocessor hardware system organization is determined primarily by the interconnection structure to be used between the memories and processors (and between memories and I/O channels, if needed). Three different interconnections have been practiced in the past:

- Time-shared common bus
- Crossbar switch network
- Multiport memories

These organizations and their possible extensions for multiprocessor systems will be described in detail in Chapter 7. Techniques for exploiting concurrency in multiprocessors will be studied, including the development of some parallel language features and the possible detection of parallelism in user programs.

Special memory organization for multiprocessors will be treated in Section 7.3. We will cover hierarchical virtual memory, cache structures, parallel memories,


I/O channels
Input-output interconnection network
Interprocessor-memory connection network (buses, crossbar, or multiport)
MM_1
MM_2
MM_m
Shared memory
Interprocessor interrupt network
P_1
LM_1
P_2
LM_2
P_n
LM_n
MM: memory module
LM: local memory
P: processor

paging, and various memory management issues. Multiprocessor operating systems will also be studied in Chapter 8. Important topics include protection schemes, system deadlock resolution methods, interprocess communication mechanisms, and various multiple processor scheduling strategies. Parallel algorithms for multiprocessors will also be studied. Both synchronous and asynchronous algorithms will be specified and evaluated.

We will present several exploratory and commercial multiprocessor systems in Chapter 9, including the C.mmp system and Cm* system developed at Carnegie Mellon University, the S-1 multiprocessor system developed at the Lawrence Livermore National Laboratory, the IBM System 370/Model 168 MP system, the IBM 3081, the Univac 1100/80 and 90 MP, the Tandem multiprocessor, Denelcor HEP system, and the Cray X-MP and Cray-2 systems.

What we have discussed so far are *centralized* computing systems, in which all hardware-software resources are housed in the same computing center with negligible communication delays among subsystems. The continuing decline of computer hardware and communication costs has made possible the decentralization of hardware, controls, and databases in a computer system. Claims made for *distributed processing* systems include fast response, high availability, graceful degradation, resource sharing, high adaptability to changes in work load, and better expandability. Distributed computing is being widely practiced in banking institutions, airline companies, government services, nationwide dealership, and chain department stores. Computer networks and distributed processing are beyond the scope of this book.

1.3.4 Performance of Parallel Computers

The speedup that can be achieved by a parallel computer with n identical processors working concurrently on a single problem is at most n times faster than a single processor. In practice, the speedup is much less, since some processors are idle at a given time because of conflicts over memory access or communication paths, inefficient algorithms for exploiting the natural concurrency in the computing problem, or many other reasons to be discussed in subsequent chapters. Figure 1.14 shows the various estimates of the actual speedup, ranging from a lower-bound $\log_2 n$ to an upper-bound $n/\ln n$.

The lower-bound $\log_2 n$ is known as the *Minsky's conjecture*. Most commercial multiprocessor systems have from $n = 2$ to $n = 4$ processors. Exploratory research multiprocessors have challenged $n = 16$ processors in the C.mmp and S-1 systems. Using Minsky's conjecture, only a speedup of 2 to 4 can be expected from existing multiprocessors with 4 to 16 processors. This sounds rather pessimistic. A more optimistic speedup estimate is upper bounded by $n/\ln n$ as derived below.

Consider a computing problem, which can be executed by a uniprocessor in unit time, $T_1 = 1$. Let f_i be the probability of assigning the same problem to i processors working equally with an average load $d_i = 1/i$ per processor. Furthermore, assume equal probability of each operating mode using i processors, that is $f_i = 1/n$, for n operating modes: $i = 1, 2, \ldots, n$. The average time required to solve

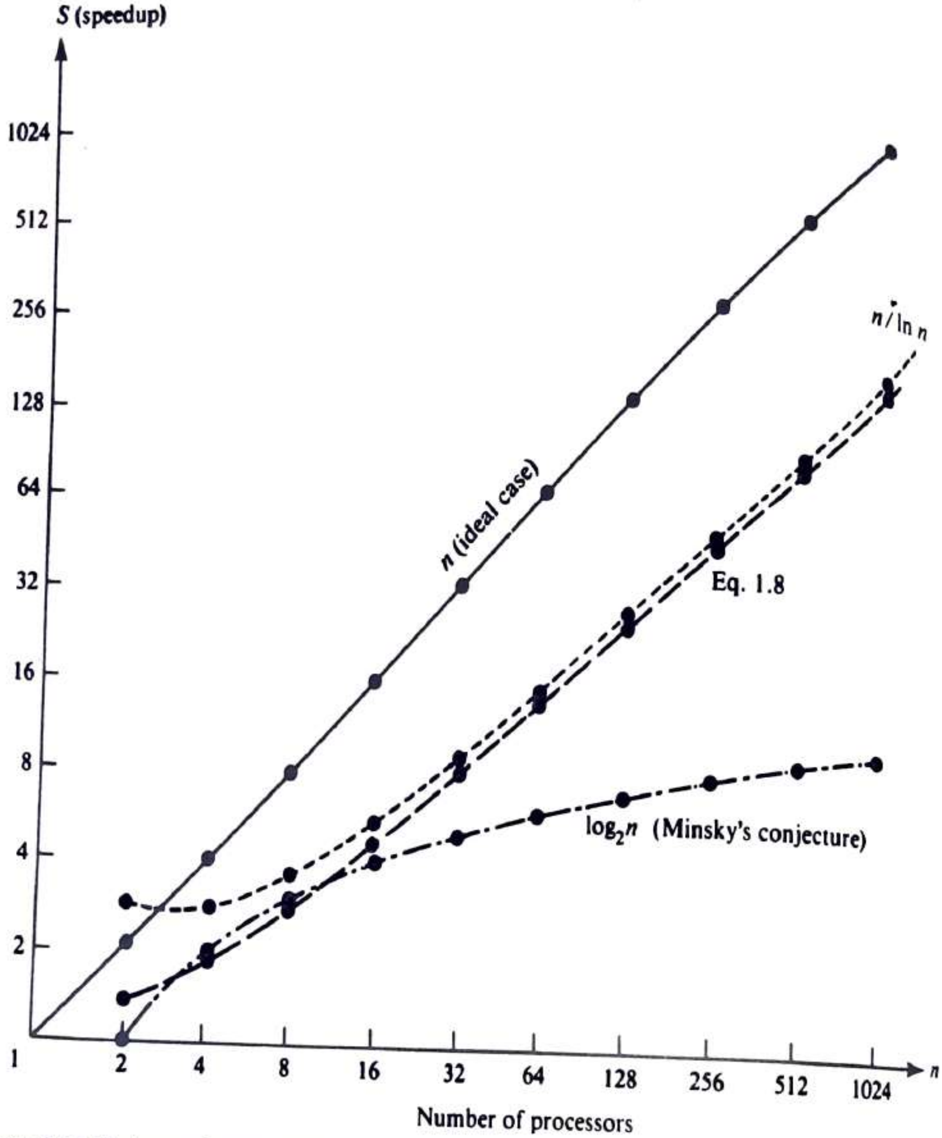


Figure 1.14 **Various estimates of the speedup of an *n*-processor system over a single processor.**

the problem on an *n*-processor system is given below, where the summation represents *n* operating modes.

$$T_n = \sum_{i=1}^{n} f_i \cdot d_i = \frac{\sum_{i=1}^{n} \frac{1}{i}}{n} \tag{1.7}$$

The average speedup S is obtained as the ratio of $T_1 = 1$ to T_n; that is,

$$S = \frac{T_1}{T_n} = \frac{n}{\sum_{i=1}^{n} \frac{1}{i}} \leq \frac{n}{\ln n} \tag{1.8}$$

For a given multiprocessor system with 2, 4, 8, or 16 processors, the respective average speedups (using Eq. 1.8) are 1.33, 1.92, 3.08, and 6.93. The speedup obtained

6

Pipelining and Superscalar Techniques

This chapter deals with pipelining and superscalar design in processor development. We begin with a discussion of conventional linear pipelines and analyze their performance. A generalized pipeline model is introduced to include nonlinear interstage connections. Collision-free scheduling techniques are described for performing dynamic functions.

Specific techniques for building instruction pipelines, arithmetic pipelines, and memory-access pipelines are presented. The discussion includes instruction prefetching, internal data forwarding, software interlocking, hardware scoreboarding, hazard avoidance, branch handling, and instruction-issuing techniques. Both static and multifunctional arithmetic pipelines are designed. Superscalar design techniques are studied along with performance analysis.

6.1 LINEAR PIPELINE PROCESSORS

→ a Small [illegible] one of [illegible]

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. In modern computers, linear pipelines are applied for instruction execution, arithmetic computation, and memory-access operations.

→ Spe[illegible]

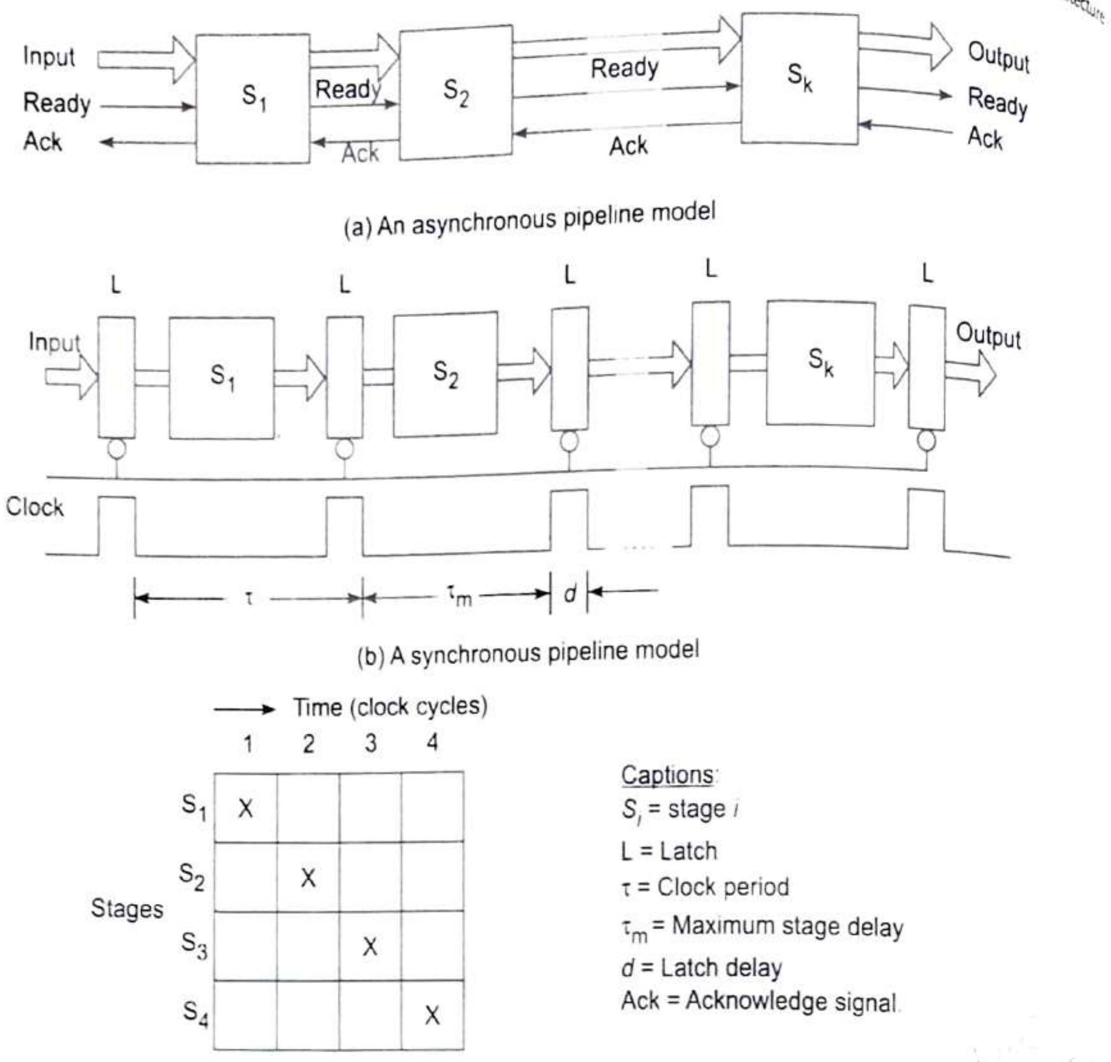
6.1.1 Asynchronous and Synchronous Models

A linear pipeline processor is constructed with k processing stages. External inputs (operands) are fed into the pipeline at the first stage S_1. The processed results are passed from stage S_i to stage S_{i+1}, for all $i = 1, 2, \ldots, k-1$. The final result emerges from the pipeline at the last stage S_k.

Depending on the control of data flow along the pipeline, we model linear pipelines in two categories: *asynchronous* and *syncnronous*.

Asynchronous Model As shown in Fig. 6.1a, data flow between adjacent stages in an asynchronous pipeline is controlled by a handshaking protocol. When stage S_i is ready to transmit, it sends a *ready* signal to stage S_{i+1}. After stage S_{i+1} receives the incoming data, it returns an *acknowledge* signal to S_i.

Asynchronous pipelines are useful in designing communication channels in message-passing multicomputers where pipelined wormhole routing is practiced (see Chapter 9). Asynchronous pipelines may have a variable throughput rate. Different amounts of delay may be experienced in different stages.



(a) An asynchronous pipeline model

(b) A synchronous pipeline model

	1	2	3	4
S_1	X			
S_2		X		
S_3			X	
S_4				X

Captions:
S_i = stage i
L = Latch
τ = Clock period
τ_m = Maximum stage delay
d = Latch delay
Ack = Acknowledge signal.

(c) Reservation table of a four-stage linear pipeline

Fig. 6.1 Two models of linear pipeline units and the corresponding reservation table

Synchronous Model Synchronous pipelines are illustrated in Fig. 6.1b. Clocked latches are used to interface between stages. The latches are made with master-slave flip-flops, which can isolate inputs from outputs. Upon the arrival of a clock pulse, all latches transfer data to the next stage simultaneously.

The pipeline stages are combinational logic circuits. It is desired to have approximately equal delays in all stages. These delays determine the clock period and thus the speed of the pipeline. Unless otherwise specified, only synchronous pipelines are studied in this book.

The utilization pattern of successive stages in a synchronous pipeline is specified by a *reservation table*. For a linear pipeline, the utilization follows the diagonal streamline pattern shown in Fig. 6.1c. This table is essentially a space-time diagram depicting the precedence relationship in using the pipeline stages. For a k-stage linear pipeline, k clock cycles are needed for data to flow through the pipeline.

Successive tasks or operations are initiated one per cycle to enter the pipeline. Once the pipeline is filled up, one result emerges from the pipeline for each additional cycle. This throughput is sustained only if the successive tasks are independent of each other.

6.1.2 Clocking and Timing Control

The *clock cycle* τ of a pipeline is determined below. Let τ_i be the time delay of the circuitry in stage S_i and d the time delay of a latch, as shown in Fig. 6.1b.

Clock Cycle and Throughput Denote the *maximum stage delay* as τ_m, and we can write τ as

$$\tau = \max_i \{\tau_i\}_1^k + d = \boxed{\tau_m + d} \qquad T_m \rightarrow T \qquad (6.1)$$

At the rising edge of the clock pulse, the data is latched to the master flip-flops of each latch register. The clock pulse has a width equal to d. In general, $\tau_m >> d$ by one to two orders of magnitude. This implies that the maximum stage delay τ_m dominates the clock period.

The *pipeline frequency* is defined as the inverse of the clock period:

$$f = \frac{1}{\tau} \qquad (6.2)$$

If one result is expected to come out of the pipeline per cycle, 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. This is because more than one clock cycle has elapsed between successive task initiations.

Clock Skewing Ideally, we expect the clock pulses to arrive at all stages (latches) at the same time. However, due to a problem known as *clock skewing*, the same clock pulse may arrive at different stages with a time offset of s. Let t_{max} be the time delay of the longest logic path within a stage and t_{min} that of the shortest logic path within a stage.

To avoid a race in two successive stages, we must choose $\tau_m \geq t_{max} + s$ and $d \leq t_{min} - s$. These constraints translate into the following bounds on the clock period when clock skew takes effect:

$$d + t_{max} + s \leq \tau \leq \tau_m + t_{min} - s \qquad (6.3)$$

In the ideal case $s = 0$, $t_{max} = \tau_m$, and $t_{min} = d$. Thus, we have $\tau = \tau_m + d$, consistent with the definition in Eq. 6.1 without the effect of clock skewing.

6.1.3 Speedup, Efficiency, and Throughput

Ideally, a linear pipeline of k stages can process n tasks in $k + (n - 1)$ clock cycles, where k cycles are needed to complete the execution of the very first task and the remaining $n - 1$ tasks require $n - 1$ cycles. Thus the total time required is

$$T_k = [k + (n - 1)]\tau \qquad (6.4)$$

where τ is the clock period. Consider an equivalent-function nonpipelined processor which has a *flow-through delay* of $k\tau$. The amount of time it takes to execute n tasks on this nonpipelined processor is $T_1 = nk\tau$.

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

$$S_k = \frac{T_1}{T_k} = \frac{nk\tau}{k\tau + (n-1)\tau} = \frac{nk}{k + (n-1)} \qquad (6.5)$$

Note 6.1 *Pipelined versus non-pipelined processors*

If each pipeline stage has a stage delay of τ, then clearly an instruction passing through k pipeline stages in a processor sees a total latency of $k\tau$. Now suppose we also have a non-pipelined processor for the same instruction set, using the same technology. This non-pipelined processor need not present a latency of $k\tau$ to every instruction, because it does not have k separate stages for an instruction to pass through. Since the non-pipelined processor would have a more compact hardware design, we can expect that the average latency seen by instructions on this processor will be smaller than $k\tau$.

In other words, the advantage of a pipelined processor lies in its instruction throughput; in terms of instruction latency, the non-pipelined version can in fact be expected do better. However, for the comparative analysis here, we have assumed that the instruction latency on the non-pipelined version is also $k\tau$. This is a simplification which does not change substantially the conclusion reached.

Example 6.1 Pipeline speedup versus stream length

The maximum speedup is $S_k \rightarrow k$ as $n \rightarrow \infty$. This maximum speedup is very difficult to achieve because of data dependences between successive tasks (instructions), program branches, interrupts, and other factors to be studied in subsequent sections.

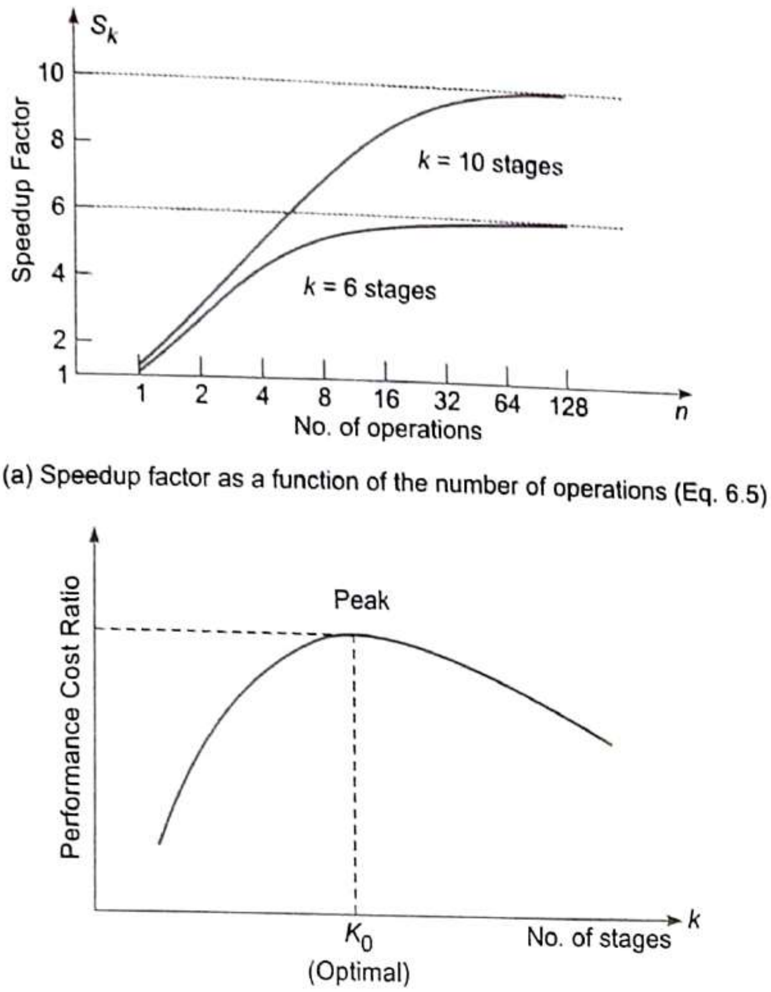
Figure 6.2a plots the speedup factor as a function of n, the number of tasks (operations or instructions) performed by the pipeline. For small values of n, the speedup can be very poor. The smallest value of S_k is 1 when $n = 1$.

The larger the number k of subdivided pipeline stages, the higher the potential speedup performance. When $n = 64$, an eight-stage pipeline has a speedup value of 7.1 and a four-stage pipeline has a speedup of 3.7. However, the number of pipeline stages cannot increase indefinitely due to practical constraints on costs, control complexity, circuit implementation, and packaging limitations. Furthermore, the stream length n also affects the speedup; the longer the better in using a pipeline.

Optimal Number of Stages In practice, most pipelining is staged at the functional level with $2 \leq k \leq 15$. Very few pipelines are designed to exceed 10 stages in real computers. The optimal choice of the number of pipeline stages should be able to maximize the performance/cost ratio for the target processing load.

Let t be the total time required for a nonpipelined sequential program of a given function. To execute the same program on a k-stage pipeline with an equal flow-through delay t, one needs a clock period of $p = t/k + d$, where d is the latch delay. Thus, the pipeline has a maximum throughput of $f = 1/p = 1/(t/k + d)$. The total pipeline cost is roughly estimated by $c + kh$, where c covers the cost of all logic stages and h represents the cost of each latch. A pipeline *performance/cost ratio* (PCR) has been defined by Larson (1973):

$$PCR = \frac{f}{c + kh} = \frac{1}{(t/k + d)(c + kh)} \tag{6.6}$$



(a) Speedup factor as a function of the number of operations (Eq. 6.5)

(b) Optimal number of pipeline stages (Eqs. 6.6 and 6.7)

Fig. 6.2 Speedup factors and the optimal number of pipeline stages for a linear pipeline unit

Figure 6.2b plots the PCR as a function of k. The peak of the PCR curve corresponds to an optimal choice for the number of desired pipeline stages:

$$k_0 = \sqrt{\frac{t \cdot c}{d \cdot h}} \tag{6.7}$$

where t is the total flow-through delay of the pipeline. Thus the total stage cost c, the latch delay d, and the latch cost h must be considered to achieve the optimal value k_0.

Efficiency and Throughput The *efficiency* E_k of a linear k-stage pipeline is defined as

$$E_k = \frac{S_k}{k} = \frac{n}{k + (n-1)} \tag{6.8}$$

Obviously, the efficiency approaches 1 when $n \to \infty$, and a lower bound on E_k is $1/k$ when $n = 1$. The *pipeline throughput* H_k is defined as the number of tasks (operations) performed per unit time:

$$H_k = \frac{n}{[k + (n-1)]\tau} = \frac{nf}{k + (n-1)} \tag{6.9}$$

The *maximum throughput* f occurs when $E_k \to 1$ as $n \to \infty$. This coincides with the speedup definition given in Chapter 3. Note that $H_k = E_k \cdot f = E_k/\tau = S_k/k\tau$. Other relevant factors of instruction pipelines will be discussed in Chapters 12 and 13.

6.2 NONLINEAR PIPELINE PROCESSORS

A *dynamic pipeline* can be reconfigured to perform variable functions at different times. The traditional linear pipelines are static pipelines because they are used to perform fixed functions.

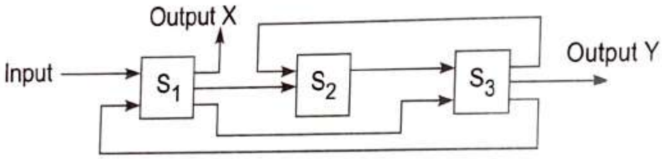
A dynamic pipeline allows feedforward and feedback connections in addition to the streamline connections. For this reason, some authors call such a structure a *nonlinear pipeline*.

6.2.1 Reservation and Latency Analysis

In a static pipeline, it is relatively easy to partition a given function into a sequence of linearly ordered subfunctions. However, function partitioning in a dynamic pipeline becomes quite involved because the pipeline stages are interconnected with loops in addition to streamline connections.

A multifunction dynamic pipeline is shown in Fig. 6.3a. This pipeline has three stages. Besides the *streamline connections* from S_1 to S_2 and from S_2 to S_3, there is a *feed forward connection* from S_1 to S_3 and two *feedback connections* from S_3 to S_2 and from S_3 to S_1.

These feedforward and feedback connections make the scheduling of successive events into the pipeline a nontrivial task. With these connections, the output of the pipeline is not necessarily from the last stage. In fact, following different dataflow patterns, one can use the same pipeline to evaluate different functions.



(a) A three-stage pipeline

→ Time

Stages	1	2	3	4	5	6	7	8
S_1	X					X		X
S_2		X		X				
S_3			X		X		X	

(b) Reservation table for function X

→ Time

Stages	1	2	3	4	5	6
S_1	Y				Y	
S_2			Y			
S_3		Y		Y		Y

(c) Reservation table for function Y

Fig. 6.3 A dynamic pipeline with feed forward and feedback connections for two different functions

Reservation Tables The reservation table for a static linear pipeline is trivial in the sense that dataflow follows a linear streamline. The *reservation table* for a dynamic pipeline becomes more interesting because a nonlinear pattern is followed. Given a pipeline configuration, multiple reservation tables can be generated for the evaluation of different functions.

Two reservation tables are given in Figs. 6.3b and 6.3c, corresponding to a function X and a function Y, respectively. Each function evaluation is specified by one reservation table. A static pipeline is specified by a single reservation table. A dynamic pipeline may be specified by more than one reservation table.

Each reservation table displays the time-space flow of data through the pipeline for one function evaluation. Different functions follow different paths through the pipeline.

The number of columns in a reservation table is called the *evaluation time* of a given function. For example, the function X requires eight clock cycles to evaluate, and function Y requires six cycles, as shown in Figs. 6.3b and 6.3c, respectively.

A pipeline *initiation* table corresponds to each function evaluation. All initiations to a static pipeline use the same reservation table. On the other hand, a dynamic pipeline may allow different initiations to follow a mix of reservation tables. The checkmarks in each row of the reservation table correspond to the time instants (cycles) that a particular stage will be used.

There may be multiple checkmarks in a row, which means repeated usage of the same stage in different cycles. Contiguous checkmarks in a row simply imply the extended usage of a stage over more than one cycle. Multiple checkmarks in a column mean that multiple stages need to be used in parallel during a particular clock cycle.

Latency Analysis The number of time units (clock cycles) between two initiations of a pipeline is *the latency* between them. Latency values must be nonnegative integers. A latency of *k* means that two initiations are separated by *k* clock cycles. Any attempt by two or more initiations to use the same pipeline stage at the same time will cause a *collision*.

A collision implies resource conflicts between two initiations in the pipeline. Therefore, all collisions must be avoided in scheduling a sequence of pipeline initiations. Some latencies will cause collisions, and some will not. Latencies that cause collisions are called *forbidden latencies*. In using the pipeline in Fig. 6.3 to evaluate the function X, latencies 2 and 5 are forbidden, as illustrated in Fig. 6.4.

→ Time

Stages	1	2	3	4	5	6	7	8	9	10	11
S_1	X_1		X_2		X_3	X_1	X_4	X_1, X_2		X_2, X_3	
S_2		X_1		X_1, X_2		X_2, X_3		X_3, X_4		X_4	
S_3			X_1		X_1, X_2		X_1, X_2, X_3		X_2, X_3, X_4		

• • •

(a) Collision with scheduling latency 2

→ Time

Stages	1	2	3	4	5	6	7	8	9	10	11
S_1	X_1					X_1, X_2		X_1			
S_2		X_1		X_1			X_2		X_2		
S_3			X_1		X_1		X_1	X_2		X_2	

• • •

(b) Collision with scheduling latency 5

Fig. 6.4 Collisions with forbidden latencies 2 and 5 in using the pipeline in Fig. 6.3 to evaluate the function X

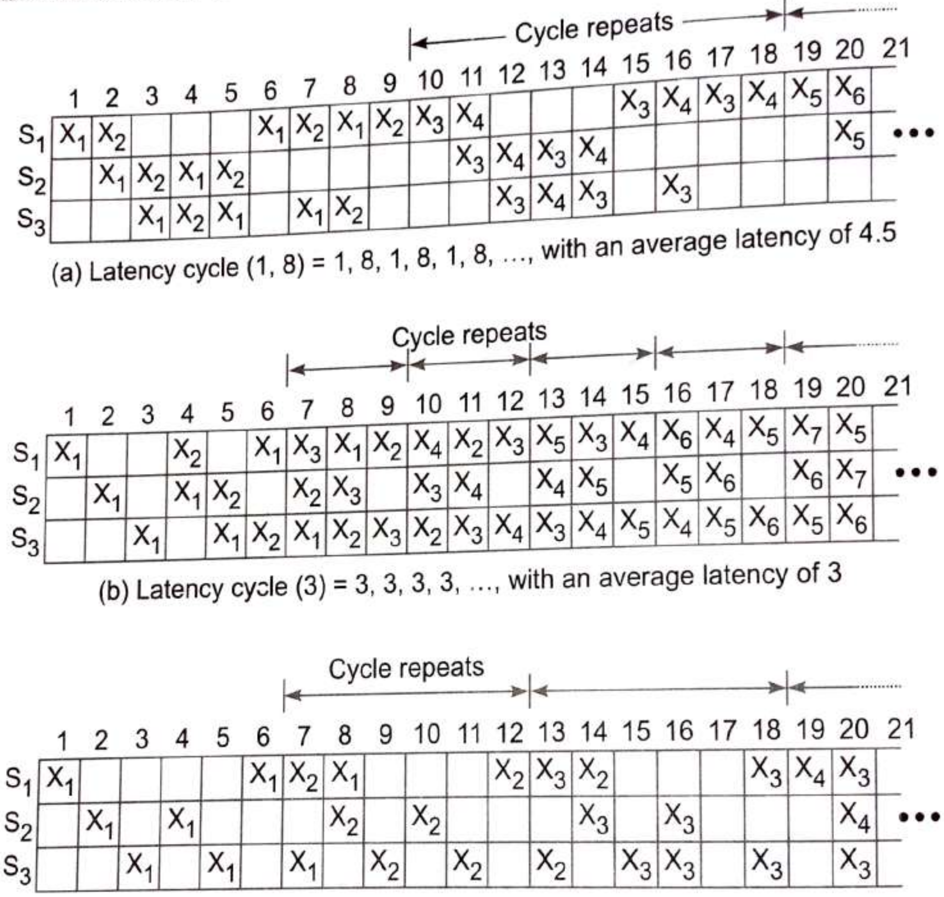
The *i*th initiation is denoted as X_i in Fig. 6.4. With latency 2, initiations X_1 and X_2 collide in stage 2 at time 4. At time 7, these initiations collide in stage 3. Similarly, other collisions are shown at times 5, 6, 8, ..., etc.

The collision patterns for latency 5 are shown in Fig. 6.4b, where X_1 and X_2 are scheduled 5 clock cycles apart. Their first collision occurs at time 6.

To detect a forbidden latency, one needs simply to check the distance between any two checkmarks in the same row of the reservation table. For example, the distance between the first mark and the second mark in row S_1 in Fig. 6.3b is 5, implying that 5 is a forbidden latency.

Similarly, latencies 2, 4, 5, and 7 are all seen to be forbidden from inspecting the same reservation table. From the reservation table in Fig. 6.3c, we discover the forbidden latencies 2 and 4 for function Y. A *latency sequence* is a sequence of permissible nonforbidden latencies between successive task initiations.

A *latency cycle* is a latency sequence which repeats the same subsequence (cycle) indefinitely. Figure 6.5 illustrates latency cycles in using the pipeline in Fig. 6.3 to evaluate the function X without causing a collision. For example, the latency cycle (1, 8) represents the infinite latency sequence 1, 8, 1, 8, 1, 8, ... This implies that successive initiations of new tasks are separated by one cycle and eight cycles alternately.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
S_1	X_1	X_2				X_1	X_2	X_1	X_2	X_3	X_4				X_3	X_4	X_3	X_4	X_5	X_6	
S_2		X_1	X_2	X_1	X_2						X_3	X_4	X_3	X_4						X_5	
S_3			X_1	X_2	X_1		X_1	X_2				X_3	X_4	X_3		X_3					

(a) Latency cycle (1, 8) = 1, 8, 1, 8, 1, 8, …, with an average latency of 4.5

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
S_1	X_1			X_2		X_1	X_3	X_1	X_2	X_4	X_2	X_3	X_5	X_3	X_4	X_6	X_4	X_5	X_7	X_5	
S_2		X_1		X_1	X_2		X_2	X_3		X_3	X_4		X_4	X_5		X_5	X_6		X_6	X_7	
S_3			X_1		X_1	X_2	X_1	X_2	X_3	X_2	X_3	X_4	X_3	X_4	X_5	X_4	X_5	X_6	X_5	X_6	

(b) Latency cycle (3) = 3, 3, 3, 3, …, with an average latency of 3

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
S_1	X_1					X_1	X_2	X_1				X_2	X_3	X_2				X_3	X_4	X_3	
S_2		X_1		X_1				X_2		X_2				X_3		X_3				X_4	
S_3			X_1		X_1		X_1		X_2		X_2		X_2		X_3	X_3		X_3		X_3	

(c) Latency cycle (6) = 6, 6, 6, 6, …, with an average latency of 6

Fig. 6.5 Three valid latency cycles for the evaluation of function X

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. A *constant cycle* is a latency cycle which contains only one latency value. Cycles (3) and (6) in Figs. 6.5b and 6.5c are both constant cycles. The average latency of a constant cycle is simply the latency itself. In the next section, we describe how to obtain these latency cycles systematically.

6.2.2 Collision-Free Scheduling

When scheduling events in a nonlinear pipeline, the main objective is to obtain the shortest average latency between initiations without causing collisions. In what follows, we present a systematic method for achieving such collision-free scheduling.

We study below *collision vectors, state diagrams, single cycles, greedy cycles,* and *minimal average latency* (MAL). This pipeline design theory was originally developed by Davidson (1971) and his students.

Collision Vectors By examining the reservation table, one can distinguish the set of permissible latencies from the set of forbidden latencies. 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. The choice is made in the range $1 \leq p \leq m - 1$.

A permissible latency of $p = 1$ corresponds to the ideal case. In theory, a latency of 1 can always be achieved in a static pipeline which follows a linear (diagonal or streamlined) reservation table as shown in Fig. 6.1c.

The combined set of permissible and forbidden latencies can be easily displayed by a collision vector, which is an m-bit binary vector $C = (C_m C_{m-1} \ldots C_2 C_1)$. The value of $C_i = 1$ if latency i causes a collision and $C_i = 0$ if latency i is permissible. Note that it is always true that $C_m = 1$, corresponding to the maximum forbidden latency.

For the two reservation tables in Fig. 6.3, 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. Similarly, 4 and 2 are forbidden latencies and 3 and 1 are permissible latencies for function Y.

State Diagrams From the above collision vector, one can construct a *state diagram* specifying the permissible state transitions among successive initiations. The collision vector, like C_X above, corresponds to the *initial state* of the pipeline at time 1 and thus is called an *initial collision vector*. Let p be a permissible latency within the range $1 \leq p \leq m - 1$.

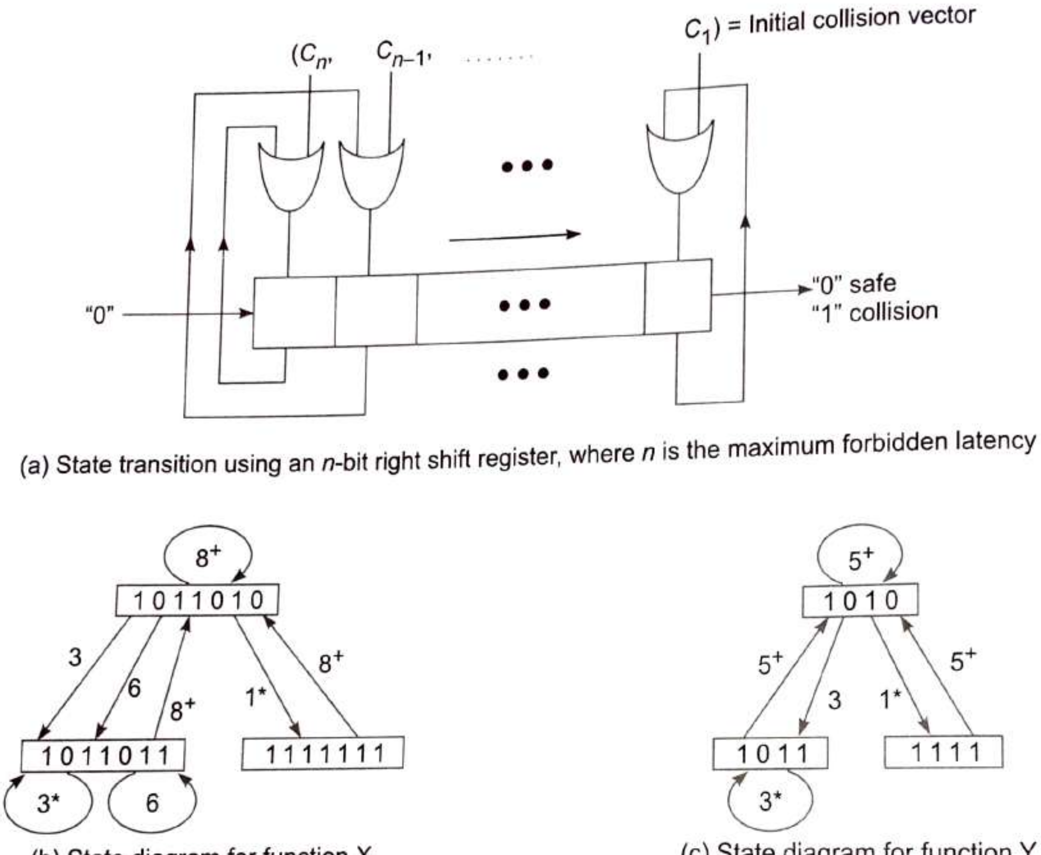
The *next state* of the pipeline at time $t + p$ is obtained with the assistance of an m-bit right shift register as in Fig. 6.6a. The initial collision vector C is initially loaded into the register. The register is then shifted to the right. Each 1-bit shift corresponds to an increase in the latency by 1. When a 0 bit emerges from the right end after p shifts, it means p is a permissible latency. Likewise, a 1 bit being shifted out means a collision, and thus the corresponding latency should be forbidden.

Logical 0 enters from the left end of the shift register. The next state after p shifts is thus obtained by bitwise-ORing the initial collision vector with the shifted register contents. For example, from the initial state $C_X = (1011010)$, the next state (1111111) is reached after one right shift of the register, and the next state (1011011) is reached after three shifts or six shifts.

Example 6.2 The state transition diagram for a pipeline unit

A *state diagram* is obtained in Fig. 6.6b for function X. 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. Similarly, from state (1011011), one reaches the same state after either three shifts or six shifts.

When the number of shifts is $m + 1$ or greater, all transitions are redirected back to the initial state. For example, after eight or more (denoted as 8^+) shifts, the next state must be the initial state, regardless of which state the transition starts from. In Fig. 6.6c, a state diagram is obtained for the reservation table in Fig. 6.3c using a 4-bit shift register. Once the initial collision vector is determined, the corresponding state diagram is uniquely determined.



(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

Fig. 6.6 Two state diagrams obtained from the two reservation tables in Fig. 6.3, respectively

The 0's and 1's in the present state, say at time t, of a state diagram indicate the permissible and forbidden latencies, respectively, at time t. The bitwise ORing of the shifted versicn of the present state with the initial collision vector is meant to prevent collisions from future initiations starting at time $t + 1$ and onward.

Thus the state diagram covers all permissible state transitions that avoid collisions. All latencies equal to or greater than m are permissible. This implies that collisions can always be avoided if events are scheduled far apart (with latencies of m^+). However, such long latencies are not tolerable from the viewpoint of pipeline throughput.

Greedy Cycles From the state diagram, we can determine optimal latency cycles which result in the MAL. There are infinitely many latency cycles one can trace from the state diagram. For example, (1, 8), (1, 8, 6, 8), (3), (6), (3, 8), (3, 6, 3) ..., are legitimate cycles traced from the state diagram in Fig. 6.6b. Among these cycles, only *simple cycles* are of interest.

A simple cycle is a latency cycle in which each state appears only once. In the state diagram in Fig. 6.6b, only (3), (6), (8), (1, 8), (3, 8), and (6, 8) are simple cycles. The cycle (1, 8, 6, 8) is not simple because it travels through the state (1011010) twice. Similarly, the cycle (3, 6, 3, 8, 6) is not simple because it repeats the state (1011011) three times.

Some of the simple cycles are *greedy cycles*. A greedy cycle is one whose edges are all made with minimum latencies from their respective starting states. For example, in Fig. 6.6b the cycles (1, 8) and (3) are greedy cycles. Greedy cycles in Fig. 6.6c are (1, 5) and (3). Such cycles must first be simple, and their average latencies must be lower than those of other simple cycles. The greedy cycle (1, 8) in Fig. 6.6b has an average latency of $(1 + 8)/2 = 4.5$, which is lower than that of the simple cycle $(6, 8) = (6 + 8)/2 = 7$. The greedy cycle (3) has a constant latency which equals the MAL for evaluating function X without causing a collision.

The MAL in Fig. 6.6c is 3, corresponding to either of the two greedy cycles. The minimum-latency edges in the state diagrams are marked with asterisks.

At least one of the greedy cycles will lead to the MAL. The collision-free scheduling of pipeline events is thus reduced to finding greedy cycles from the set of simple cycles. The greedy cycle yielding the MAL is the final choice.

6.2.3 Pipeline Schedule Optimization

An optimization technique based on the MAL is given below. 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 In 1972, Shar determined the following bounds on the *minimal average latency* (MAL) achievable by any control strategy on a statically reconfigured pipeline executing a given reservation table:

(1) The MAL is lower-bounded by the maximum number of checkmarks 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.

Interested readers may refer to Shar (1972) or find proofs of these bounds in Kogge (1981). These results suggest that the optimal latency cycle must be selected from one of the lowest greedy cycles. However, a greedy cycle is not sufficient to guarantee the optimality of the MAL. The lower bound guarantees the optimality. For example, the MAL = 3 for both function X and function Y and has met the lower bound of 3 from their respective reservation tables.

From Fig. 6.6b, the upper bound on the MAL for function X is equal to 4 + 1 = 5, a rather loose bound. On the other hand, Fig. 6.6c shows a rather tight upper bound of 2 + 1 = 3 on the MAL. Therefore, all greedy cycles for function Y lead to the optimal latency value of 3, which cannot be lowered further.

To optimize the MAL, one needs to find the lower bound by modifying the reservation table. The approach is to reduce the maximum number of checkmarks in any row. The modified reservation table must preserve the original function being evaluated. Patel and Davidson (1976) have suggested the use of noncompute delay stages to increase pipeline performance with a shorter MAL. Their technique is described below.

a contrary conclusion. The relationship between the two measures is a function of the reservation table and of the initiation cycle adopted.

At least one stage of the pipeline should be fully (100%) utilized at the steady state in any acceptable initiation cycle; otherwise, the pipeline capability has not been fully explored. In such cases, the initiation cycle may not be optimal and another initiation cycle should be examined for improvement.

6.3 INSTRUCTION PIPELINE DESIGN

A stream of instructions can be executed by a pipeline in an overlapped manner. We describe below instruction pipelines for CISC and RISC scalar processors. Topics to be studied include instruction prefetching, data forwarding, hazard avoidance, interlocking for resolving data dependences, dynamic instruction scheduling, and branch handling techniques for improving pipelined processor performance. Further discussion on instruction level parallelism will be found in Chapter 12.

6.3.1 Instruction Execution Phases

A typical instruction execution consists of a sequence of operations, including instruction fetch, decode, operand fetch, execute, and write-back phases. These phases are ideal for overlapped execution on a linear pipeline.

Pipelined Instruction Processing A typical instruction pipeline is depicted in Fig. 6.9. The *fetch stage* (F) fetches instructions from a cache memory, ideally one per cycle. The *decode stage* (D) reveals the instruction function to be performed and identifies the resources needed. Resources include general-purpose registers, buses, and functional units. The *issue stage* (I) reserves resources. The operands are also read from registers during the issue stage.

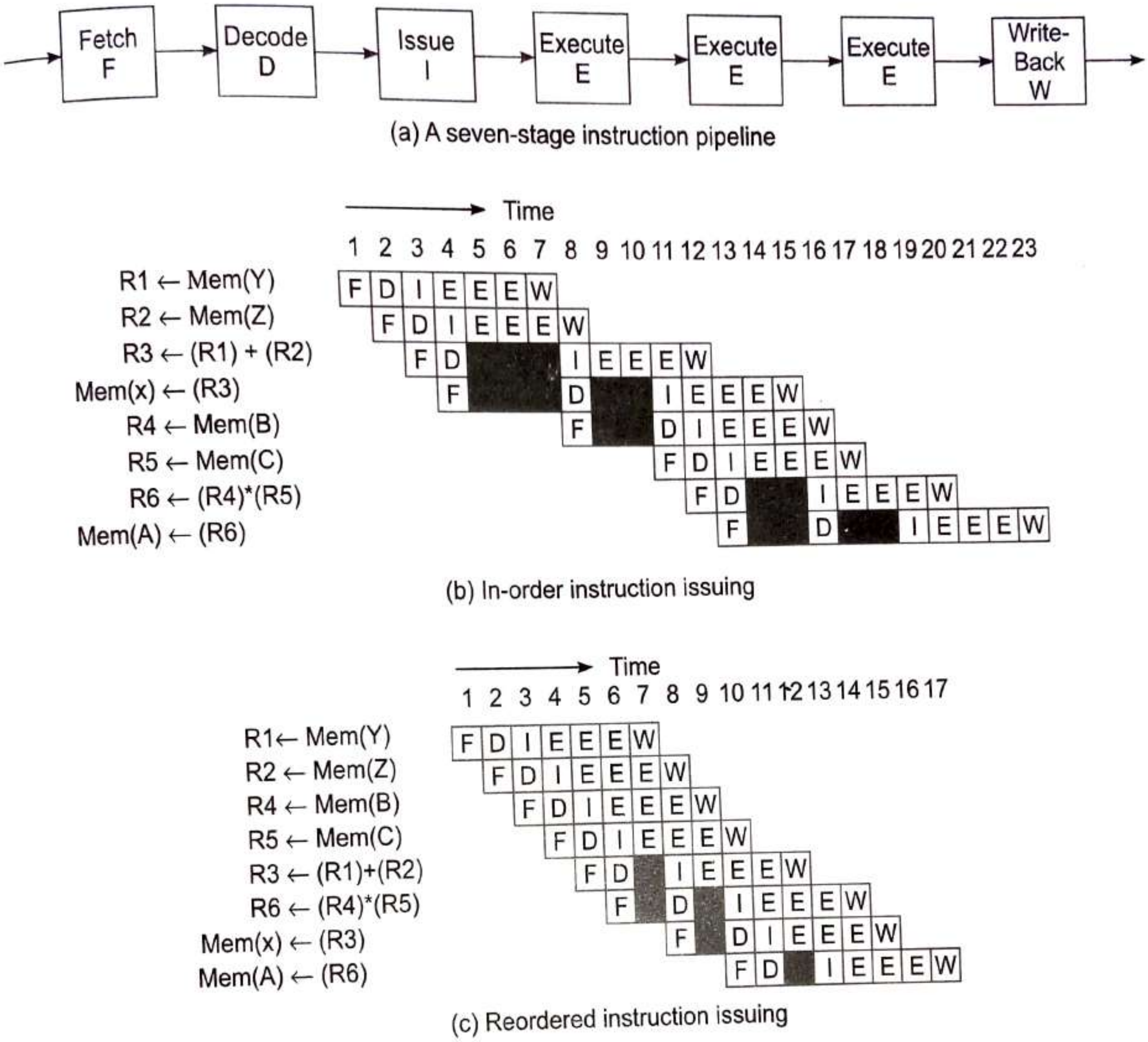
The instructions are executed in one or several *execute stages* (E). Three execute stages are shown in Fig. 6.9a. The last *writeback stage* (W) is used to write results into the registers. Memory load or store operations are treated as part of execution. Figure 6.9 shows the flow of machine instructions through a typical pipeline. These eight instructions are for pipelined execution of the high-level language statements X = Y + Z and A = B × C. Here we have assumed that *load* and *store* instructions take four execution clock cycles, while floating-point *add* and *multiply* operations take three cycles.

The above timing assumptions represent typical values found in an older CISC processor. In many RISC processors, fewer clock cycles are needed. On the other hand, Cray 1 required 11 cycles for a load and a floating-point addition took six. With in-order instruction issuing, if an instruction is blocked from issuing due to a data or resource dependence, all instructions following it are blocked.

Figure 6.9b illustrates the issue of instructions following the original program order. The shaded boxes correspond to idle cycles when instruction issues are blocked due to resource latency or conflicts or due to data dependences. The first two *load* instructions issue on consecutive cycles. The *add* is dependent on both *loads* and must wait three cycles before the data (Y and Z) are loaded in.

Similarly, the *store* of the sum to memory location X must wait three cycles for the *add* to finish due to a flow dependence. There are similar blockages during the calculation of A. The total time required is 17 clock cycles. This time is measured beginning at cycle 4 when the first instruction starts execution until cycle 20 when the last instruction starts execution. This timing measure eliminates the undue effects of the pipeline "startup" or "draining" delays.

Figure 6.9c shows an improved timing after the instruction issuing order is changed to eliminate unnecessary delays due to dependence. The idea is to issue all four *load* operations in the beginning. Both the *add* and *multiply* instructions are blocked fewer cycles due to this data prefetching. The reordering should not change the end results. The time required is being reduced to 11 cycles, measured from cycle 4 to cycle 14.



(a) A seven-stage instruction pipeline

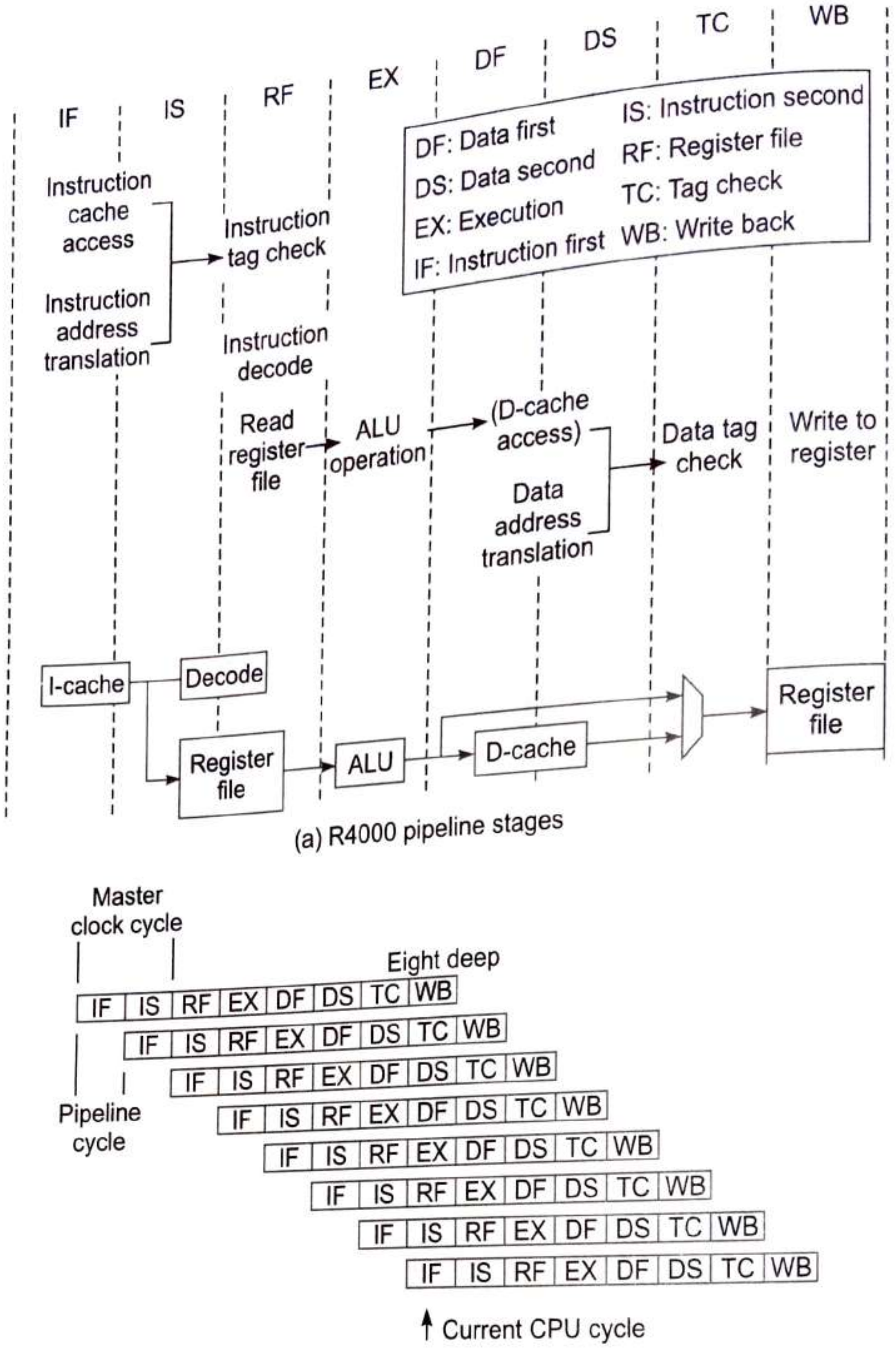
(b) In-order instruction issuing

(c) Reordered instruction issuing

Fig. 6.9 Pipelined execution of X = Y + Z and A = B × C (Courtesy of James Smith; reprinted with permission from *IEEE Computer*, July 1989)

Example 6.4 The MIPS R4000 instruction pipeline

The MIPS R4000 was a pipelined 64-bit processor using separate instruction and data caches and an eight-stage pipeline for executing register-based instructions. As illustrated in Fig. 6.10, the processor pipeline design was targeted to achieve an execution rate approaching one instruction per cycle.



(a) R4000 pipeline stages

(b) R4000 instruction overlapping in pipeline

Fig. 6.10 The architecture of the MIPS R4000 instruction pipeline (Courtesy of MIPS Computer Systems)

The execution of each R4000 instruction consisted of eight major steps as summarized in Fig. 6.10a. Each of these steps required approximately one clock cycle. The instruction and data memory references are split across two stages. The single-cycle ALU stage took slightly more time than each of the cache access stages.

The overlapped execution of successive instructions is shown in Fig. 6.10b. This pipeline operated efficiently because different CPU resources, such as address and bus access, ALU operations, register accesses, and so on, were utilized simultaneously on a noninterfering basis.

The internal pipeline clock rate (100 MHz) of the R4000 was twice the external input or master clock frequency. Figure 6.10b shows the optimal pipeline movement, completing one instruction every internal clock cycle. Load and branch instructions introduce extra delays.

6.3.2 Mechanisms for Instruction Pipelining

We introduce instruction buffers and describe the use of cacheing, collision avoidance, multiple functional units, register tagging, and internal forwarding to smooth pipeline flow and to remove bottlenecks and unnecessary memory access operations.

Prefetch Buffers Three types of buffers can be used to match the instruction fetch rate to the pipeline consumption rate. In one memory-access time, a block of consecutive instructions are fetched into a prefetch buffer as illustrated in Fig. 6.11. The block access can be achieved using interleaved memory modules or using a cache to shorten the effective memory-access time as demonstrated in the MIPS R4000.

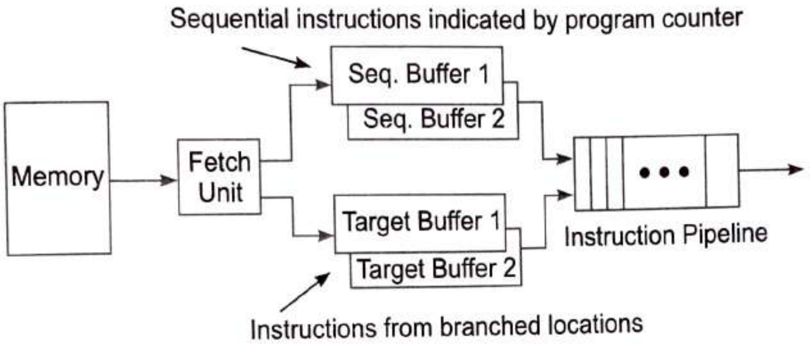


Fig. 6.11 The use of sequential and target buffers

Sequential instructions are loaded into a pair of *sequential buffers* for in-sequence pipelining. Instructions from a branch target are loaded into a pair of *target buffers* for out-of-sequence pipelining. Both buffers operate in a first-in-first-out fashion. These buffers become part of the pipeline as additional stages.

A conditional branch instruction causes both sequential buffers and target buffers to fill with instructions. After the branch condition is checked, appropriate instructions are taken from one of the two buffers, and instructions in the other buffer are discarded. Within each pair, one can use one buffer to load instructions from memory and use another buffer to feed instructions into the pipeline. The two buffers in each pair alternate to prevent a collision between instructions flowing into and out of the pipeline.

A third type of prefetch buffer is known as a *loop buffer*. This buffer holds sequential instructions contained in a small loop. 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. The loop buffer operates in two steps. First, it contains instructions sequentially ahead of the current instruction. This saves the instruction fetch time from memory. Second, it recognizes when the target of a branch falls within the loop boundary. In this case, unnecessary memory accesses can be avoided if the target instruction is already in the loop buffer. The CDC 6600 and Cray 1 made use of loop buffers.

Multiple Functional Units Sometimes a certain pipeline stage becomes the bottleneck. This stage corresponds to the row with the maximum number of checkmarks in the reservation table. This bottleneck problem can be alleviated by using multiple copies of the same stage simultaneously. This leads to the use of multiple execution units in a pipelined processor design (Fig. 6.12).

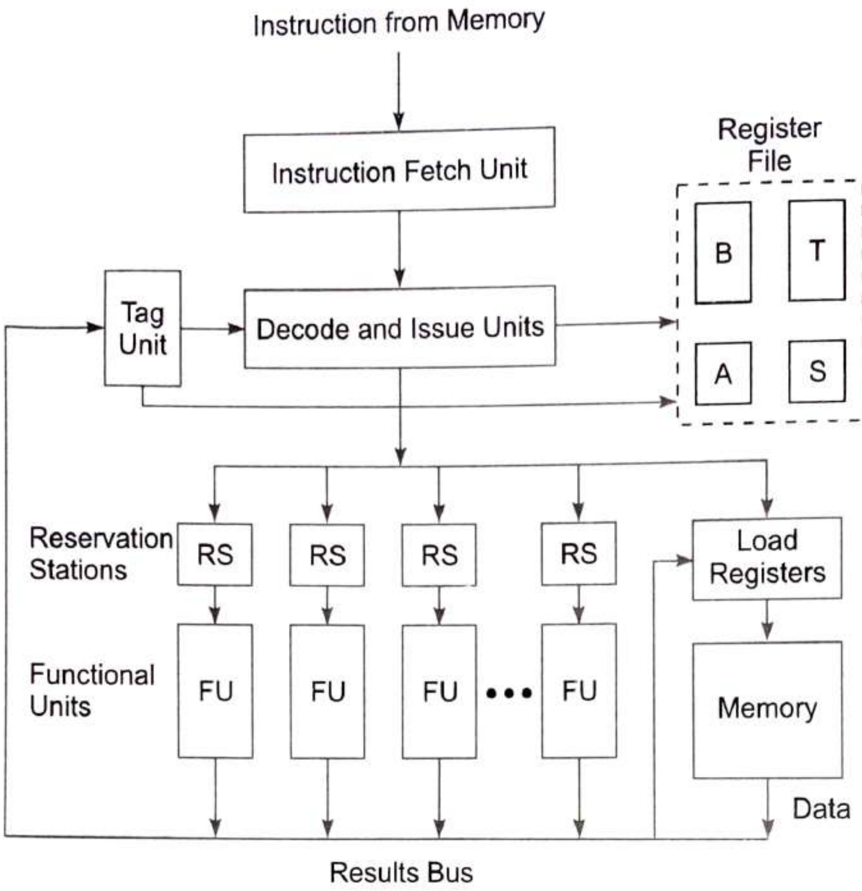


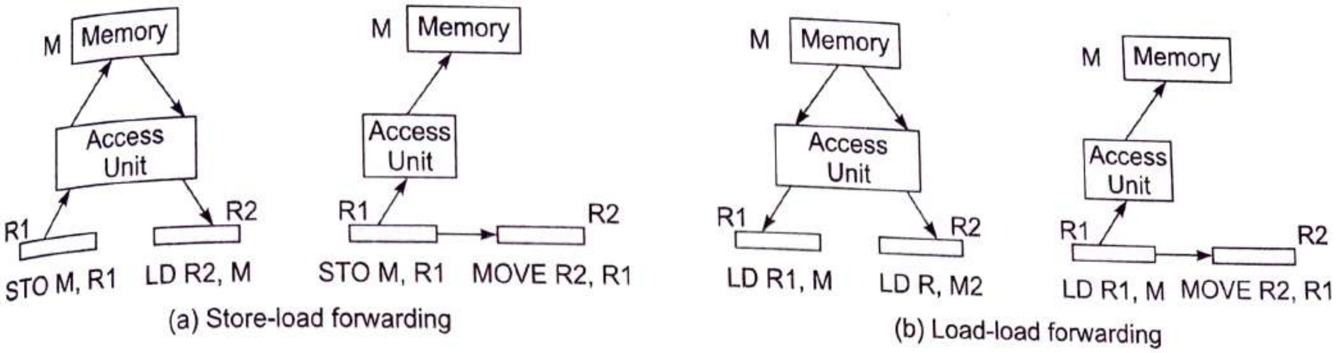
Fig. 6.12 A pipelined processor with multiple functional units and distributed reservation stations supported by tagging (Courtesy of G. Sohi; reprinted with permission from *IEEE Transactions on Computers*, March 1990)

Sohi (1990) used a model architecture for a pipelined scalar processor containing multiple functional units (Fig. 6.12). In order to resolve data or resource dependences among the successive instructions entering the pipeline, the *reservation stations* (RS) are used with each functional unit. Operations wait in the RS until their data dependences have been resolved. Each RS is uniquely identified by a *tag*, which is monitored by a *tag unit*.

The tag unit keeps checking the tags from all currently used registers or RSs. This register tagging technique allows the hardware to resolve conflicts between source and destination registers assigned for multiple instructions. Besides resolving conflicts, the RSs also serve as buffers to interface the pipelined functional units with the decode and issue units. The multiple functional units operate in parallel, once the dependences are resolved. This alleviates the bottleneck in the execution stages of the instruction pipeline.

Internal Data Forwarding The throughput of a pipelined processor can be further improved with internal data forwarding among multiple functional units. In some cases, some memory-access operations can be replaced by register transfer operations. The idea is described in Fig. 6.13.

A *store-load forwarding* is shown in Fig. 6.13a in which the *load operation* (LD R2, M) from memory to register R2 can be replaced by the *move* operation (MOVE R2, R1) from register R1 to register R2. Since register transfer is faster than memory access, this data forwarding will reduce memory traffic and thus results in a shorter execution time. Similarly, *load-load forwarding* (Fig. 6.13b) eliminates the second *load* operation (LD R2, M) and replaces it with the *move* operation (MOVE R2, R1). Further discussion on operand forwarding will be continued in Chapter 12.



(a) Store-load forwarding

(b) Load-load forwarding

Fig. 6.13 Internal data forwarding by replacing memory-access operations with register transfer operations

Example 6.5 Implementing the dot-product operation with internal data forwarding between a multiply unit and an add unit

One can feed the output of a multiplier directly to the input of an adder (Fig. 6.14) for implementing the following dot-product operation:

$$s = \sum_{i=1}^{n} a_i \times b_i \tag{6.10}$$

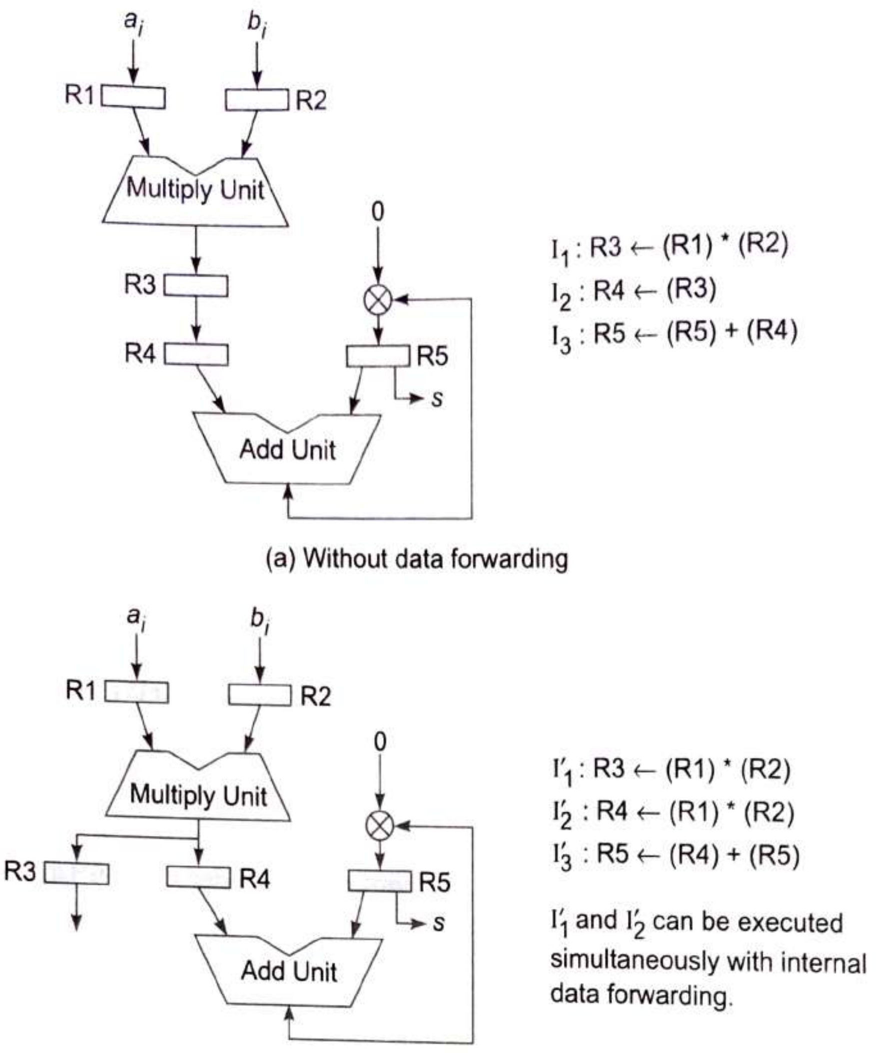
Without internal data forwarding between the two functional units, the three instructions must be sequentially executed in a looping structure (Fig. 6.14a). With data forwarding, the output of the multiplier is fed directly into the input register R4 of the adder (Fig. 6.14b). At the same time, the output of the multiplier is also routed to register R3. Internal data forwarding between the two functional units thus reduces the total execution time through the pipelined processor.

Hazard Avoidance 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. As illustrated in Fig. 6.15, three types of logic *hazards* are possible.

Consider two instructions I and J. Instruction J is assumed to logically follow instruction I according to program order. If the actual execution order of these two instructions violates the program order, incorrect results may be read or written, thereby producing hazards.

Hazards should be prevented before these instructions enter the pipeline, such as by holding instruction J until the dependence on instruction I is resolved. We use the notation D(I) and R(I) for the *domain* and *range* of an instruction I.

The domain contains the *input set* (such as operands in registers or in memory) to be used by instruction I. The range corresponds to the *output set* of instruction I. Listed below are conditions under which possible hazards can occur:



$I_1 : R3 \leftarrow (R1) * (R2)$

$I_2 : R4 \leftarrow (R3)$

$I_3 : R5 \leftarrow (R5) + (R4)$

(a) Without data forwarding

$I'_1 : R3 \leftarrow (R1) * (R2)$

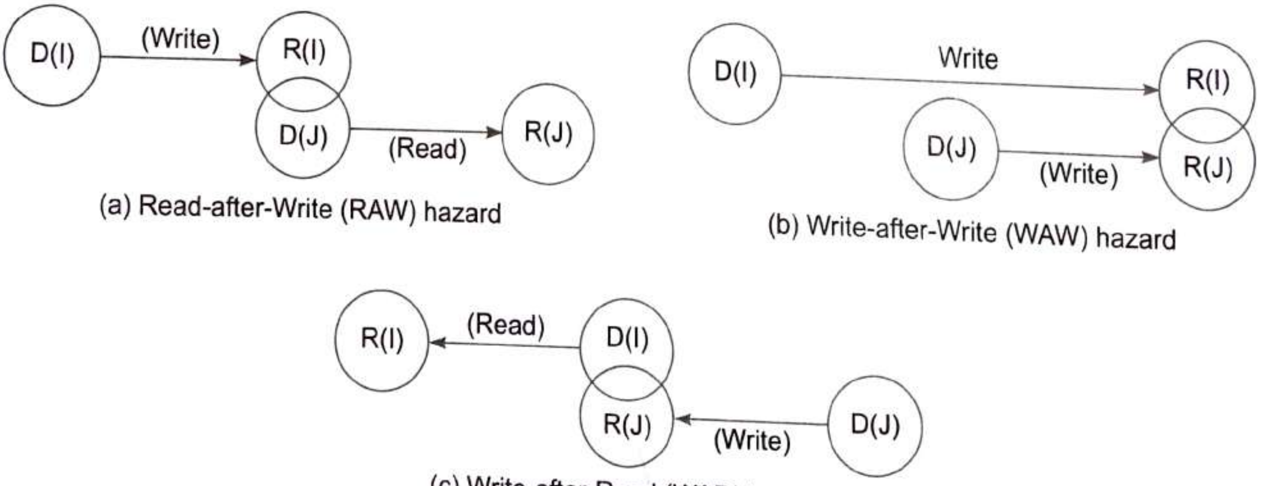
$I'_2 : R4 \leftarrow (R1) * (R2)$

$I'_3 : R5 \leftarrow (R4) + (R5)$

I'_1 and I'_2 can be executed simultaneously with internal data forwarding.

(b) With internal data forwarding

Fig. 6.14 Internal data forwarding for implementing the dot-product operation



(a) Read-after-Write (RAW) hazard

(b) Write-after-Write (WAW) hazard

(c) Write-after-Read (WAR) hazard

Fig. 6.15 Possible hazards between read and write operations in an instruction pipeline (instruction I is ahead of instruction J in program order)

$$\begin{aligned} R(I) \cap D(J) &\neq \phi \text{ for RAW hazard} \\ R(I) \cap R(J) &\neq \phi \text{ for WAW hazard} \\ D(I) \cap R(J) &\neq \phi \text{ for WAR hazard} \end{aligned} \tag{6.11}$$

These conditions are necessary but not sufficient. This means the hazard may not appear even if one or more of the conditions exist. The RAW hazard corresponds to the flow dependence, WAR to the antidependence, and WAW to the output dependence introduced in Section 2.1. The occurrence of a logic hazard depends on the order in which the two instructions are executed. Chapter 12 discusses techniques to handle such hazards.

6.3.3 Dynamic Instruction Scheduling

In this section, we describe three methods for scheduling instructions through an instruction pipeline. The *static scheduling* scheme is supported by an optimizing compiler. *Dynamic scheduling* is achieved using a technique such as Tomasulo's *register-tagging* scheme built in the IBM 360/91, or the *scoreboarding* scheme built in the CDC 6600 processor.

Static Scheduling Data dependences in a sequence of instructions create interlocked relationships among them. Interlocking can be resolved through a compiler-based static scheduling approach. A compiler or a postprocessor can be used to increase the separation between interlocked instructions.

Consider the execution of the following code fragment. The *multiply* instruction cannot be initiated until the preceding *load* is complete. This data dependence will stall the pipeline for three clock cycles since the two *loads* overlap by one cycle.

Stage delay:	Instruction:		
2 cycles	Add	R0, R1	/R0 ← (R0) + (R1)/
1 cycle	Move	R1, R5	/R1 ← (R5)/
2 cycles	Load	R2, M(α)	/R2 ← (Memory (α))/
2 cycles	Load	R3, M(β)	/R3 ← (Memory (β))/
3 cycles	Multiply	R2, R3	/R2 ← (R2) × (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. The following program is obtained after this modification:

Load	R2, M(α)	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

Through this code rearrangement, the data dependences and program semantics are preserved, and the *multiply* can be initiated without delay. While the operands are being loaded from memory cells α and β into registers R2 and R3, the two instructions *add* and *move* consume three cycles and thus pipeline stalling is avoided.

6.4 ARITHMETIC PIPELINE DESIGN

Pipelining techniques can be applied to speed up numerical arithmetic computations. We start with a review of arithmetic principles and standards. Then we consider arithmetic pipelines with fixed functions.

A fixed-point multiply pipeline design and the MC68040 floating-point unit are used as examples to illustrate the design techniques involved. A multifunction arithmetic pipeline is studied with the TI-ASC arithmetic processor as an example.

6.4.1 Computer Arithmetic Principles

In a digital computer, arithmetic is performed with *finite precision* due to the use of fixed-size memory words or registers. Fixed-point or integer arithmetic offers a fixed range of numbers that can be operated upon. Floating-point arithmetic operates over a much increased dynamic range of numbers.

In modern processors, fixed-point and floating-point arithmetic operations are very often performed by separate hardware on the same processor chip.

Finite precision implies that numbers exceeding the limit must be truncated or rounded to provide a precision within the number of significant bits allowed. In the case of floating-point numbers, exceeding the exponent range means error conditions, called *overflow* or *underflow*. 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.

Fixed-Point Operations Fixed-point numbers are represented internally in machines in *sign-magnitude, one's complement*, or *two's complement* notation. Most computers use the two's complement notation because of its unique representation of all numbers (including zero). One's complement notation introduces a second zero representation called *dirty zero*.

Add, subtract, multiply, and *divide* are the four primitive arithmetic operations. For fixed-point numbers, the add or subtract of two n-bit integers (or fractions) produces an n-bit result with at most one carry-out.

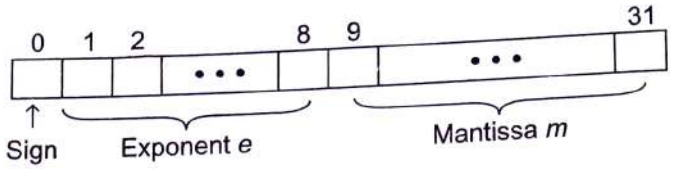
The multiplication of two n-bit numbers produces a $2n$-bit result which requires the use of two memory words or two registers to hold the full-precision result.

The division of an n-bit number by another may create an arbitrarily long quotient and a remainder. Only an approximate result is expected in fixed-point division with rounding or truncation. However, one can expand the precision by using a $2n$-bit dividend and an n-bit divisor to yield an n-bit quotient.

Floating-Point Numbers 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*). The algebraic value is represented as $X = m \times r^e$. The sign of X can be embedded in the mantissa.

Example 6.9 The IEEE 754 floating-point standard

A 32-bit floating-point number is specified in the IEEE 754 Standard as follows:



A binary base is assumed with $r = 2$. The 8-bit exponent e field uses an *excess-127* code. The dynamic range of e is $(-127, 128)$, internally represented as $(0, 255)$. The sign s and the 23-bit mantissa field m form a 25-bit sign-magnitude fraction, including an implicit or "hidden" 1 bit to the left of the binary point. Thus the complete mantissa actually represents the value $1.m$ in binary.

This hidden bit is not stored with the number. If $0 < e < 255$, then a nonzero normalized number represents the following algebraic value:

$$X = (-1)^s \times 2^{e-127} \times (1.m) \tag{6.15}$$

When $e = 255$ and $m \neq 0$, a *not-a-number* (NaN) is represented. NaNs can be caused by dividing a zero by a zero or taking the square root of a negative number, among many other nondeterminate cases. When $e = 255$ and $m = 0$, an infinite number $X = (-1)^s \infty$ is represented. Note that $+\infty$ and $-\infty$ are represented differently.

When $e = 0$ and $m \neq 0$, the number represented is $X = (-1)^s 2^{-126}(0.m)$. When $e = 0$ and $m = 0$, a zero is represented as $X = (-1)^s 0$. Again, $+0$ and -0 are possible.

The 64-bit (double-precision) floating point can be defined similarly using an excess-l023 code in the exponent field and a 52-bit mantissa field. A number which is nonzero, finite, non-NaN, and normalized, has the following value:

$$X = (-1)^s \times 2^{e-1023} \times (1.m) \tag{6.16}$$

Special rules are given in the standard to handle overflow or underflow conditions. Interested readers may check the published IEEE standards for details.

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 $Y = (m_y, e_y)$. For clarity, we assume $e_x \leq e_y$ and base $r = 2$.

$$X + Y = (m_x \times 2^{e_x - e_y} + m_y) \times x^{e_y} \tag{6.17}$$

$$X - Y = (m_x \times 2^{e_x - e_y} - m_y) \times x^{e_y} \tag{6.18}$$

$$X \times Y = (m_x \times m_y) \times 2^{e_x + e_y} \tag{6.19}$$

$$X \div Y = (m_x \div m_y) \times 2^{e_x - e_y} \tag{6.20}$$

The above equations clearly identify the number of arithmetic operations involved in each floating-point function. These operations can be divided into two halves: One half is for exponent operations such as comparing their relative magnitudes or adding/subtracting them; the other half is for mantissa operations, including four types of fixed-point operations.

Floating-point units are ideal for pipelined implementation. The two halves of the operations demand almost twice as much hardware as that required in a fixed-point unit. Arithmetic shifting operations are needed for equalizing the two exponents before their mantissas can be added or subtracted.

Shifting a binary fraction m to the right k places corresponds to the weighting $m \times 2^{-k}$, and shifting k places to the left corresponds to $m \times 2^k$. In addition, normalization of a floating-point number also requires left shifts to be performed.

Elementary Functions Elementary functions include trigonometric, exponential, logarithmic, and other transcendental functions. Truncated polynomials or power series can be used to evaluate the elementary functions, such as sin x, ln x, e^x, cosh x, $\tan^{-1} y$, $\sqrt{x}$, x^3, etc. Interested readers may refer to the book by Hwang (1979) for details of computer arithmetic functions and their hardware implementation.

It should be noted that computer arithmetic can be implemented by hardwired logic circuitry as well as by table lookup using fast memory. Frequently used constants and special function values can also be generated by table lookup.

6.4.2 Static Arithmetic Pipelines

Most of today's arithmetic pipelines are designed to perform fixed functions. These *arithmetic/logic units* (ALUs) perform fixed-point and floating-point operations separately. The fixed-point unit is also called the integer unit. The floating-point unit can be built either as part of the central processor or on a separate coprocessor.

These arithmetic units perform scalar operations involving one pair of operands at a time. The pipelining in scalar arithmetic pipelines is controlled by software loops. Vector arithmetic units can be designed with pipeline hardware directly under firmware or hardwired control.

Scalar and vector arithmetic pipelines differ mainly in the areas of register files and control mechanisms involved. Vector hardware pipelines are often built as add-on options to a scalar processor or as an attached processor driven by a control processor. Both scalar and vector processors are used in modern supercomputers.

Arithmetic Pipeline Stages Depending on the function to be implemented, different pipeline stages in an arithmetic unit require different hardware logic. Since all arithmetic operations (such as *add, subtract, multiply, divide, squaring, square rooting, logarithm,* etc.) can be implemented with the basic add and shifting operations, the core arithmetic stages require some form of hardware to add and to shift.

For example, a typical three-stage floating-point adder includes a first stage for exponent comparison and equalization which is implemented with an integer adder and some shifting logic; a second stage for fraction addition using a high-speed carry lookahead adder; and a third stage for fraction normalization and exponent readjustment using a shifter and another addition logic.

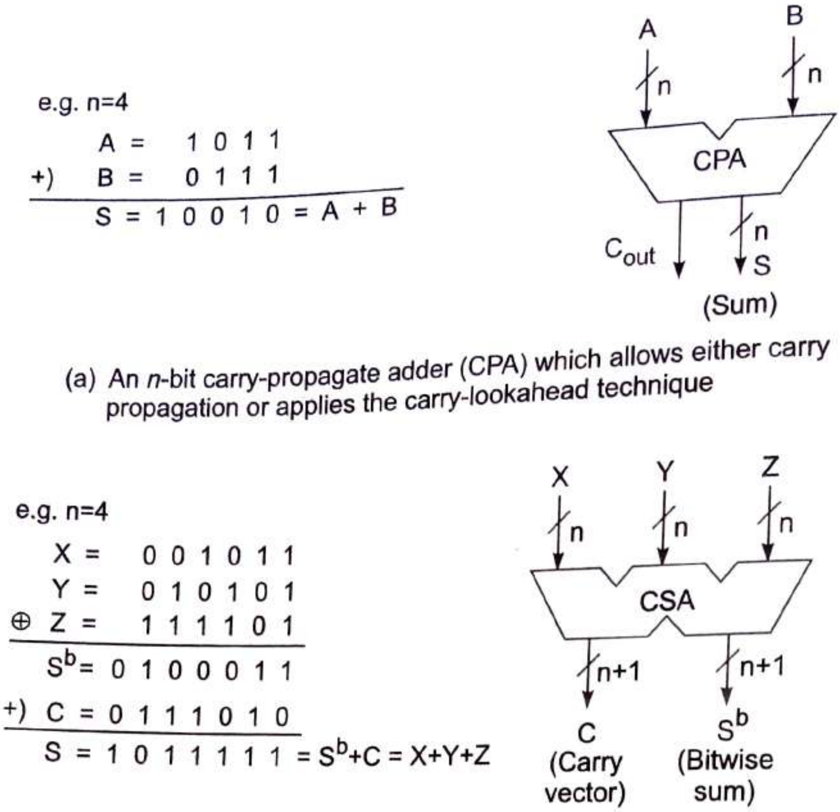
Arithmetic or logical shifts can be easily implemented with *shift registers*. High-speed addition requires either the use of a *carry-propagation adder* (CPA) which adds two numbers and produces an arithmetic sum as shown in Fig. 6.22a, or the use of a *carry-save adder* (CSA) to "add" three input numbers and produce one sum output and a carry output as exemplified in Fig. 6.22b.

In a CPA, the carries generated in successive digits are allowed to propagate from the low end to the high end, using either ripple carry propagation or some carry looka-head technique.

In a CSA, the carries are not allowed to propagate but instead are saved in a carry vector. In general, an n-bit CSA is specified as follows: Let X, Y, and Z be three n-bit input numbers, expressed as $X = (x_{n-1}, x_{n-2}, \ldots, x_1, x_0)$ and so on. The CSA performs bitwise operations simultaneously on all columns of digits to produce two n-bit output numbers, denoted as $S^b = (0, S_{n-1}, S_{n-2}, \ldots, S_1, S_0)$ and $C = (C_n, C_{n-1}, \ldots, C_1, 0)$.

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. The input-output relationships are expressed below:

$$\begin{aligned} S_i &= x_i \oplus y_i \oplus z_i \\ C_{i+1} &= x_i y_i \vee y_i z_i \vee z_i x_i \end{aligned} \tag{6.21}$$



```
e.g. n=4
        A =   1 0 1 1
+)      B =   0 1 1 1
---------------------------------
        S = 1 0 0 1 0 = 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
-------------------------
    Sb=  0 1 0 0 0 1 1
+)   C = 0 1 1 1 0 1 0
-------------------------
     S = 1 0 1 1 1 1 1 = Sb+C = X+Y+Z
```

(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

Fig. 6.22 Distinction between a carry-propagate adder (CPA) and a carry-save adder (CSA)

for $i = 0, 1, 2, \ldots, n-1$, where ⊕ is the exclusive OR and ∨ is the logical OR operation. Note that the arithmetic sum of three input numbers, i.e., $S = X + Y + Z$, is obtained by adding the two output numbers, i.e., $S = S^b + C$, using a CPA. We use the CPA and CSAs to implement the pipeline stages of a fixed-point multiply unit as follows.

Multiply Pipeline Design Consider as an example the multiplication of two 8-bit integers $A \times B = P$, where P is the 16-bit product. This fixed-point multiplication can be written as the summation of eight partial products as shown below: $P = A \times B = P_0 + P_1 + P_2 + \cdots + P_7$, where × and + are arithmetic multiply and add operations, respectively.

```
                            1  0  1  1  0  1  0  1  =  A
                         ×) 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
                1  0  1  1  0  1  0  1  0  0  0  0  =  P4
             0  0  0  0  0  0  0  0  0  0  0  0  0  =  P5
          0  0  0  0  0  0  0  0  0  0  0  0  0  0  =  P6
    +) 1  0  1  1  0  1  0  1  0  0  0  0  0  0  0  =  P7
    ------------------------------------------------
    0  1  1  0  0  1  1  1  1  1  1  0  1  1  1  1  =  P
```

Note that 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 Fig. 6.23.

The first stage (S_1) generates all eight partial products, ranging from 8 bits to 15 bits, simultaneously. The second stage (S_2) is made up of two levels of four CSAs, and it essentially merges eight numbers into four numbers ranging from 13 to 15 bits. The third stage (S_3) consists of two CSAs, and it merges four numbers from S_2 into two 16-bit numbers. The final stage (S_4) is a CPA, which adds up the last two numbers to produce the final product P.

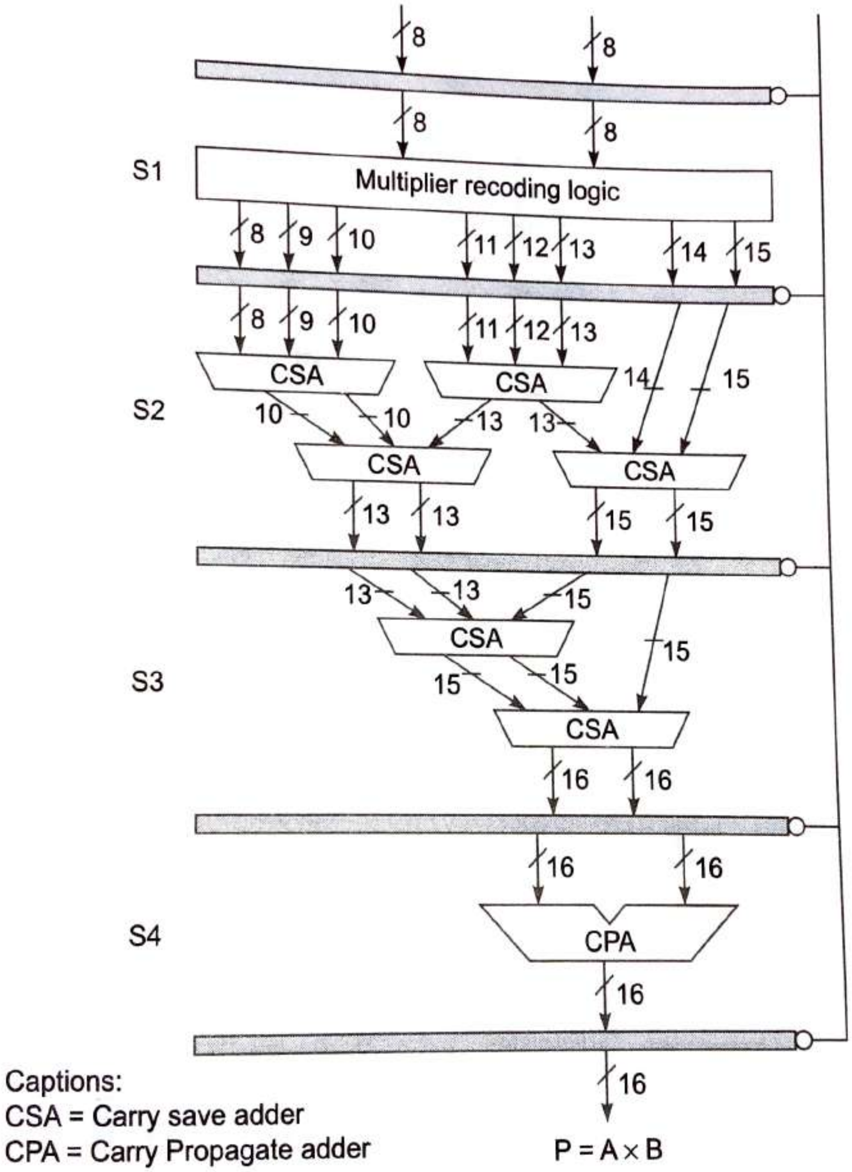


Fig. 6.23 A pipeline unit for fixed-point multiplication of 8-bit integers (The number along each line indicates the line width.)

For a maximum width of 16 bits, the CPA is estimated to need four gate levels of delay. Each level of the CSA can be implemented with a two-gate-level logic. The delay of the first stage (S_1) also involves two gate levels. Thus all the pipeline stages have an approximately equal amount of delay.

The matching of stage delays is crucial to the determination of the number of pipeline stages, as well as the clock period (Eq. 6.1). If the delay of the CPA stage can be further reduced to match that of a single CSA level, then the pipeline can be divided into six stages with a clock rate twice as fast. The basic concepts can be extended to operands with a larger number of bits, as we see in the example below.

Example 6.10 The floating-point unit in the Motorola MC68040

Figure 6.24 shows the design of a pipelined floating-point unit built as an on-chip feature in the Motorola M68040 processor.

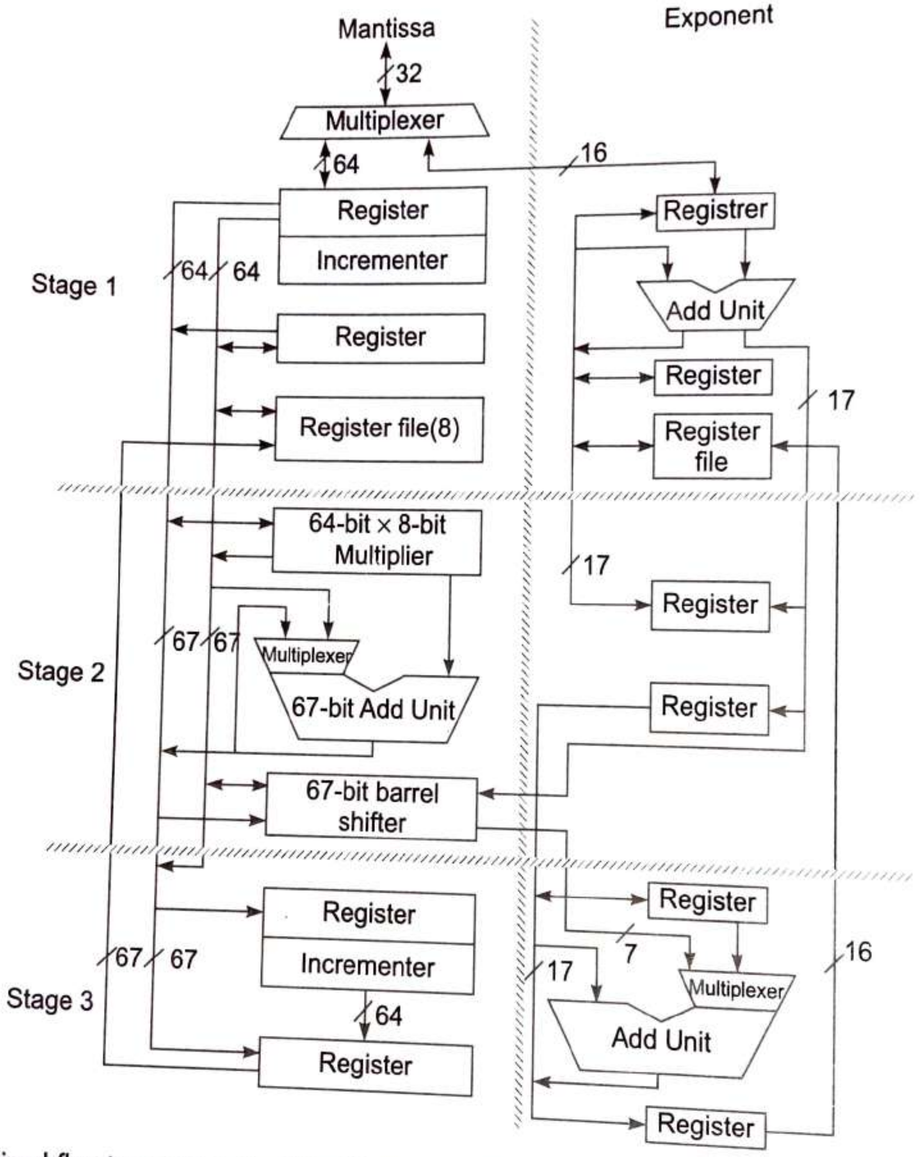


Fig. 6.24 Pipelined floating-point unit of the Motorola MC68040 processor (Courtesy of Motorola, Inc., 1992)

This arithmetic pipeline has three stages. The mantissa section and exponent section are essentially two separate pipelines. The mantissa section can perform floating-point add or multiply operations, either single-precision (32 bits) or double-precision (64 bits).

In the mantissa section, stage 1 receives input operands and returns with computation results; 64-bit registers are used in this stage. Note that all three stages are connected to two 64-bit data buses. Stage 2 contains the array multiplier (64 × 8) which must be repeatedly used to carry out a long multiplication of the two mantissas.

The 67-bit adder performs the addition/subtraction of two mantissas, the barrel shifter is used for normalization. Stage 3 contains registers for holding results before they are loaded into the register file in stage 1 for subsequent use by other instructions.

On the exponent side, a 16-bit bus is used between stages. Stage 1 has an exponent adder for comparing the relative magnitude of two exponents. The result of stage 1 is used to equalize the exponents before mantissa addition can be performed. Therefore, a shift count (from the output of the exponent adder) is sent to the barrel shifter for mantissa alignment.

After normalization of the final result (getting rid of leading zeros), the exponent needs to be readjusted in stage 3 using another adder. The final value of the resulting exponent is fed from the register in stage 3 to the register file in stage 1, ready for subsequent usage.

Convergence Division One technique for division involves repeated multiplications. Mantissa division is carried out by a *convergence method.* This convergence division obtains the quotient $Q = M/D$ of two normalized fractions $0.5 \le M < D < 1$ in two's complement notation by performing two sequences of chain multiplications as follows:

$$Q = \frac{M \times R_1 \times R_2 \times \cdots \times R_k}{D \times R_1 \times R_2 \times \cdots \times R_k} \tag{6.22}$$

where the successive multipliers

$$R_i = 1 + \delta^{2^{i-1}} = 2 - D^{(i)} \quad \text{for } i = 1, 2, \ldots, k \quad \text{and} \quad D = 1 - \delta$$

The purpose is to choose R_i such that the denominator $D^{(k)} = D \times R_1 \times R_2 \times \cdots \times R_k \rightarrow 1$ for a sufficient number of k iterations, and then the resulting numerator $M \times R_1 \times R_2 \times \cdots \times R_k \rightarrow Q$.

Note that the multiplier R_i can be obtained by finding the two's complement of the previous chain product $D^{(i)} = D \times R_1 \times \cdots \times R_{i-1} = 1 - \delta^{2i-1}$ because $2 - D^{(i)} = R_i$. The reason why $D^{(k)} \rightarrow 1$ for large k is that

$$\begin{aligned} D^{(i)} &= (1-\delta)(1+\delta)(1+\delta^2)(1+\delta^4) \cdots (1+\delta^{2^{i-1}}) \\ &= (1-\delta^2)(1+\delta^2)(1+\delta^4) \cdots (1+\delta^{2^{i-1}}) \\ &= (1-\delta^{2^i}) \quad \text{for } i = 1, 2, \ldots, k \end{aligned} \tag{6.23}$$

Since $0 < \delta = 1 - D \le 0.5$, $\delta^{2^i} \rightarrow 0$ as i becomes sufficiently large, say, $i = k$ for some k; thus $D^{(k)} = 1 - \delta^{2^k} = 1$ for large k. The end result is

$$Q = M \times (1+\delta) \times (1+\delta^2) \times \cdots \times (1+\delta^{2^{k-1}}) \tag{6.24}$$

The above two sequences of chain multiplications are carried out alternately between the numerator and denominator through the pipeline stages. To summarize, in this technique division is carried out by repeated multiplications. Thus divide and multiply can share the same hardware pipeline.

Example 6.11 The IBM 360/Model 91 floating-point unit design

In the history of building scientific computers, IBM 360 Model 91 was certainly a milestone. Many of the pipeline design features introduced in previous sections were implemented in this machine. Therefore, it is worth the effort to examine the architecture of Model 91. In particular, we describe how floating-point add and multiply/divide operations were implemented in this machine.

As shown in Fig. 6.25, the floating-point execution unit in Model 91 consisted of two separate functional pipelines: the *add unit* and the *multiply/divide unit,* which could be used concurrently. The former was a two-stage pipeline, and the latter was a six-stage pipeline.

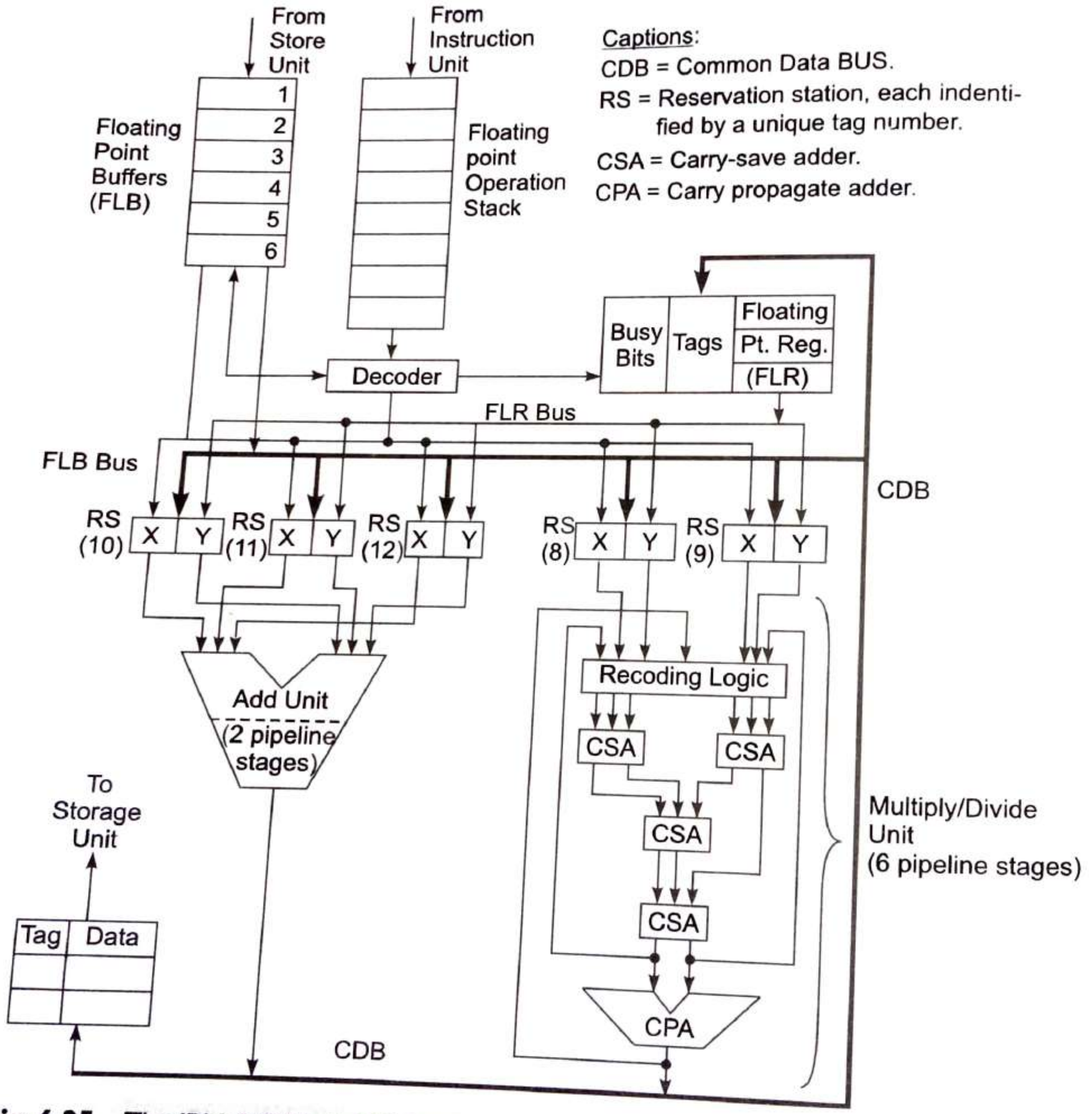


Fig. 6.25 The IBM 360 Model 91 floating-point unit (Courtesy of IBM Corporation, 1967)

The floating-point operation stack was a kind of prefetch buffer holding eight floating-point instructions for subsequent execution through the two functional pipelines. The floating-point buffers were used to input operands.

Operands may also come from the floating-point registers which were connected via the common data bus to the output bus. Results from the two functional units could be sent back to the memory via the store data buffers, or they could be routed back to the FLR or to the reservation stations at the input ends.

The add unit allowed three pairs of operands to be loaded into three reservation stations. Only one pair could be used at a time. The other two pairs held operands for subsequent use. The use of these reservation stations made the add unit behave like three virtual functional units.

Similarly, the two pairs at the input end of the multiply/divide unit made it behave like two virtual units. Internal data forwarding in Model 91 was accomplished using source tags on all registers and reservation stations. Divide was implemented in Model 91 based on the convergence method.

Every source of an input operand was uniquely identified with a 4-bit tag. Every destination of an input operand had an associated tag register that held the tag naming the source of data if the destination was busy. Through this *register tagging* technique, operands/results could be directly passed among the virtual functional units. This forwarding significantly cut down the data flow time between them.

Dynamic scheduling logic was built into Model 91 using Tomasulo's algorithm to resolve the data dependence problem. Either the add unit or the multiply/divide unit could execute an operation using operands from one of the reservation stations.

Under Tomasulo's algorithm, data dependences are preserved by copying source tags when the sources are busy. When data is generated by a source, it passes its identification and the data onto the common data bus. Awaiting destinations continuously monitor the bus in a tag watch.

When the source tag matches, the destination takes in the data from the bus. Other variations of Tomasulo's algorithm can be made to store the source tags within the destinations, to use a special tag (such as 0000) to indicate nonbusy register/buffers, or to use direct-mapped tags to avoid associative hardware.

Besides the IBM 360/370, the CDC 6600/7600 also implemented convergence division. It took two pipeline cycles to perform the floating-point add, six cycles to multiply, and 18 cycles to divide in the IBM System/360 Model 91 due to five iterations involved in the convergence division process.

6.4.3 Multifunctional Arithmetic Pipelines

Static arithmetic pipelines are designed to perform a fixed function and are thus called *unifunctional*. When a pipeline can perform more than one function, it is called *multifunctional*. A multifunctional pipeline can be either *static* or *dynamic*. Static pipelines perform *one* function at a time, but different functions can be performed at different times. A dynamic pipeline allows several functions to be performed simultaneously through the pipeline, as long as there are no conflicts in the shared usage of pipeline stages. In this section, we study a static multifunctional pipeline which was designed into the TI Advanced Scientific Computer (ASC).

Example 6.12 The TI/ASC arithmetic processor design

There were four pipeline arithmetic units built into the TI-ASC system, as shown in Fig. 6.26. The instruction-processing unit handled the fetching and decoding of instructions. There were a large number of working registers in the processor which also controlled the operations of the memory buffer unit and of the arithmetic units.

There were two sets of operand buffers, {*X*, *Y*, *Z*} and {*X*′, *Y*′, *Z*′}, in each arithmetic unit. *X*′, *X*, *Y*′ and *Y* were used for input operands, and *Z*′ and *Z* were used to output results. Note that intermediate results could be also routed from *Z*-registers to either *X*- or *Y*-registers. Both processor and memory buffers accessed the main memory for instructions and operands/results, respectively.

Each pipeline arithmetic unit had eight stages as shown in Fig. 6.27a. The PAU was a static multifunction pipeline which could perform only one function at a time. Figure 6.27a shows all the possible interstage connections for performing arithmetic, logical, shifting, and data conversion functions.

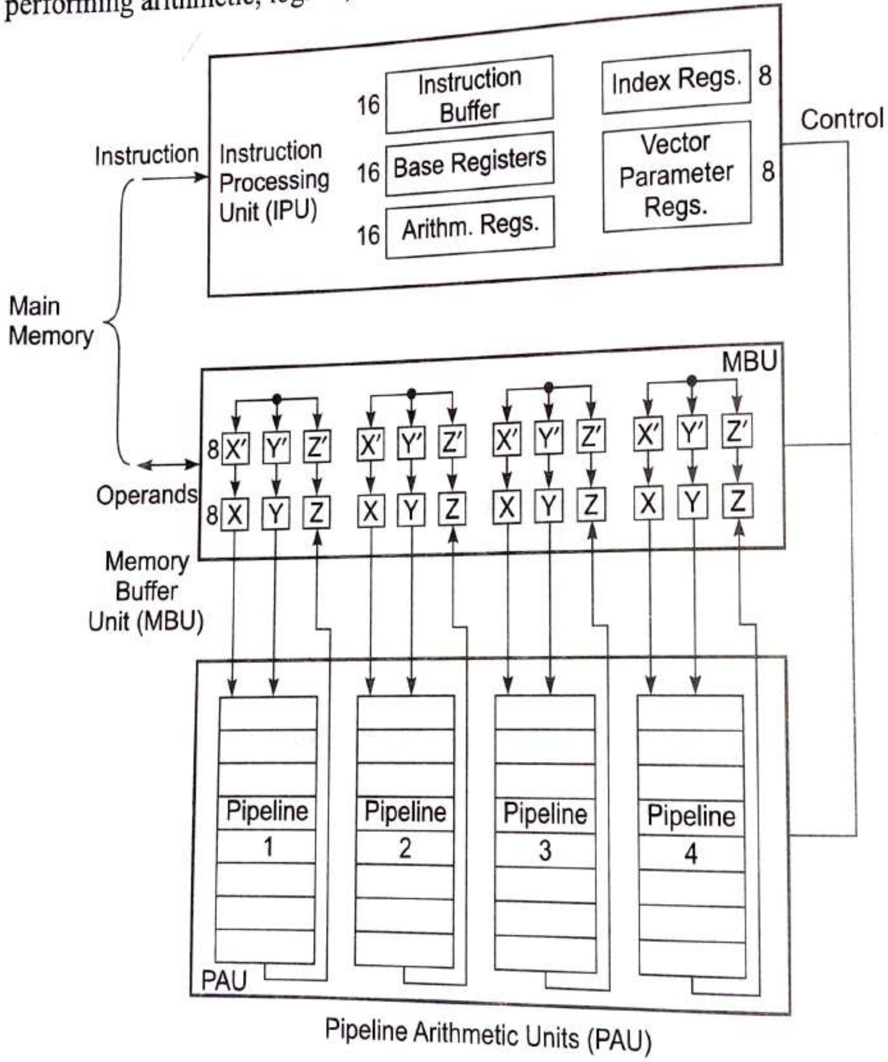
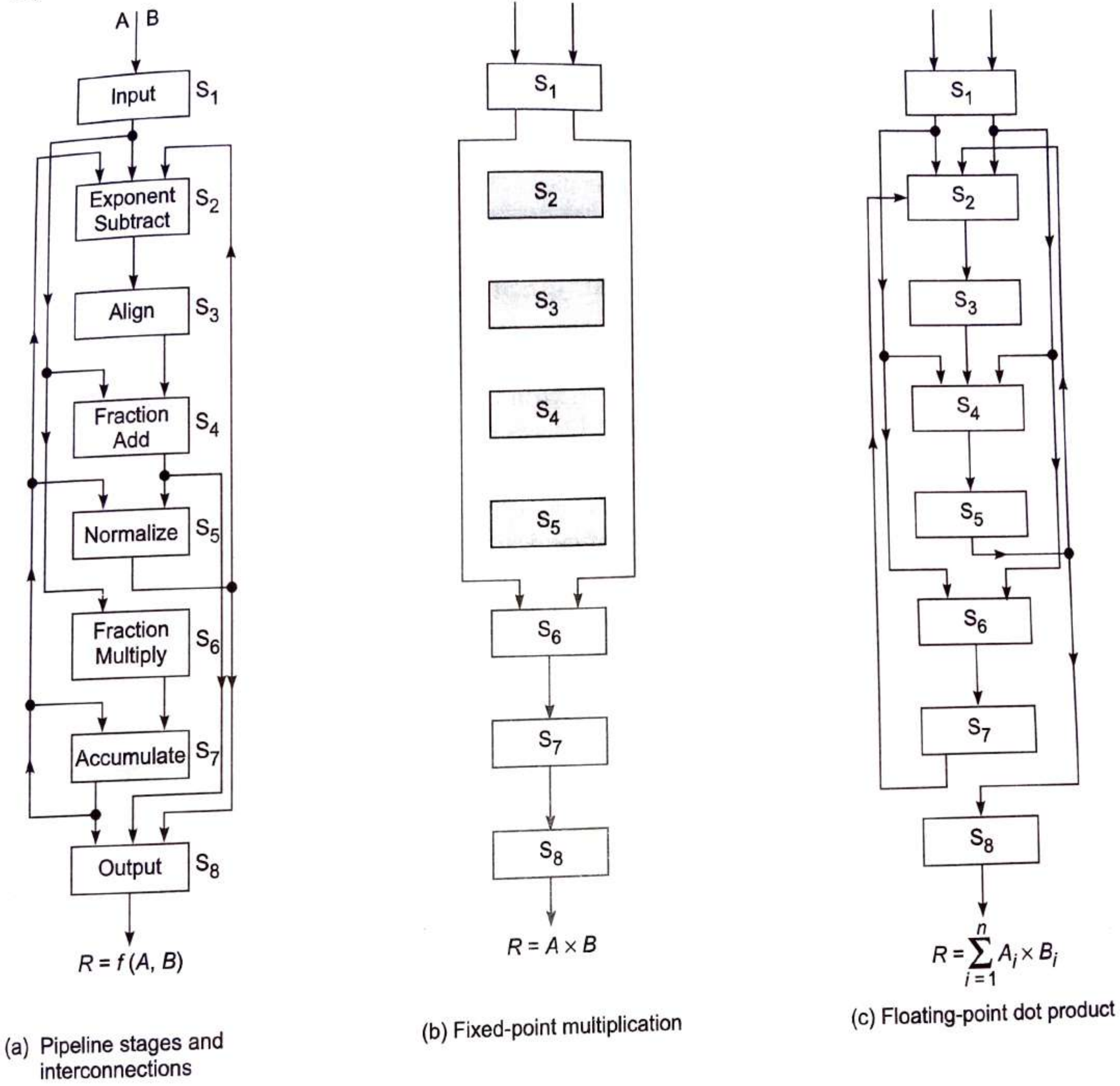


Fig. 6.26 The architecture of the TI Advanced Scientific Computer (ASC) (Courtesy of Texas Instruments, Inc.)

Both fixed-point and floating-point arithmetic functions could be performed by this pipeline. The PAU also supported vector in addition to scalar arithmetic operations. It should be noted that different functions required different pipeline stages and different interstage connection patterns.



(a) Pipeline stages and interconnections

(b) Fixed-point multiplication

(c) Floating-point dot product

Fig. 6.27 The multiplication arithmetic pipeline of the TI Advanced Scientific Computer and the interstage connections of two representative functions (Shaded stages are unutilized)

For example, fixed-point multiplication required the use of only segments S_1, S_6, S_7, and S_8 as shown in Fig. 6.27b. On the other hand, the floating-point dot product function, which performs the dot product operation between two vectors, required the use of all segments with the complex connections shown in Fig. 6.27c. This dot product was implemented by essentially the following accumulated summation of a sequence of multiplications through the pipeline:

$$Z \leftarrow A_i \times B_i + Z \tag{6.25}$$

where the successive operands (A_i, B_i) were fed through the X- and Y-buffers, and the accumulated sums through the Z-buffer recursively.

The entire pipeline could perform the *multiply* (×) and the *add* (+) in a single flow through the pipeline. The two levels of buffer registers isolated the loading and fetching of operands to or from the PAU, respectively, as in the concept of using a pair in the prefetch buffers described in Fig. 6.11.

Even though the TI-ASC is no longer in production, the system provided a unique design for multifunction arithmetic pipelines. Today, most supercomputers implement arithmetic pipelines with dedicated functions for much simplified control circuitry and faster operations.

6.5 SUPERSCALAR PIPELINE DESIGN

Pipeline Design Parameters Some parameters used in designing the scalar base processor and superscalar processor are summarized in Table 6.1 for the pipeline processors to be studied below. All pipelines discussed are assumed to have k stages.

The *pipeline cycle* for the scalar base processor is assumed to be 1 time unit, called the *base cycle*. We defined the instruction *issue rate, issue latency,* and *simple operation latency* in Section 4.1.1. The *instruction-level parallelism* (ILP) is the maximum number of instructions that can be simultaneously executed in the pipeline.

For the base processor, all of these parameters have a value of 1. All processor types are designed relative to the base processor. The ILP is needed to fully utilize a given pipeline processor.

Table 6.1 *Design Parameters for Pipeline Processors*

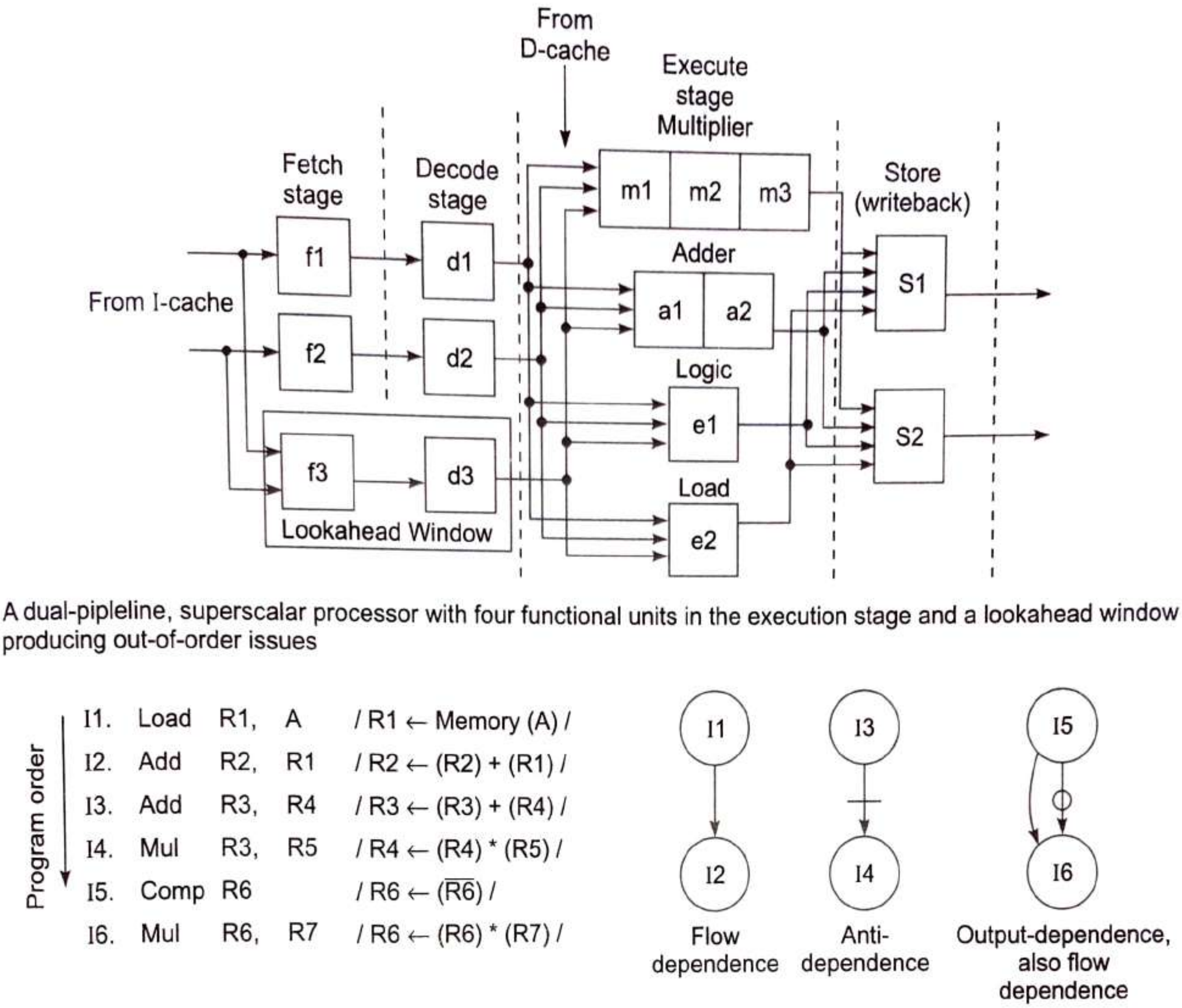
Machine type	*Scalar base machine of k pipeline stages*	*Superscalar machine of degree m*
Machine pipeline cycle	1 (base cycle)	1
Instruction issue rate	1	m
Instruction issue latency	1	1
Simple operation latency	1	1
ILP to fully utilize the pipeline	1	m

Note: All timing is relative to the base cycle for the scalar base machine. ILP: Instruction level parallelism.

We study below the structure of superscalar pipelines, the data dependence problem, the factors causing pipeline stalling, and multi-instruction issuing mechanisms for achieving parallel pipelining operations. 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. As a matter of fact, the scalar base processor can be considered a degenerate case of a superscalar processor of degree 1.

Superscalar Pipeline Structure In an m-issue superscalar processor, the instruction decoding and execution resources are increased to form effectively m pipelines operating concurrently. At some pipeline stages, the functional units may be shared by multiple pipelines.

This resource-shared multiple-pipeline structure is illustrated by a design example in Fig. 6.28a. In this design, the processor can issue two instructions per cycle if there is no resource conflict and no data dependence problem. There are essentially two pipelines in the design. Both pipelines have four processing stages labeled fetch, decode, execute, and store, respectively.



(a) A dual-pipleline, superscalar processor with four functional units in the execution stage and a lookahead window producing out-of-order issues

```
I1.  Load  R1,  A    / R1 ← Memory (A) /
I2.  Add   R2,  R1   / R2 ← (R2) + (R1) /
I3.  Add   R3,  R4   / R3 ← (R3) + (R4) /
I4.  Mul   R3,  R5   / R4 ← (R4) * (R5) /
I5.  Comp  R6        / R6 ← (R6) /
I6.  Mul   R6,  R7   / R6 ← (R6) * (R7) /
```

(b) A sample program and its dependence graph, where I2 and I3 share the adder and I4 and I6 share the multiplier

Fig. 6.28 A two-issue superscalar processor and a sample program for parallel execution

Each pipeline essentially has its own fetch unit, decode unit, and store unit. The two instruction streams flowing through the two pipelines are retrieved from a single source stream (the I-cache). The fan-out from a single instruction stream is subject to resource constraints and a data dependence relationship among the successive instructions.

For simplicity, we assume that each pipeline stage requires one cycle, except the execute stage which may require a variable number of cycles. Four functional units, multiplier, adder, logic unit, and load unit, are available for use in the execute stage. These functional units are shared by the two pipelines on a dynamic basis. The multiplier itself has three pipeline stages, the adder has two stages, and the others each have only one stage.

The two store units (S1 and S2) can be dynamically used by the two pipelines, depending on availability at a particular cycle. There is a *lookahead window* with its own fetch and decoding logic. This window is used for instruction lookahead in case out-of-order instruction issue is desired to achieve better pipeline throughput.

It requires complex logic to schedule multiple pipelines simultaneously, especially when the instructions are retrieved from the same source. The aim is to avoid pipeline stalling and minimize pipeline idle time.

Data Dependences Consider the example program in Fig. 6.28b. A dependence graph is drawn to indicate the relationship among the instructions. Because the register content in R1 is loaded by I1 and then used by I2, we have flow dependence: I1 → I2.

Because the result in register R4 after executing I4 may affect the operand register R4 used by I3, we have antidependence: I3 $\nrightarrow$ I4. Since both I5 and I6 modify the register R6, and R6 supplies an operand for I6, we have both flow and output dependence: I5 → I6 and I5 $\circ\!\!\rightarrow$ I6 as shown in the dependence graph.

To schedule instructions through one or more pipelines, these data dependences must not be violated. Otherwise, erroneous results may be produced.

Pipeline Stalling This is a problem which may seriously lower pipeline utilization. Proper scheduling avoids pipeline stalling. The problem exists in both scalar and superscalar processors. However, it is more serious in a superscalar pipeline. Stalling can be caused by data dependences or by resource conflicts among instructions already in the pipeline or about to enter the pipeline. We use an example to illustrate the conditions causing pipeline stalling.

Consider the scheduling of two instruction pipelines in a two-issue superscalar processor. Figure 6.29a shows the case of no data dependence on the left and flow dependence (I1 → I2) on the right. Without data dependence, all pipeline stages are utilized without idling.

With dependence, instruction I2 entering the second pipeline must wait for two cycles (shaded time slots) before entering the execution stages. This delay may also pass to the next instruction I4 entering the pipeline.

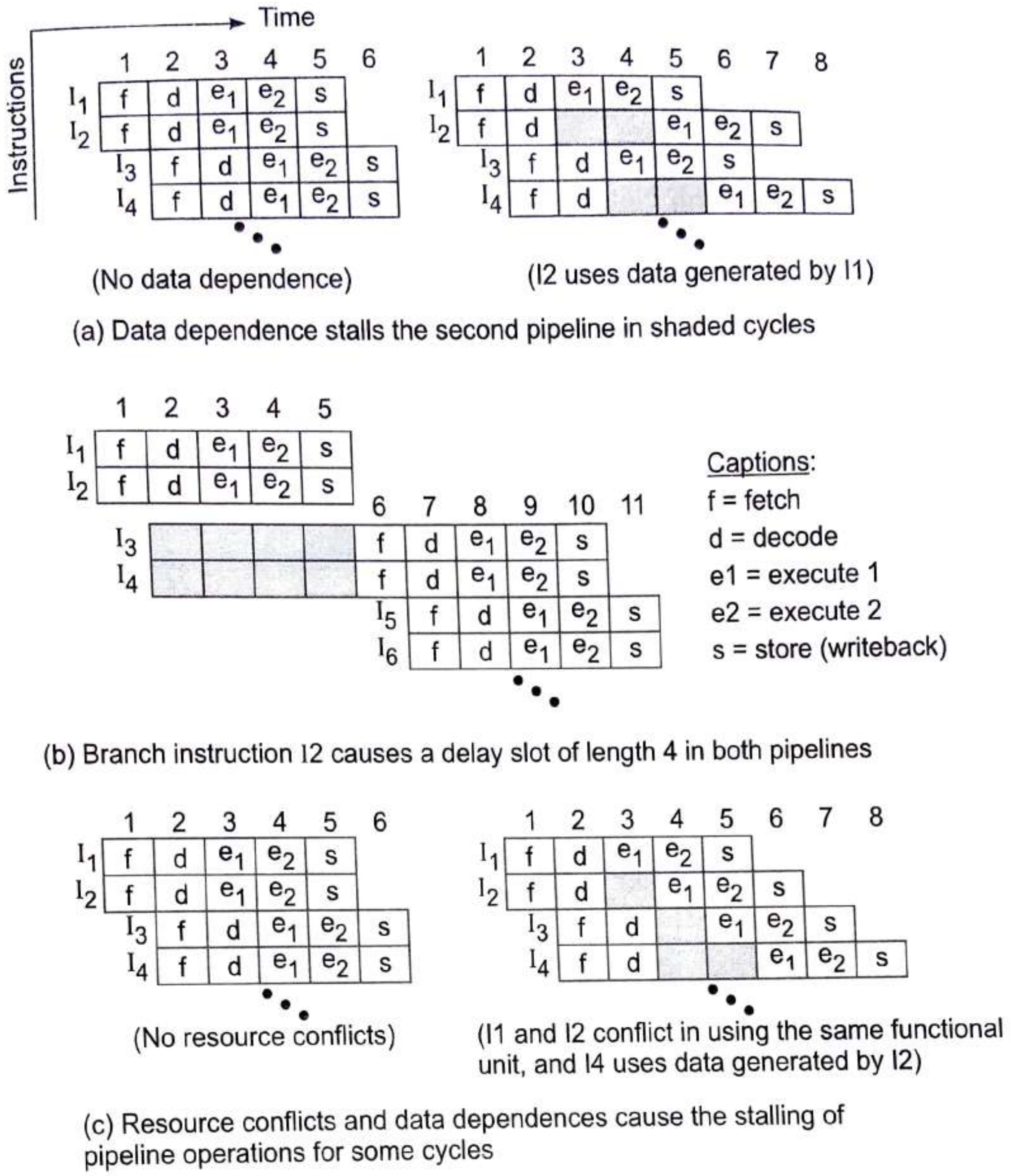
In Fig. 6.29b, we show the effect of branching (instruction I2). A delay slot of four cycles results from a branch taken by I2 at cycle 5. Therefore, both pipelines must be flushed before the target instructions I3 and I4 can enter the pipelines from cycle 6. Here, delayed branch or other amending actions are not taken.

In Fig. 6.29c, we show a combined problem involving both resource conflict and data dependence. Instructions I1 and I2 need to use the same functional unit, and I2 → I4 exists.

The net effect is that I2 must be scheduled one cycle behind because the two pipeline stages (e_1 and e_2) of the same functional unit must be used by I1 and I2 in an overlapped fashion. For the same reason, I3 is also delayed by one cycle. Instruction I4 is delayed by two cycles due to the flow dependence on I2. The shaded boxes in all the timing charts correspond to idle stages.

Superscalar Pipeline Scheduling Instruction issue and completion policies are critical to superscalar processor performance. Three scheduling policies are introduced below. When instructions are issued in program order, we call it *in-order issue.* When program order is violated, *out-of-order issue* is being practiced.

Similarly, if the instructions must be completed in program order, it is called *in-order completion.* Otherwise, *out-of-order completion* may result. In-order issue is easier to implement but may not yield the optimal performance. In-order issue may result in either in-order or out-of-order completion.



(No data dependence)

(I2 uses data generated by I1)

(a) Data dependence stalls the second pipeline in shaded cycles

(b) Branch instruction I2 causes a delay slot of length 4 in both pipelines

(No resource conflicts)

(I1 and I2 conflict in using the same functional unit, and I4 uses data generated by I2)

(c) Resource conflicts and data dependences cause the stalling of pipeline operations for some cycles

Fig. 6.29 Dependences and resource conflicts may stall one or two pipelines in a two-issue superscalar processor

Out-of-order issue usually ends up with out-of-order completion. The purpose of out-of-order issue and completion is to improve performance. These three scheduling policies are illustrated in Fig. 6.30 by execution of the example program in Fig. 6.28b on the dual-pipeline hardware in Fig. 6.28a.

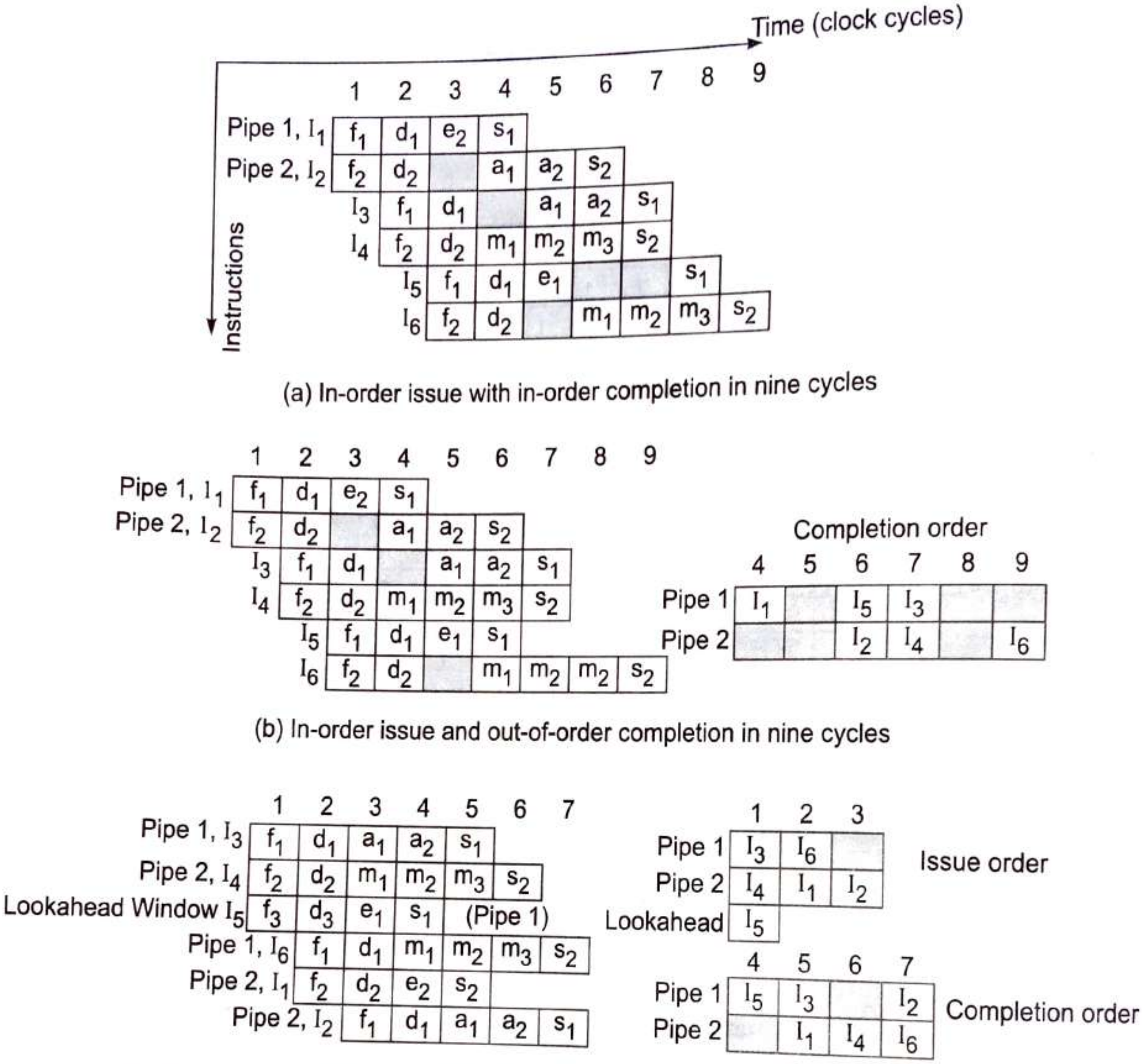
It is demonstrated that performance can be improved from an in-order to an out-of-order schedule. The performance is often indicated by the total execution time and the utilization rate of pipeline stages. Not all programs can be scheduled out of order. Data dependence and resource conflicts do impose constraints.

In-Order Issue Figure 6.30a shows a schedule for the six instructions being issued in program order I1, I2, …, I6. Pipeline 1 receives I1, I3, and I5, and pipeline 2 receives I2, I4, and I6 in three consecutive cycles. Due to I1 → I2, I2 has to wait one cycle to use the data loaded in by I1.

I3 is delayed one cycle for the same adder used by I2. I6 has to wait for the result of I5 before it can enter the multiplier stages. In order to maintain in-order completion, I5 is forced to wait for two cycles to come out of pipeline 1. In total, nine cycles are needed and five idle cycles (shaded boxes) are observed.

In Fig. 6.30b, out-of-order completion is allowed even if in-order issue is practiced. 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. The total execution time does not improve. However, the pipeline utilization rate does.

Only three idle cycles are observed. Note that in Figs. 6.29a and 6.29b, we did not use the lookahead window. In order to shorten the total execution time, the window can be used to reorder the instruction issues.



(a) In-order issue with in-order completion in nine cycles

(b) In-order issue and out-of-order completion in nine cycles

(c) Out-of-order issue and out-of-order completion in seven cycles using an instruction lookahead window in the recoding process

Fig. 6.30 Instruction issue and completion policies for a superscalar processor with and without instruction lookahead support (Timing charts correspond to parallel execution of the program in Fig. 6.28)

Out-of-Order Issue By using the lookahead window, instruction I5 can be decoded in advance because it is independent of all the other instructions. The six instructions are issued in three cycles as shown: I5 is fetched and decoded by the window, while I3 and I4 are decoded concurrently.

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 Fig. 6.30c. Now, the total execution time has been reduced to seven cycles with no idle stages during the execution of these six instructions.

The in-order issue and completion is the simplest one to implement. It is rarely used today even in a conventional scalar processor due to some unnecessary delays in maintaining program order. However, in a multiprocessor environment, this policy is still attractive. Allowing out-of-order completion can be found in both scalar and superscalar processors.

Some long-latency operations, such as loads and floating-point operations, can be hidden in out-of-order completion to achieve a better performance. Output dependence and antidependence are the two relations preventing out-of-order completion. Out-of-order issue gives the processor more freedom to exploit parallelism, and thus pipeline efficiency is enhanced.

The above example clearly demonstrates the advantages of instruction lookahead and of out-of-order issue and completion as far as pipeline optimization is concerned. It should be noted that multiple-pipeline scheduling is an NP-complete problem. Optimal scheduling is very expensive to obtain.

Simple data dependence checking, a small lookahead window, and scoreboarding mechanisms are needed, along with an optimizing compiler, to exploit instruction parallelism in a superscalar processor.

Motorola 88110 Architecture The Motorola 88110 was an early superscalar RISC processor. It combined the three-chip set, one CPU (88100) chip and two cache (88200) chips, in a single-chip implementation, with additional improvements. The 88110 employed advanced techniques for exploiting instruction-level parallelism, including instruction issue, out-of-order instruction completion, speculative execution, dynamic instruction rescheduling, and two on-chip caches. The unit also supported demanding graphics and digital signal processing applications.

The 88110 employed a symmetrical superscalar instruction dispatch unit which dispatched two instructions each clock cycle into an array of 10 concurrent units. It allowed out-of-order instruction completion and some out-of-order instruction issue, and branch prediction with speculative execution past branches.

The instruction set of the 88110 extended that of the 88100 in integer and floating-point operations. It added a new set of capabilities to support 3-D color graphics image rendering. The 88110 had separate, independent instruction and data paths, along with split caches for instructions and data. The instruction cache was 8K-byte, 2-way set-associative with 128 sets, two blocks for each set, and 32 bytes (8 instructions) per block. The data cache resembled that of the instruction set.

The 88110 employed the MESI cache coherence protocol. A write-invalidate procedure guaranteed that one processor on the bus had a modified copy of any cache block at any time. The 88110 was implemented with 1.3 million transistors in a 299-pin package and driven by a 50-MHz clock. Interested readers may refer to Diefendorff and Allen (1992) for details.

Superscalar Performance To compare the relative performance of a superscalar processor with that of a scalar base machine, we estimate the ideal execution time of N independent instructions through the pipeline.

The time required by the scalar base machine is

$$T(1, 1) = k + N - 1 \text{ (base cycles)} \tag{6.26}$$

The ideal execution time required by an m-issue superscalar machine is

$$T(m, 1) = k + \frac{N - m}{m} \text{ (base cycles)} \tag{6.27}$$

where k is the time required to execute the first m instructions through the m pipelines simultaneously, and the second term corresponds to the time required to execute the remaining $N - m$ instructions, m per cycle, through m pipelines.

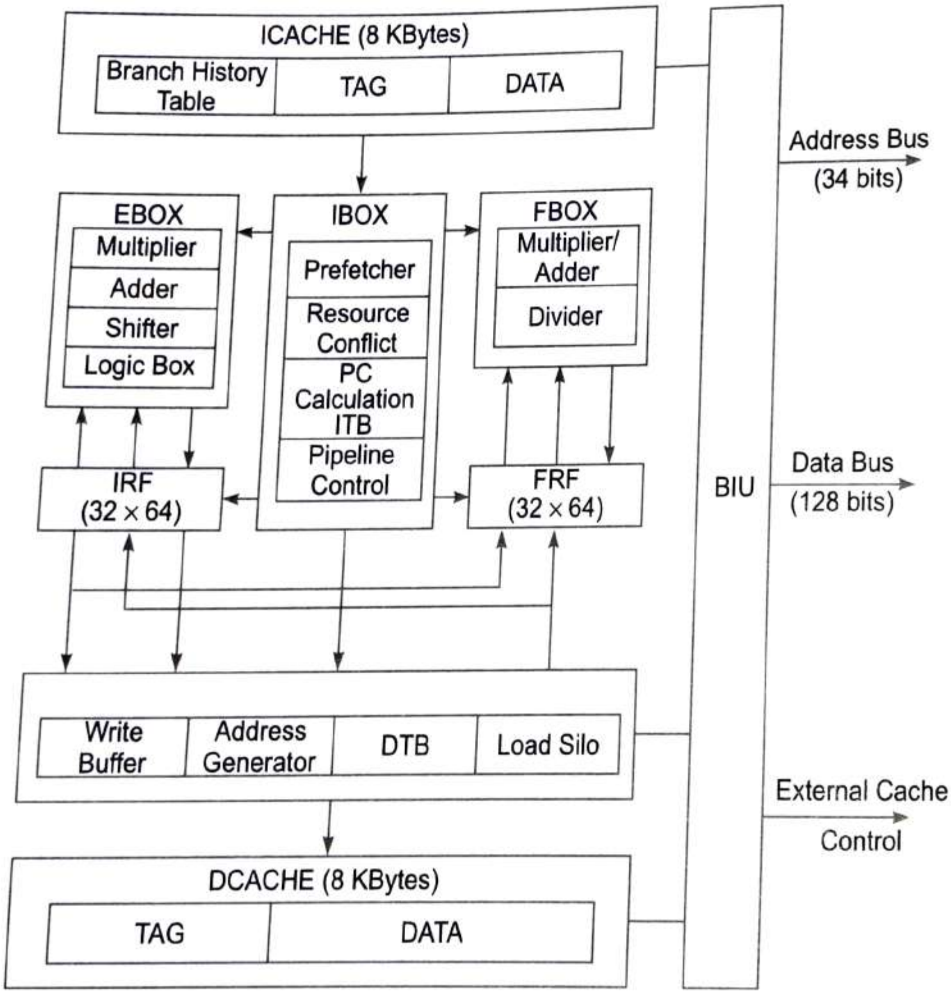
The ideal speedup of the superscalar machine over the base machine is

$$S(m, 1) = \frac{T(1,1)}{T(m,1)} = \frac{N+k-1}{N/m+k-1} = \frac{m(N+k-1)}{N+m(k-1)} \qquad (6.28)$$

As $N \to \infty$, the speedup limit $S(m, 1) \to m$, as expected.

Example 6.13 DEC Alpha 21064 superscalar architecture

As illustrated in Fig. 6.31, this was a 64-bit superscalar processor. The design emphasized speed, multiple-instruction issue, multiprocessor applications, software migration from the VAX/VMS and MIPS/OS, and a long list of usable features. The clock rate was 150 MHz with the first chip implementation.



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

Fig. 6.31 Architecture of the DEC Alpha 21064 processor (Courtesy of Digital Equipment Corporation)

Unlike others, the Alpha architecture had thirty-two 64-bit integer registers and thirty-two 64-bit floating-point registers. The integer pipeline had 7 stages, and the floating-point pipeline had 10 stages. All Alpha instructions were 32 bits.

The first Alpha implementation issued two instructions per cycle, with larger number of issues in later implementations. Pipeline timing hazards, load delay slots, and branch delay slots were all minimized by hardware support. The Alpha was designed to support fast multiprocessor interlocking and interrupts.

A privileged library of software was developed to run full VMS and to run OSF/1 using different versions of the software library that mirrored many of the VAX/VMS and MIPS/OS features, respectively. This library made Alpha an attractive architecture for multiple operating systems. The processor was designed to have a 300-MIPS peak and a 150-Mflops peak at 150 MHz.

Note 6.2 *Innovation versus commercial success*

The relationship between innovative design ideas and the commercial success of a product is not always simple, as an idealist may believe.

Most of the processors used as examples in this chapter are no longer in commercial production. Rapid advances in technology and immense pressures from the market-place are usually the two main reasons behind the introduction and the demise of newer processor models. However, the innovative design ideas introduced in a new processor often have a life longer than the processor itself, since these same ideas are often carried forward in subsequent designs of the same or other processor families.

For example, IBM 360/91, Motorola 68040, Motorola 88110 and DEC Alpha 21064 were all recognized for their innovative designs when they were introduced, but they achieved different degrees of commercial success. Our aim in this book is to study the innovative ideas embodied in processor and system designs; but we must also appreciate that the commercial success of a product often depends on many other crucial factors.

Summary

Instruction pipelines in processors usually have a linear structure—the execution of each instruction progresses linearly, one stage at a time, from the first to the last pipeline stage. In theory, such a linear pipeline can be designed with synchronous or asynchronous timing mode; in practice, processor pipelines today operate in synchronous mode, i.e. with a common clock signal. We studied the timing and clocking requirements of linear pipelines, and discussed the related speedup, efficiency and throughput issues. A simple model was presented which can be used in determining the optimal number of pipeline stages, based on a trade-off between cost and throughput.

Dynamic or nonlinear pipelines are designed to perform a number of different functions, by appropriate scheduling of operations on the pipeline stages. Reservation tables are used for different functions; collision free schedules and latency analysis are needed for efficient operation of nonlinear pipelines. We studied how concepts of collision vectors, state transition diagrams and greedy cycles are used to determine bounds on minimum average latency (MAL), and thereby optimum schedules in terms of MAL

CHAPTER FIVE

STRUCTURES AND ALGORITHMS FOR ARRAY PROCESSORS

This chapter deals with the interconnection structures and parallel algorithms for SIMD array processors and associative processors. The various organizations and control mechanisms of array processors are presented first. Interconnection networks used in array processors will be characterized by their routing functions and implementation methods. We then study the structure of associative memory and parallel search in associative array processors. SIMD algorithms are presented for matrix manipulation, parallel sorting, fast Fourier transform, and associative search and retrieval operations.

5.1 SIMD ARRAY PROCESSORS

A synchronous array of parallel processors is called an *array processor*, which consists of multiple processing elements (PEs) under the supervision of one control unit (CU). An array processor can handle *single instruction and multiple data* (SIMD) streams. In this sense, array processors are also known as *SIMD computers*. SIMD machines are especially designed to perform vector computations over matrices or arrays of data. In this book, the terms array processors, parallel processors, and SIMD computers are used interchangeably.

SIMD computers appear in two basic architectural organizations: *array processors*, using random-access memory; and *associative processors*, using content-addressable (or associative) memory. The first three sections of this chapter deal primarily with array processors. We will study associative processors in Section 5.4 as a special type of array processor whose PEs correspond to the words of an associative memory.

5.2 SIMD INTERCONNECTION NETWORKS

Various interconnection networks have been suggested for SIMD computers. In this section, we distinguish between single-stage, recirculating networks and multistage SIMD networks. Important network classes to be presented include the Illiac network, the flip network, the *n* cube, the Omega network, the data manipulator, the barrel shifter, and the shuffle-exchange network. We shall concentrate on inter-PE communications as modeled by configuration I in Figure 5.1. The interprocessor-memory communication networks will be studied in Chapter 7 for MIMD operations.

5.2.1 Static Versus Dynamic Networks

The topological structure of an SIMD array processor is mainly characterized by the data-routing network used in interconnecting the processing elements. Formally, such an inter-PE communication network can be specified by a set of data-routing functions. If we identify the addresses of all the PEs in an SIMD machine by the set $S = \{0, 1, 2, \ldots, N - 1\}$, each routing function f is a *bijection* (a one-to-one and onto mapping) from S to S. When a routing function f is executed via the interconnection network, the PE_i copies the contents of its R_i register into the $R_{f(i)}$ register of $PE_{f(i)}$. This data-routing operation occurs in all active PEs simultaneously. An inactive PE may receive data from another PE if a routing function is executed, but it cannot transmit data. To pass data between PEs that are not directly connected in the network, the data must be passed through intermediate PEs by executing a sequence of routing functions through the interconnection network.

The SIMD interconnection networks are classified into the following two categories based on network topologies: *static networks* and *dynamic networks.*

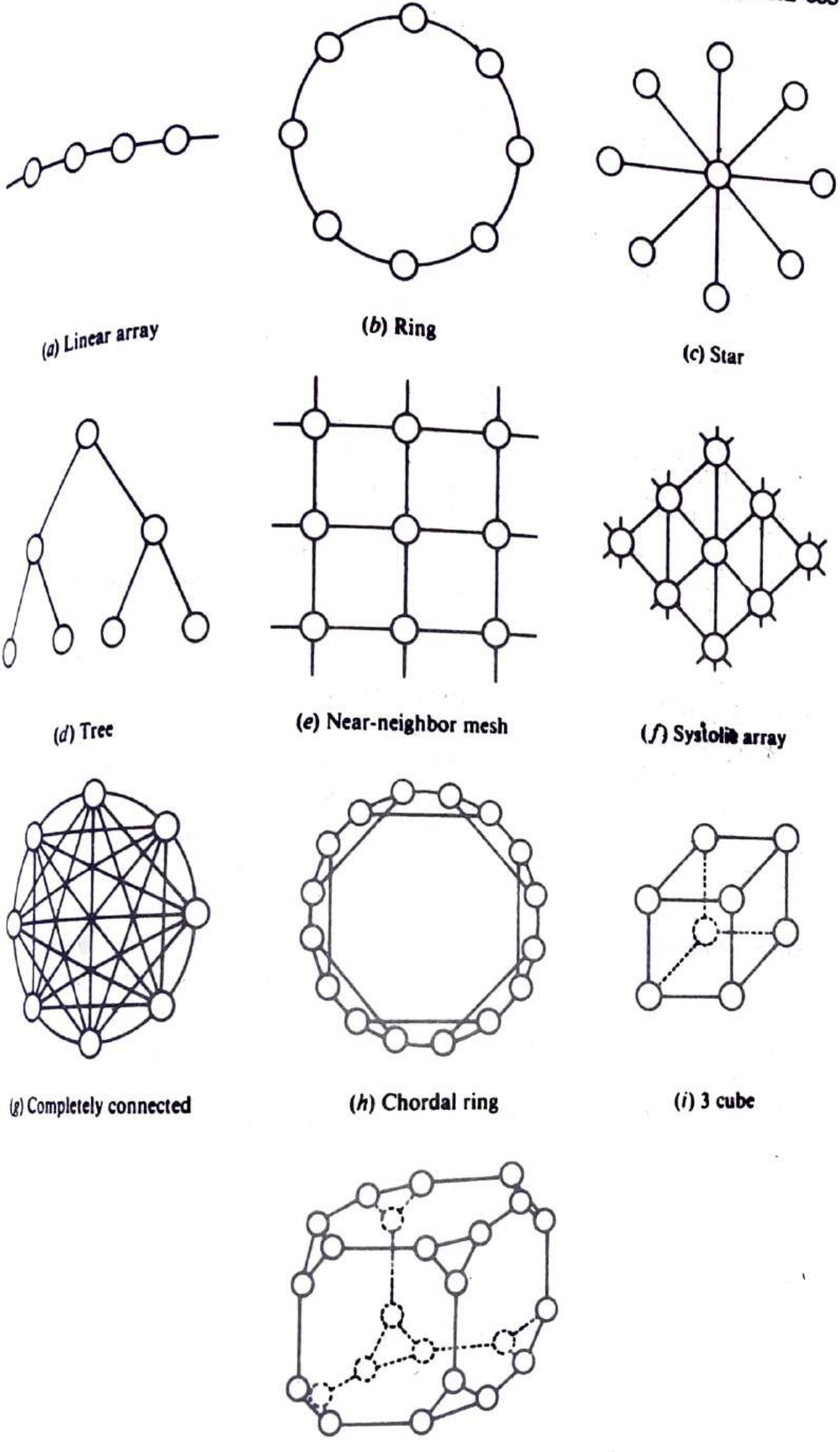
Static networks Topologies in the static networks can be classified according to the dimensions required for layout. For illustration, one-dimensional, two-dimensional, three-dimensional, and hypercube are shown in Figure 5.4. Examples of one-dimensional topologies include the *linear array* used for some pipeline architectures (Figure 5.4*a*). Two-dimensional topologies include the *ring*, *star*, *tree*, *mesh*, and *systolic array*. Examples of these structures are shown in Figures 5.4*b* through 5.4*f*.

Three-dimensional topologies include the *completely connected chordal ring*, 3 *cube*, and 3-*cube-connected-cycle* networks depicted in Figures 5.4*g* through 5.4*j*. A D-dimensional, W-wide hypercube contains W nodes in each dimension, and there is a connection to a node in each dimension. The mesh and the 3 cube are actually two- and three-dimensional hypercubes, respectively. The cube-connected-cycle is a deviation of the hypercube. For example, the 3-cube-connected-cycle shown in Figure 5.4*j* is obtained from the 3 cube.

Dynamic networks We consider two classes of dynamic networks: *single-stage* versus *multistage*, as described below separately:

Single-stage networks A single-stage network is a switching network with N *input selectors* (IS) and N *output selectors* (OS), as demonstrated in Figure 5.5. Each IS is essentially a 1-to-D demultiplexer and each OS is an M-to-1 multiplexer where $1 \leq D \leq N$ and $1 \leq M \leq N$. Note that the crossbar-switching network is a single-stage network with $D = M = N$. To establish a desired connecting path, different path control signals will be applied to all IS and OS selectors.

The single-stage network is also called a *recirculating* network. Data items may have to recirculate through the single stage several times before reaching their final destinations. The number of recirculations needed depends on the



(*a*) Linear array

(*b*) Ring

(*c*) Star

(*d*) Tree

(*e*) Near-neighbor mesh

(*f*) Systolic array

(*g*) Completely connected

(*h*) Chordal ring

(*i*) 3 cube

(*j*) 3-cube-connected cycle

Figure 5.4 Static interconnection network topologies. (Courtesy of Feng, *IEEE Computer*, December 1981.)

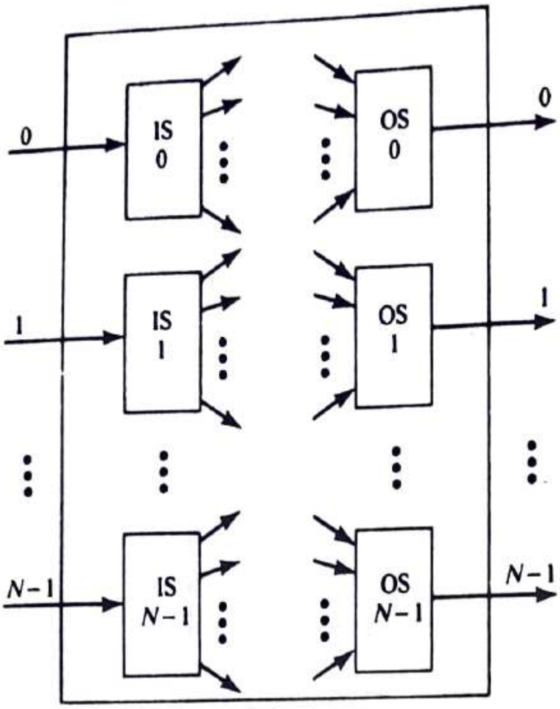


Figure 5.5 Conceptual view of a single-stage interconnection network.

connectivity in the single-stage network. In general, the higher is the hardware connectivity, the less is the number of recirculations. The crossbar network is an extreme case in which only one circulation is needed to establish any connection path. However, the fully connected crossbar networks have a cost $O(N^2)$, which may be prohibitive for large N. Most recirculating networks have cost $O(N \log N)$ or lower, which is definitely more cost-effective for large N.

Multistage networks Many stages of interconnected switches form a *multistage SIMD network*. Multistage networks are described by three characterizing features: the *switch box*, the *network topology*, and the *control structure*. Many switch boxes are used in a multistage network. Each box is essentially an interchange device with two inputs and two outputs, as depicted in Figure 5.6. Illustrated are four states of a switch box: *straight*, *exchange*, *upper broadcast*, and *lower broadcast*. A two-function switch box can assume either the straight or the exchange states. A four-function switch box can be in any one of the four legitimate states.

A multistage network is capable of connecting an arbitrary input terminal to an arbitrary output terminal. Multistage networks can be one-sided or two-sided. The *one-sided networks*, sometimes called full switches, have input-output ports on the same side. The *two-sided multistage networks*, which usually have an input side and an output side, can be divided into three classes: blocking, rearrangeable, and nonblocking.

In *blocking networks*, simultaneous connections of more than one terminal pair may result in conflicts in the use of network communication links. Examples of a blocking network are the *data manipulator*, *Omega*, *flip*, *n cube*, and *baseline*.

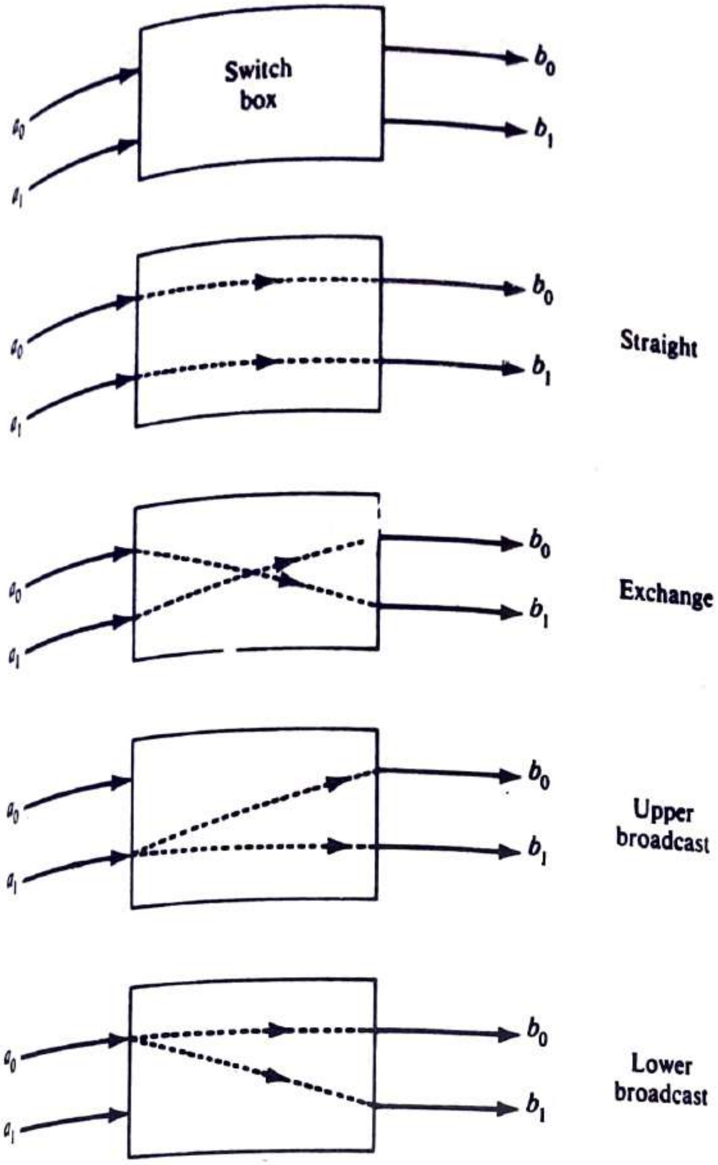
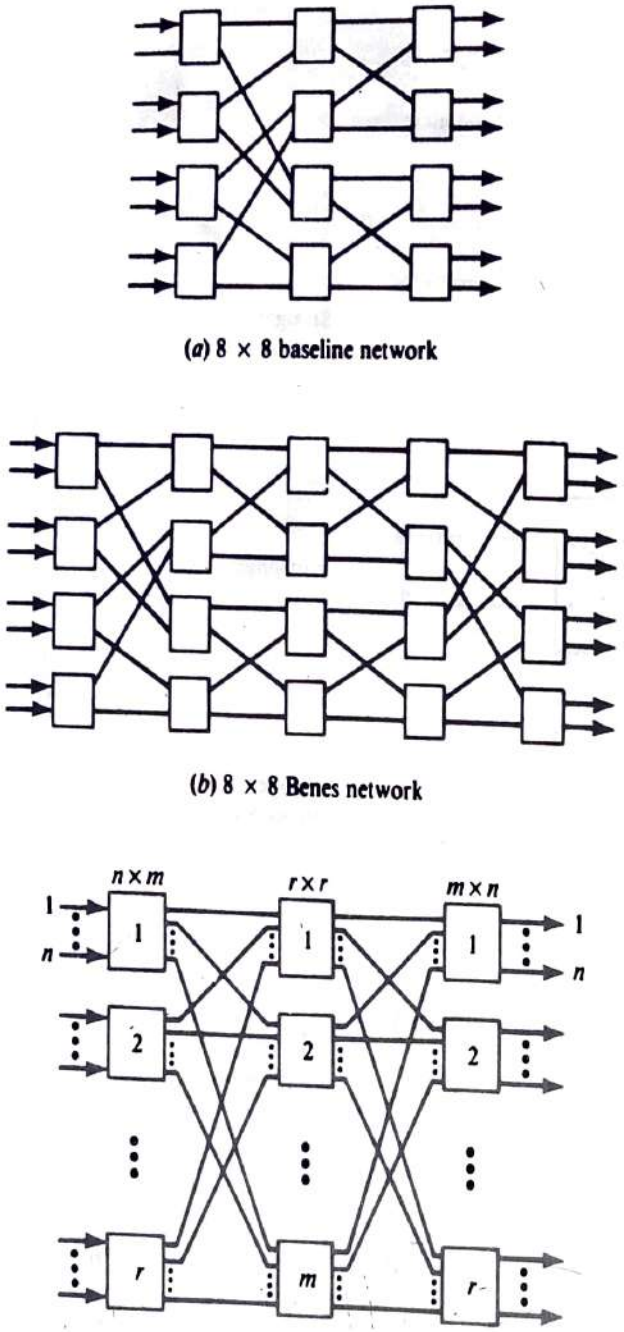


Figure 5.6 A two-by-two switching box and its four interconnection states.

Some of these networks will be introduced in subsequent sections. Figure 5.7*a* shows the interconnection pattern in the baseline network.

A network is called a *rearrangeable network* if it can perform all possible connections between inputs and outputs by rearranging its existing connections so that a connection path for a new input-output pair can always be established. A well-defined network, the Benes network, shown in Figure 5.7*b*, belongs to this class. The Benes rearrangeable network topology has been extensively studied for use in synchronous data permutation and in asynchronous interprocessor communication.

A network which can handle all possible connections without blocking is called a *nonblocking network*. Two cases have been considered in the literature. In the first case, the *Clos network*, shown in Figure 5.7*c*, a one-to-one connection is made between an input and an output. The other case considers one-to-many connections. Here, a generalized connection network topology is generated to



(*a*) 8 × 8 baseline network

(*b*) 8 × 8 Benes network

(*c*) Clos network

Figure 5.7 Several multistage interconnection networks.

pass any of multiple mappings of inputs onto outputs. The *crossbar switch network* can connect every input port to a free output port without blocking.

Generally, a multistage network consists of n stages where $N = 2^n$ is the number of input and output lines. Therefore, each stage may use $N/2$ switch boxes. The interconnection patterns from stage to stage determine the network topology. Each stage is connected to the next stage by at least N paths. The network delay is proportional to the number n of stages in a network. The cost of a size N multistage network is proportional to $N \log_2 N$. The control structure of a network determines how the states of the switch boxes will be set. Two types of control structures are used in a network construction. The *individual stage control* uses the same control signal to set all switch boxes in the same stage. In other words, all boxes at the same stage must be set to assume that same state. Therefore, it requires n sets of control signals to set up the states of all n stages of switch boxes.

Another control philosophy is to apply *individual box control*. A separate control signal is used to set the state of each switch box. This offers higher flexibility in setting up the connecting paths, but requires $n^2/2$ control signals, which will significantly increase the complexity of the control circuitry. A compromise design is to use *partial stage control*, in which $i + 1$ control signals are used at stage i for $0 \leq i \leq n - 1$. Various network topologies and control structures of both recirculating and multistage inter-PE communication networks are described in subsequent sections.

5.2.2 Mesh-Connected Illiac Network

A single-stage recirculating network has been implemented in the Illiac-IV array processor with $N = 64$ PEs. Each PE_i is allowed to send data to any one of PE_{i+1}, PE_{i-1}, PE_{i+r}, and PE_{i-r} where $r = \sqrt{N}$ (for the case of the Illiac-IV, $r = \sqrt{64} = 8$) in one circulation step through the network. Formally, the Illiac network is characterized by the following four routing functions:

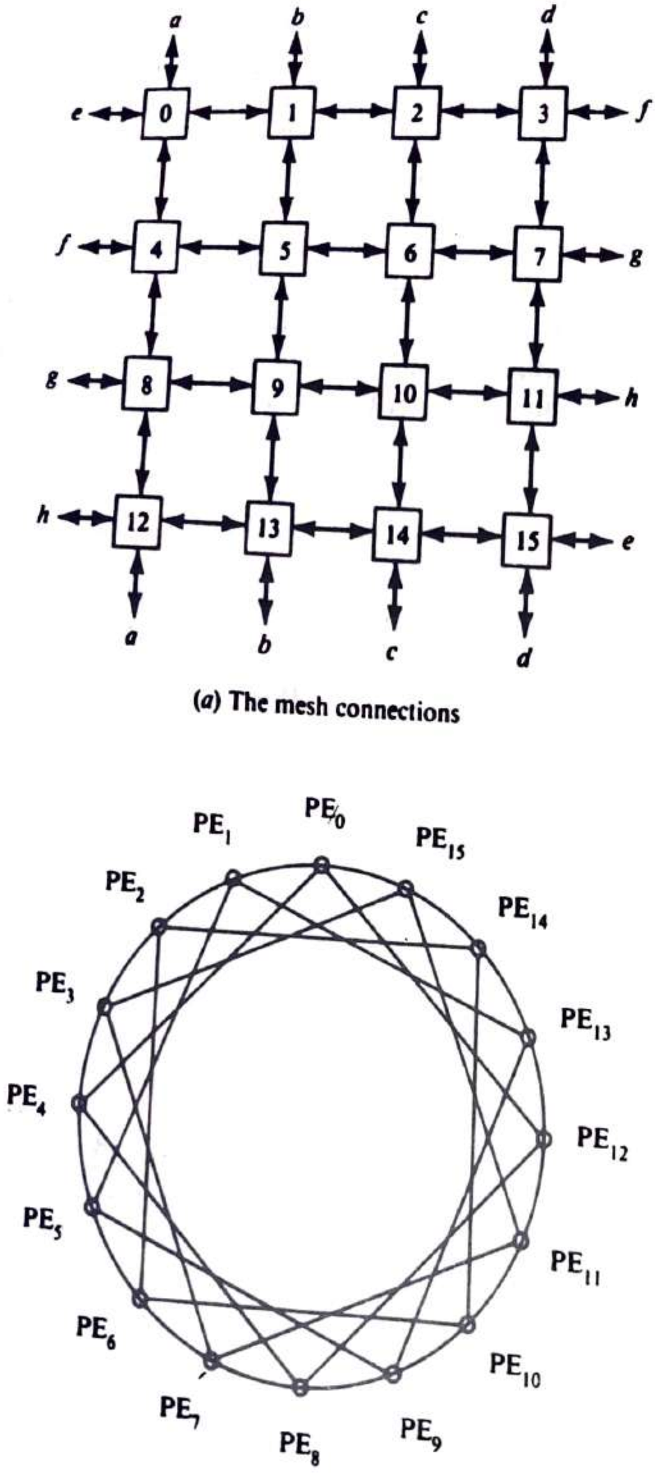
$$\begin{aligned} R_{+1}(i) &= (i + 1) \bmod N \\ R_{-1}(i) &= (i - 1) \bmod N \\ R_{+r}(i) &= (i + r) \bmod N \\ R_{-r}(i) &= (i - r) \cdot \bmod N \end{aligned} \tag{5.5}$$

where $0 \leq i \leq N - 1$. In practice, N is commonly a perfect square, such as $N = 64$ and $r = 8$ in the Illiac-IV network.

A reduced Illiac network is illustrated in Figure 5.8*a* for $N = 16$ and $r = 4$. The real Illiac network has a similar structure except larger in size. All the index arithmetic in Eq. 5.5 is modulo N. Comparing with the formal model shown in Figure 5.5, we observe that the outputs of IS_i are connected to the inputs of OS_j for $j = i + 1, i - 1, i + r$, and $i - r$. On the other hand, OS_j gets its inputs from IS_i for $i = j - 1, j + 1, j - r$, and $j + r$, respectively.

Each PE_i in Figure 5.8 is directly connected to its four nearest neighbors in the mesh network. In terms of permutation cycles, we can express the above routing functions as follows: Horizontally, all the PEs of all rows form a linear circular list as governed by the following two permutations, each with a single cycle of order N. The permutation cycles $(a\ b\ c)\ (d\ e)$ stand for the permutation $a \to b$, $b \to c$, $c \to a$ and $d \to e$, $e \to d$ in a circular fashion within each pair of parentheses:

$$\begin{aligned} R_{+1} &= (0\ 1\ 2\ \cdots\ N-1) \\ R_{-1} &= (N-1\ \cdots\ 2\ 1\ 0) \end{aligned} \tag{5.6}$$



(*a*) The mesh connections

(*b*) The mesh redrawn

Figure 5.8 An Illiac network with $N = 16$ PEs.

Vertically, the distance r shifting operations are characterized by the following two permutations, each with r cycles of order r each:

$$R_{+r} = \prod_{i=0}^{r-s} (i \;\; i+r \;\; i+2r \;\; \cdots \;\; i+N-r)$$

$$R_{-r} = \prod_{i=0}^{r-1} (i+N-r \;\; \cdots \;\; i+2r \;\; i+r \;\; i) \tag{5.7}$$

For the example network of $N = 16$ and $r = \sqrt{16} = 4$, the shift by a distance of four is specified by the following two permutations, each with four cycles of order four each:

$$R_{+4} = (0 \;\; 4 \;\; 8 \;\; 12)(1 \;\; 5 \;\; 9 \;\; 13)(2 \;\; 6 \;\; 10 \;\; 14)(3 \;\; 7 \;\; 11 \;\; 15)$$

$$R_{-4} = (12 \;\; 8 \;\; 4 \;\; 0)(13 \;\; 9 \;\; 5 \;\; 1)(14 \;\; 10 \;\; 6 \;\; 2)(15 \;\; 11 \;\; 7 \;\; 3)$$

It should be noted that when either the R_{+1} or R_{-1} routing function is executed, data is routed as described in Eq. 5.6 only if all PEs in the cycle are active. When the routing function R_{+r} or R_{-r} is executed, data are permuted as described in Eq. 5.7 only if PE_{i+kr} where $0 \le k \le r - 1$ are active for each i. The shifting operation in a cycle will be suspended if any PE required in the cycle is disabled. For an example, the cycle (1 5 9 13) in the above permutation R_4 will not be executed if one or more among PE_1, PE_5, PE_9, and PE_{13} is disabled by masking.

The Illiac network is only a partially connected network. Figure 5.8*b* shows the connectivity of the example Illiac network with $N = 16$. This graph shows that four PEs can be reached from any PE in one step, seven PEs in two steps, and eleven PEs in three steps. In general, it takes I steps (recirculations) to route data from PE_i to any other PE_j in an Illiac network of size N where I is upper-bounded by

$$I \le \sqrt{N} - 1 \tag{5.8}$$

Without a loss of generality, we illustrate the cases when PE_0 is a source node in Figure 5.8. PE_1, PE_4, PE_{12}, or PE_{15} is reachable in one step from PE_0. In two steps, the network can route data from PE_0 to PE_2, PE_3, PE_5, PE_8, PE_{11}, PE_{13}, or PE_{14}. In the worst case of three routing steps, the following eight routing sequences take place in the network:

$$0 \xrightarrow{R_{+1}} 1 \xrightarrow{R_{+1}} 2 \xrightarrow{R_{+4}} 6 \qquad 0 \xrightarrow{R_{+4}} 4 \xrightarrow{R_{+4}} 8 \xrightarrow{R_{-1}} 7$$

$$0 \xrightarrow{R_{-4}} 12 \xrightarrow{\dot{R}_{-1}} 11 \xrightarrow{R_{-4}} 7 \qquad 0 \xrightarrow{R_{-4}} 12 \xrightarrow{R_{-4}} 8 \xrightarrow{R_{-1}} 7$$

$$0 \xrightarrow{R_{-4}} 12 \xrightarrow{R_{-4}} 8 \xrightarrow{R_{+1}} 9 \qquad 0 \xrightarrow{R_{-1}} 15 \xrightarrow{R_{-4}} 11 \xrightarrow{R_{-1}} 10$$

$$0 \xrightarrow{R_{+1}} 1 \xrightarrow{R_{-4}} 13 \xrightarrow{R_{-4}} 9 \qquad 0 \xrightarrow{R_{-1}} 15 \xrightarrow{R_{-1}} 14 \xrightarrow{R_{-1}} 10$$

In the Illiac-IV computer, at most seven ($\sqrt{64} - 1$) steps are needed to route data from any one PE to another PE. Of course, if we increase the connectivity in Figure 5.8, the upper bound given in Eq. 5.8 can be lowered. We shall demonstrate this by other network types in subsequent sections. When the network is strongly connected (i.e., with 15 outgoing links per node in Figure 5.8), the upper bound on recirculation steps can be reduced to one at the expense of significantly increased hardware in the crossbar network.

5.4 ASSOCIATIVE ARRAY PROCESSING

Two SIMD computers, the Goodyear Aerospace STARAN and the Parallel Element Processing Ensemble (PEPE), have been built around an *associative memory* (AM) instead of using the conventional *random-access memory* (RAM). The fundamental distinction between AM and RAM is that AM is content-addressable, allowing parallel access of multiple memory words, whereas the RAM must be sequentially accessed by specifying the word addresses. The inherent parallelism in associative memory has a great impact on the architecture of *associative processors*, a special class of SIMD array processors which update with the associative memories.

In this section, we describe the functional organization of an associative array processor and various parallel processing functions that can be performed on an associative processor. We classify associative processors based on associative-memory organizations. Finally, we identify the major searching applications of associative memories and associative processors. Associative processors have been built only as special-purpose computers for dedicated applications in the past.

5.4.1 Associative Memory Organizations

Data stored in an associative memory are addressed by their contents. In this sense, associative memories have been known as *content-addressable memory*, *parallel search memory*, and *multiaccess memory*. The major advantage of associative memory over RAM is its capability of performing parallel search and parallel comparison operations. These are frequently needed in many important applications, such as the storage and retrieval of rapidly changing databases, radar-signal tracking, image processing, computer vision, and artificial intelligence. The major shortcoming of associative memory is its much increased hardware cost. Presently, the cost of associative memory is much higher than that of RAMs.

The structure of a basic AM is modeled in Figure 5.32. The associative memory array consists of n words with m bits per word. Each bit cell in the $n \times m$ array consists of a flip-flop associated with some comparison logic gates for pattern match and read-write control. This logic-in-memory structure allows parallel read or parallel write in the memory array. A *bit slice* is a vertical column of bit cells of all the words at the same position. We denote the jth bit cell of the ith word as B_{ij} for $1 \leq i \leq n$ and $1 \leq j \leq m$. The ith word is denoted as:

$$W_i = (B_{i1} B_{i2} \cdots B_{im}) \qquad \text{for } i = 1, 2, \ldots, n$$

and the jth bit slice is denoted as:

$$B_j = (B_{1j} B_{2j} \cdots B_{nj}) \qquad \text{for } j = 1, 2, \ldots, m$$

Each bit cell B_{ij} can be written in, read out, or compared with an external interrogating signal. The parallel search operations involve both comparison and masking and are executed according to the organization of the associative memory. There are a number of registers and counters in the associative memory. The *comparand* register $C = (C_1, C_2, \ldots, C_m)$ is used to hold the key operand being searched for or being compared with. The *masking* register $M = (M_1, M_2, \ldots, M_m)$ is used to enable or disable the bit slices to be involved in the parallel comparison operations across all the words in the associative memory.

The *indicator* register $I = (I_1, I_2, \ldots, I_n)$ and one or more *temporary* registers $T = (T_1, T_2, \ldots, T_n)$ are used to hold the current and previous match patterns,

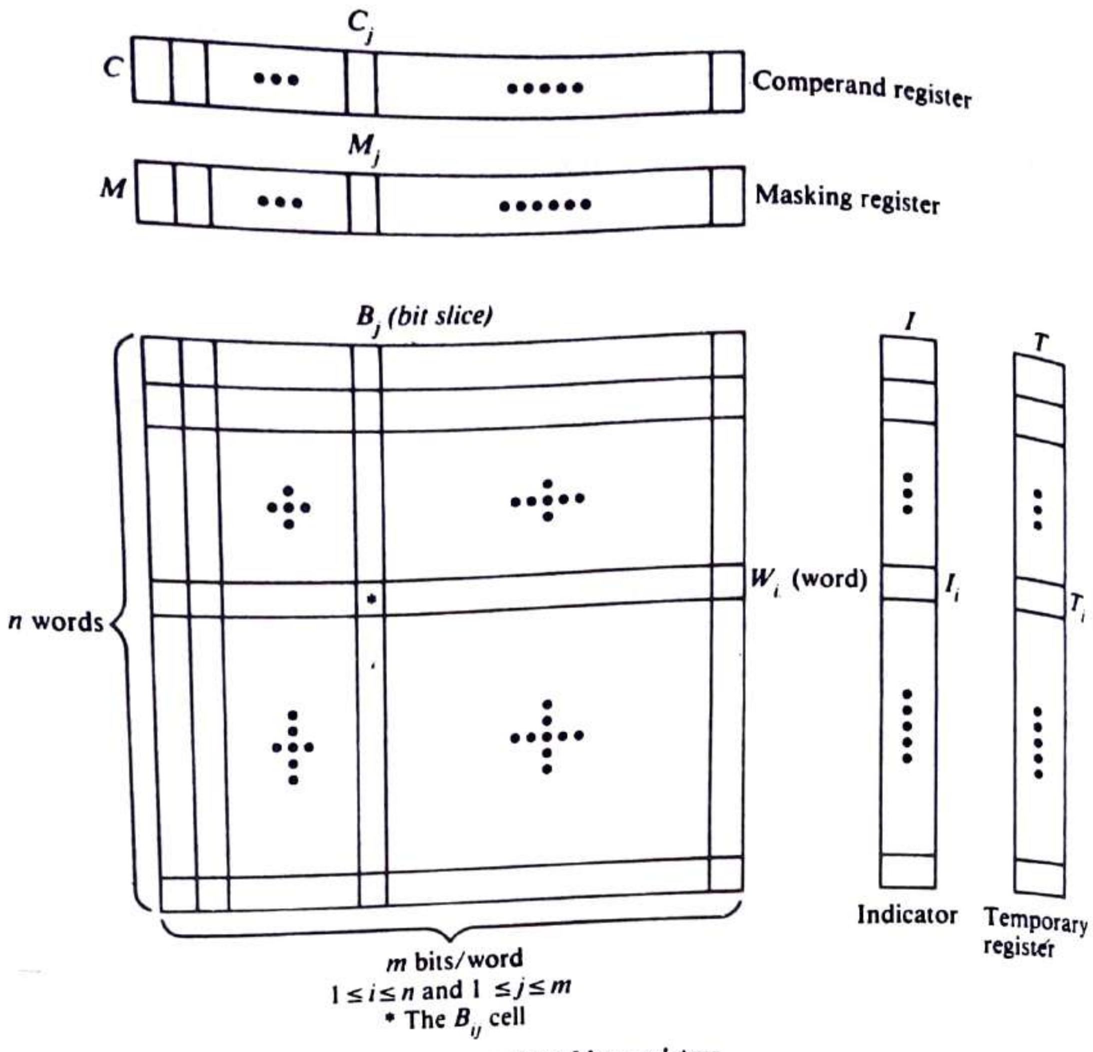


Figure 5.32 An associative memory array and working registers.

respectively. Each of these registers can be set, reset, or loaded from an external source with any desired binary patterns. The counters are used to keep track of the i and j index values. There are also some match detection circuits and priority logic, which are peripheral to the memory array and are used to perform some vector boolean operations among the bit slices and indicator patterns.

The search key in the C register is first masked by the bit pattern in the M register. This masking operation selects the effective fields of bit slices to be involved. Parallel comparisons of the masked key word with all words in the associative memory are performed by sending the proper interrogating signals to all the bit slices involved. All the involved bit slices are compared in parallel or in a sequential order, depending on the associative memory organization. It is possible that multiple words in the associative memory will match the search pattern. Therefore, the associative memory may be required to tag all the matched words. The indicator and temporary registers are mainly used for this purpose. The interrogation mechanism, read and write drives, and matching logic within a typical bit cell are depicted in Figure 5.33. The interrogating signals are associated with each bit slice, and the read-write drives are associated with each word. There are

Interrogation information	Mask*	Information stored	
		0	1
0	0	0	0
1	0	0	0
0	1	0	1
1	1	1	0

*Mask = 0 means that no comparison is performed at that bit position for all words.

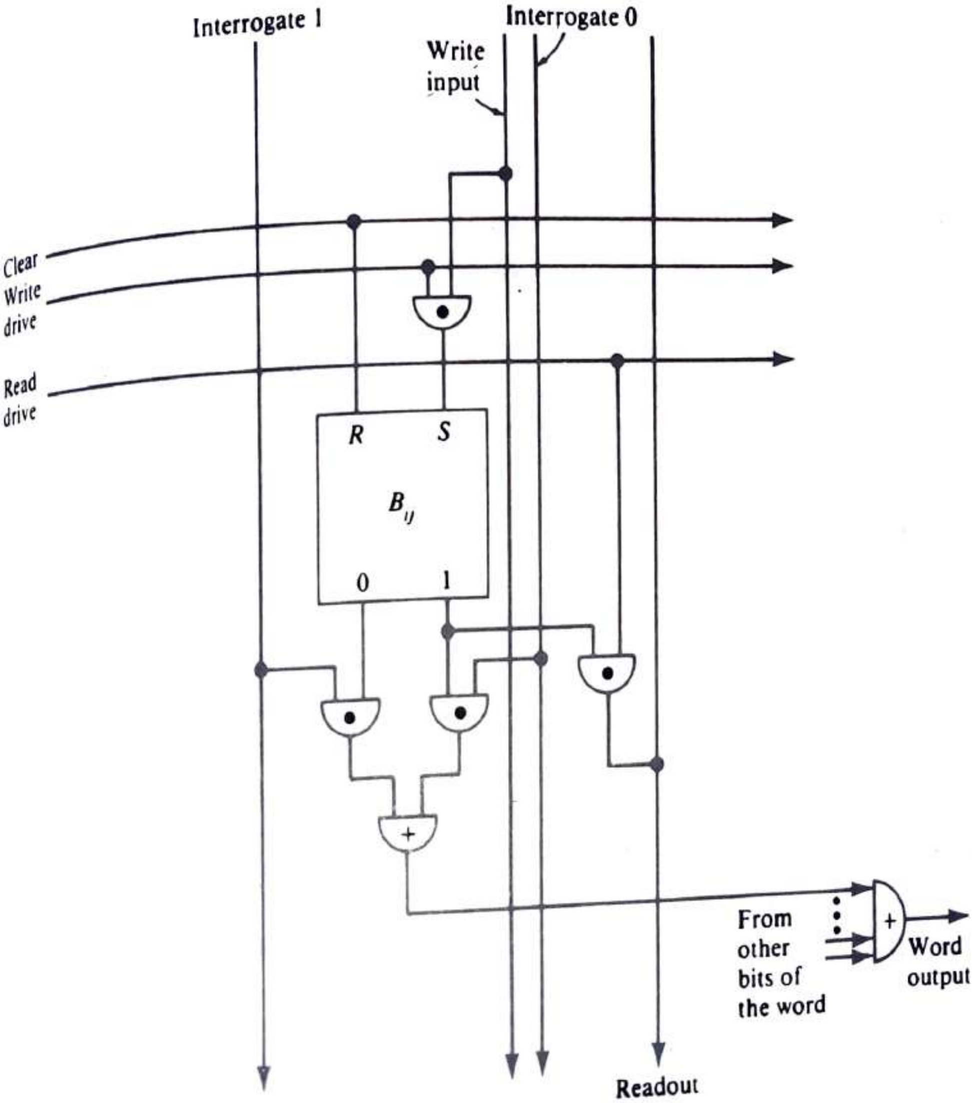


Figure 5.33 The schematic logic design of a typical cell in an associative memory.

two types of comparison readouts: the bit-cell readout and the word readout. The two types of readout are needed in two different associative memory organizations.

In practice, most associative memories have the capability of *word parallel* operations; that is, all words in the associative memory array are involved in the parallel search operations. This differs drastically from the *word serial* operations encountered in RAMs. Based on how bit slices are involved in the operation, we consider below two different associative memory organizations:

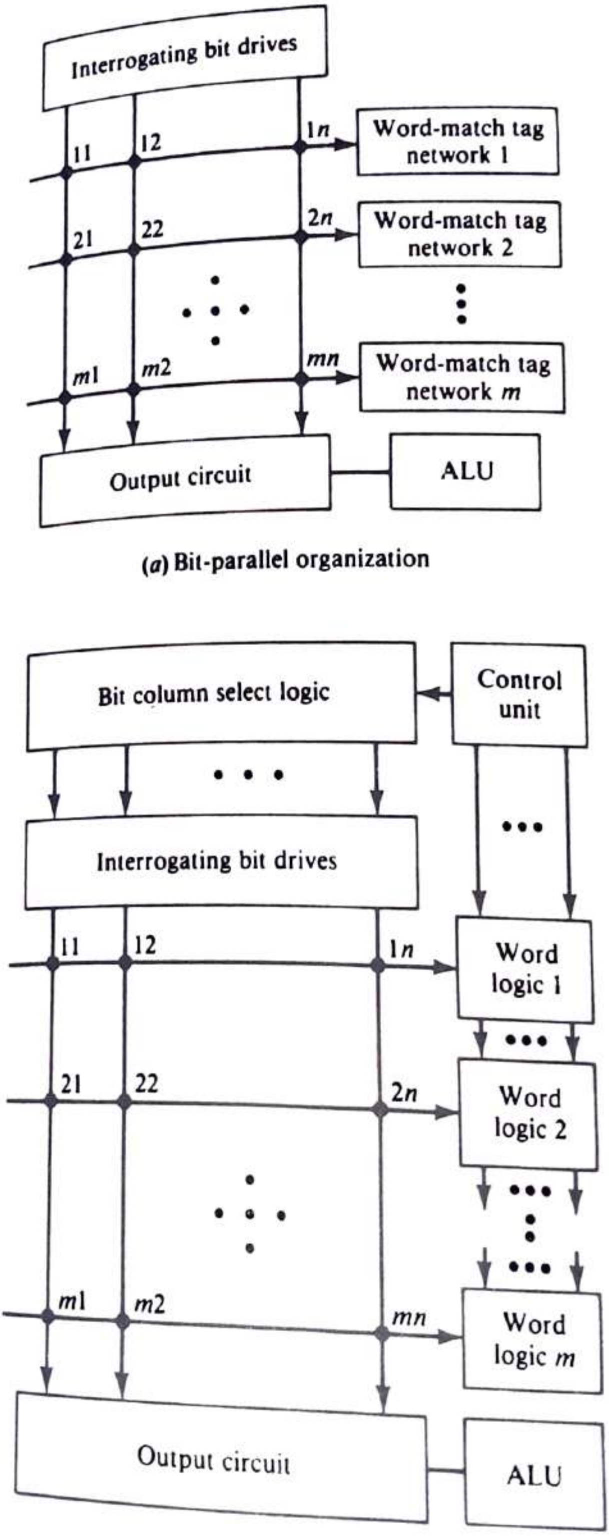
The bit parallel organization In a bit parallel organization, the comparison process is performed in a parallel-by-word and parallel-by-bit fashion. All bit slices which are not masked off by the masking pattern are involved in the comparison process. In this organization, word-match tags for all words are used (Figure 5.34*a*). Each cross point in the array is a bit cell. Essentially, the entire array of cells is involved in a search operation.

Bit serial organization The memory organization in Figure 5.34*b* operates with one bit slice at a time across all the words. The particular bit slice is selected by an extra logic and control unit. The bit-cell readouts will be used in subsequent bit-slice operations. The associative processor STARAN has the bit serial memory organization and the PEPE has been installed with the bit parallel organization.

The associative memories are used mainly for search and retrieval of non-numeric information. The bit serial organization requires less hardware but is slower in speed. The bit parallel organization requires additional word-match detection logic but is faster in speed. We present below an example to illustrate the search operation in a bit parallel associative memory. Bit serial associative memory will be presented in Section 5.4.3 with various associative search and retrieval algorithms.

Example 5.8 Consider a student-file search in a bit parallel associative memory, as illustrated in Figure 5.35. The query needs to search all students whose age is not younger than 21 but is younger than 31. This requires performing the *not-less-than* search and the *less-than* search on the age field of the file. Two matching patterns are used in the two subsequent searches. The masking pattern selects the age field. The lower-bound 21 is loaded into the *C* register as the first key word. Parallel comparisons are performed on all student records (words) in the file. Initially, the indicator register is cleared to be zero.

After the first search, those students who are not younger than 21 are marked with a *1* in the indicator register, one bit per each student word. This matching vector is then transferred to one of the *T* registers. Then the upper-bound 31 is loaded into *C* as the second matching key. After the second search, a new matching vector is sent to the *I* register. A bitwise ANDing operation is then performed between the *I* and *T* registers with the resulting vector residing in the *I* register as the final output of the search process. The whole search process requires only two accesses of the associative memory. An output circuit (shown in Figure 5.34) is used to control the reading out of the result.

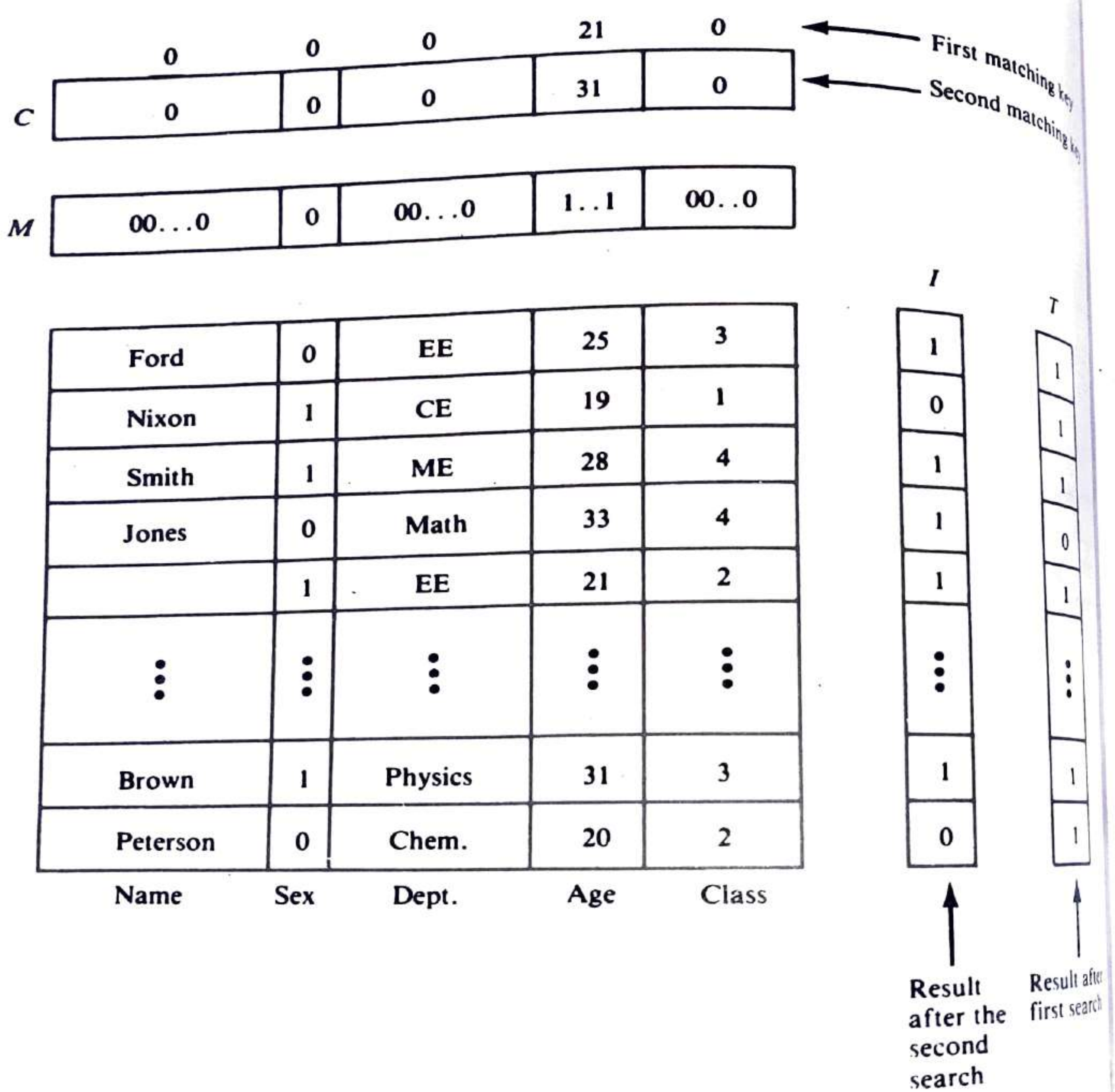


(*a*) Bit-parallel organization

(*b*) Bit-serial organization

Figure 5.34 Associative memory organ tions.

Query: Search for those students whose ages are in the range (21, 31)



	Name	Sex	Dept.	Age	Class	
	0	0	0	21	0	First matching key
C	0	0	0	31	0	Second matching key
M	00. . .0	0	00. . .0	1. .1	00. .0	

Name	Sex	Dept.	Age	Class	I	T
Ford	0	EE	25	3	1	1
Nixon	1	CE	19	1	0	1
Smith	1	ME	28	4	1	1
Jones	0	Math	33	4	1	0
	1	EE	21	2	1	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
Brown	1	Physics	31	3	1	1
Peterson	0	Chem.	20	2	0	1

Figure 5.35 An associative memory used for the storage and retrieval of a student file.

7

Multiprocessors and Multicomputers

In this chapter, we study system architectures of multiprocessors and multicomputers. Various cache coherence protocols, synchronization methods, crossbar switches, multiport memory, and multistage networks are described for building multiprocessor systems. Then we discuss multicomputers with distributed memories which are not globally shared.The Intel Paragon is used as a case study. Message-passing mechanisms required with multicomputers are also reviewed.Single-address-space multicomputers will be studied in Chapter 9.

7.1 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.

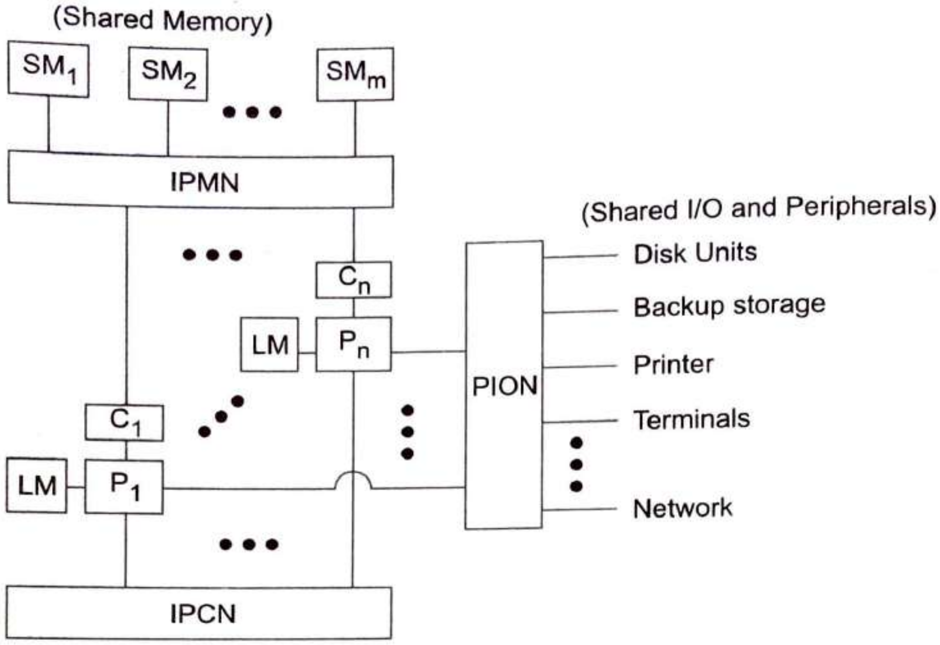
A generalized multiprocessor system is depicted in Fig. 7.1. This architecture combines features from the UMA, NUMA, and COMA models introduced in Section 1.4.1. Each processor P_i is attached to its own local memory and private cache. Multiple processors are connected to shared-memory modules through an interprocessor-memory network (IPMN).

The processors share the access of I/O and peripheral devices through a processor I/O network (PION). Both IPMN and PION are necessary in a shared-resource multiprocessor. Direct interprocessor communications are supported by an optional interprocessor communication network (IPCN) instead of through the shared memory.

Network Characteristics Each of the above types of networks can be designed with many choices. The choices are based on the topology, timing protocol, switching method, and control strategy. Dynamic networks are used in multiprocessors in which the interconnections are under program control. Timing, switching, and control are three major operational characteristics of an interconnection network. The timing control can be either *synchronous* or *asynchronous*. Synchronous networks are controlled by a global clock that synchronizes all network activities. Asynchronous networks use handshaking or interlocking mechanisms to coordinate fast and slow devices requesting use of the same network.

A network can transfer data using either *circuit switching* or *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.



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)

Fig. 7.1 **Interconnection structures in a generalized multiprocessor system with local memory, private caches, shared memory, and shared peripherals**

Network control strategy is classified as *centralized* or *distributed*. With centralized control, a global controller receives requests from all devices attached to the network and grants the network access to one or more requesters. In a distributed system, requests are handled by local devices independently.

7.1.1 Hierarchical Bus Systems

A *bus system* consists of a hierarchy of buses connecting various system and subsystem components in a computer. Each bus is formed with a number of signal, control, and power lines. Different buses are used to perform different interconnection functions.

In general, the hierarchy of bus systems are packaged at different levels as depicted in Fig. 7.2, including local buses on boards, backplane buses, and I/O buses.

Local Bus Buses implemented within processor chips or on *printed-circuit* boards are called *local buses.* On a processor board one may find a local bus which provides a common communication path among major components (chips) mounted on the board. A memory board uses a *memory bus* to connect the memory with the interface logic. An I/O or network interface chip or board uses a *data bus*. Each of these local buses consists of signal and utility lines.

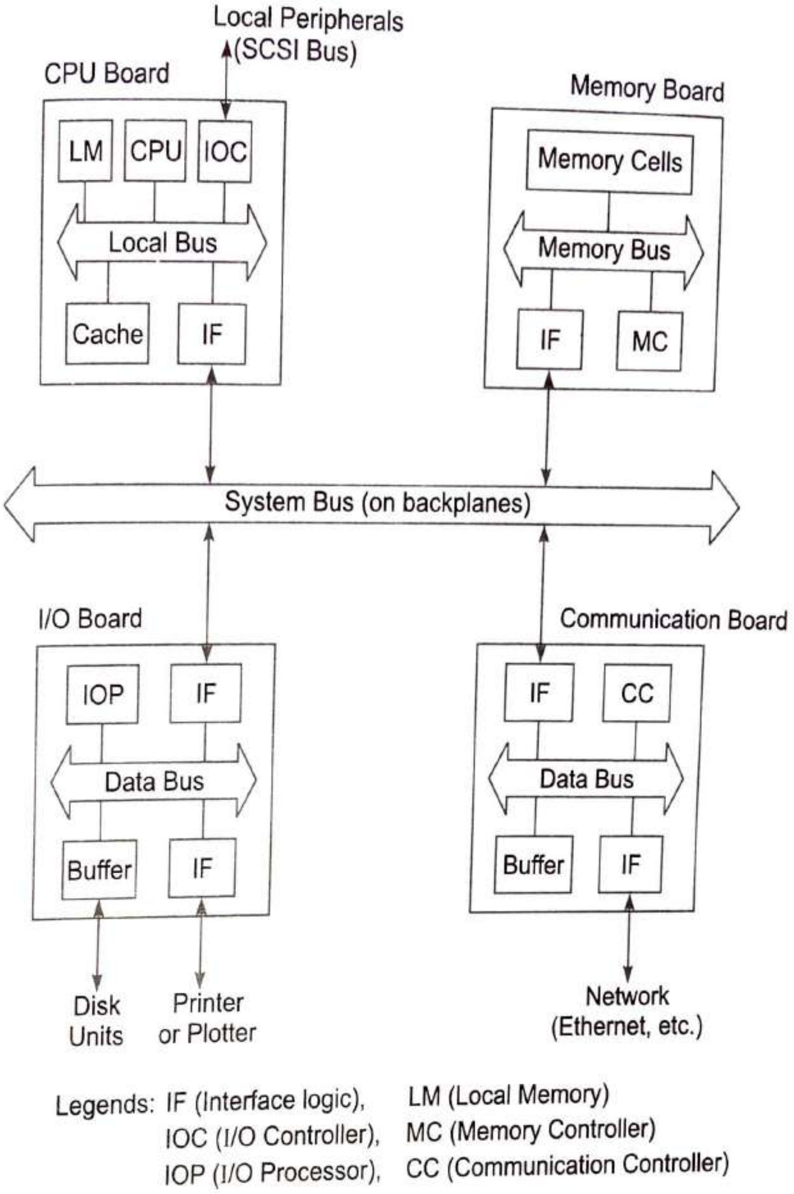


Fig. 7.2 Bus systems at board level, backplane level, and I/O level

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

Several backplane bus standards have been developed over time such as the VME bus (IEEE Standard 1014-1987), Multibus II (IEEE Standard 1296-1987), and Futurebus+ (IEEE Standard 896.1-1991) as introduced in Chapter 5. However, point to-point switched interconnects have emerged as more efficient alternatives, as discussed in Chapters 5 and 13.

I/O Bus Input/output devices are connected to a computer system through an *I/O bus* such as the SCSI (Small Computer Systems Interface) bus. This bus is made of coaxial cables with taps connecting disks, printer, and other devices to a processor through an I/O controller (Fig. 7.2). Special interface logic is used to connect various board types to the backplane bus.

Complete specifications for a bus system include logical, electrical, and mechanical properties, various application profiles, and interface requirements. Our study will be confined to the logical and application aspects of system buses. Emphasis will be placed on the scalability and bus support for cache coherence and fast synchronization.

For example, the core of the Encore Multimax multiprocessor was the Nanobus, consisting of 20 slots, a 32-bit address, a 64-bit data path, and a 14-bit vector bus, and operating at a clock rate of 12.5 MHz with a total memory bandwidth of 100 Mbytes/s. The Sequent multiprocessor bus had a 64-bit data path, a 10-MHz clock rate, and a 32-bit address, for a channel bandwidth of 80 Mbytes/s. A write-back private cache was used to reduce the bus traffic by 50%.

Digital bus interconnects can be adopted in commercial systems ranging from workstations to minicomputers, mainframes, and multiprocessors. Hierarchical bus systems can be used to build medium-sized multiprocessors with less than 100 processors. However, the bus approach is limited by bandwidth scalability and the packaging technology employed.

Hierarchical Buses and Caches Wilson (1987) proposed a hierarchical cache/ bus architecture as shown in Fig. 7.3. This is a multilevel tree structure in which the leaf nodes are processors and their private caches (denoted P_j and C_{1j} in Fig. 7.3). These are divided into several clusters, each of which is connected through a cluster bus.

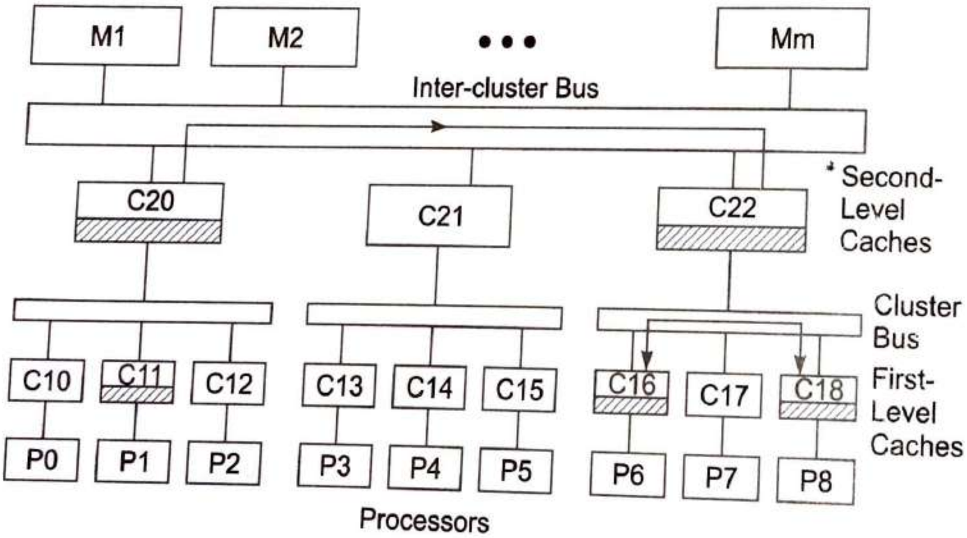


Fig. 7.3 A hierarchical cache/bus architecture for designing a scalable multiprocessor (Courtesy of Wilson; reprinted from *Proc. of Annual Int. Symp. on Computer Architecture, 1987*)

An intercluster bus is used to provide communications among the clusters. Second level caches (denoted as C_{2i}) are used between each cluster bus and the intercluster bus. Each second-level cache must have a capacity that is at least an order of magnitude larger than the sum of the capacities of all first-level caches connected beneath it.

Each single cluster operates as a single-bus system. Snoopy bus coherence protocols can be used to establish consistency among first-level caches belonging to the same cluster. Second-level caches are used to extend consistency from each local cluster to the upper level.

The upper-level caches form another level of shared memory between each cluster and the main memory modules connected to the intercluster bus. Most memory requests should be satisfied at the lower-level caches. Intercluster cache coherence is controlled among the second-level caches and the resulting effects are passed to the lower level.

Example 7.1 Encore Ultramax multiprocessor architecture

The Ultramax had a two-level hierarchical-bus architecture as depicted in Fig. 7.4. The Ultramax architecture was very similar to that characterized by Wilson, except that the global Nanobus was used only for intercluster communications.

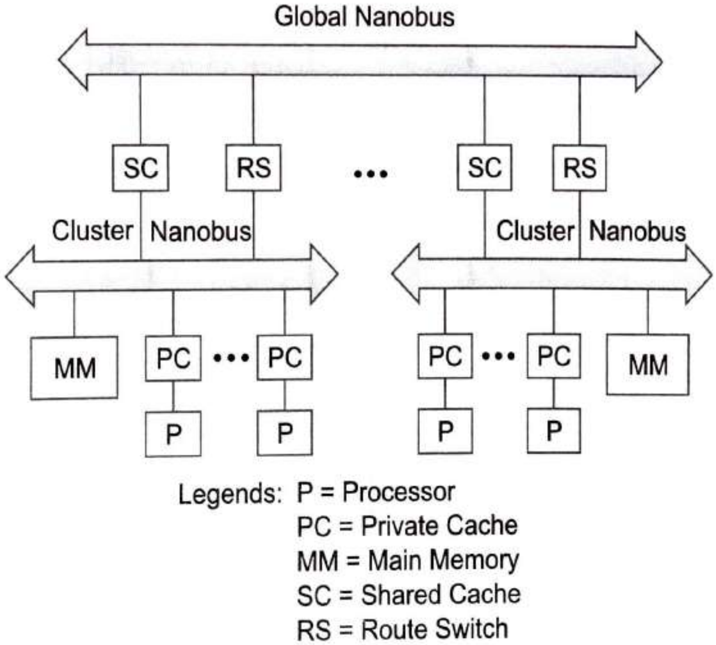


Fig. 7.4 The Ultramax multiprocessor architecture using hierarchical buses with multiple clusters (Courtesy of Encore Computer Corporation, 1987)

The shared memories were distributed to all clusters instead of being connected to the intercluster bus. The cluster caches formed the second-level caches and performed the same filtering and cache coherence control for remote accesses as in Wilson's scheme. When an access request reached the top bus, it would be routed down to the cluster memory that matched it with the reference address.

The idea of using *bridges* between multiprocessor clusters is to allow transactions initiated on a local bus to be completed on a remote bus. As exemplified in Fig. 7.5, multiple buses are used to build a very large system consisting of three multiprocessor clusters. The bus used in this example is Futurebus+, but the basic idea is more general. Bridges are used to interface the clusters. The main functions of a bridge include communication protocol conversion, interrupt handling in split transactions, and serving as cache and memory agents.

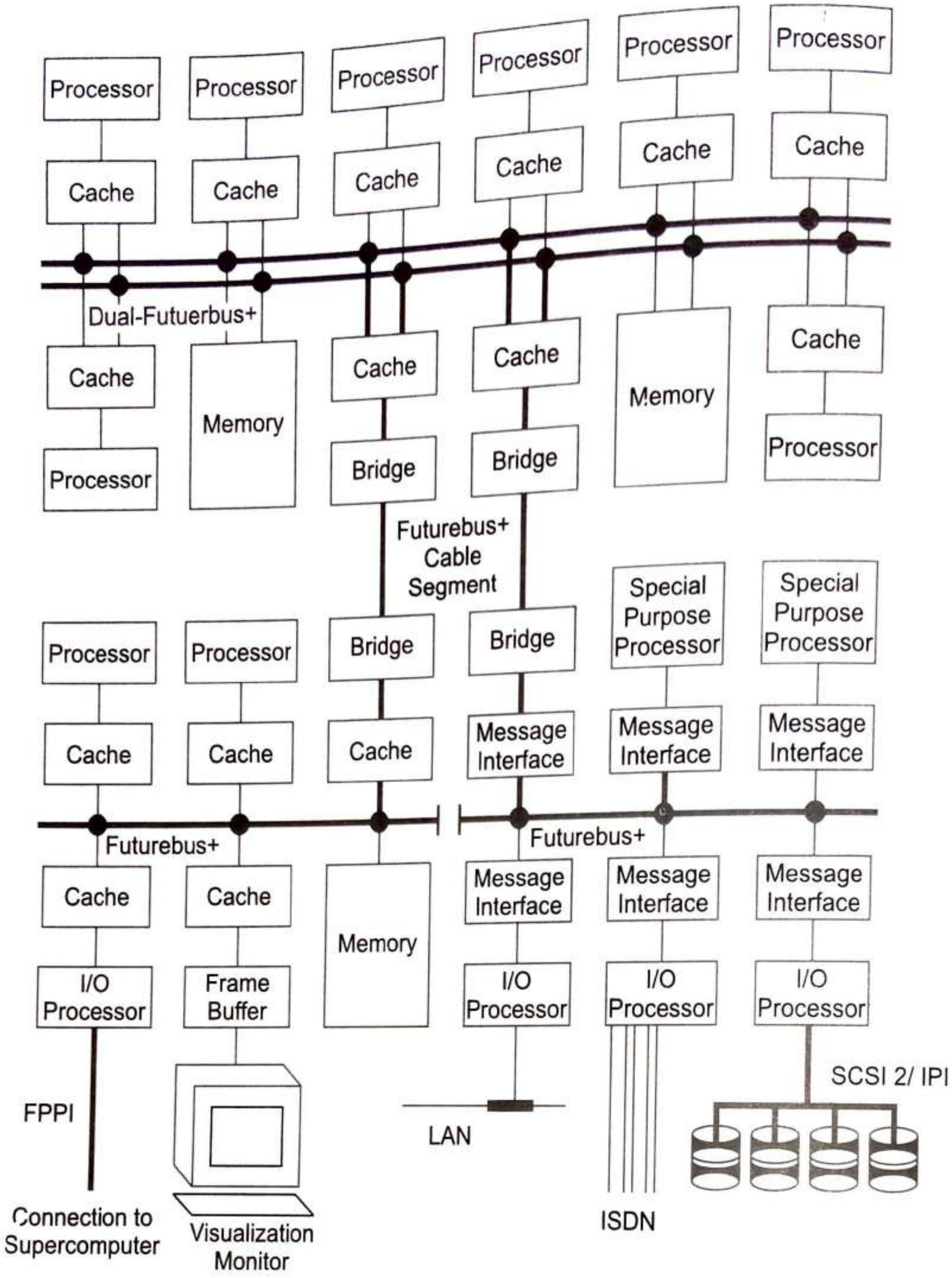


Fig. 7.5 A multiprocessor system using multiple Futurebus+ segments (Reprinted with permission from IEEE Standard 896.1-1991, copyright © 1991 by IEEE, Inc.)

7.1.2 Crossbar Switch and Multiport Memory

Switched networks provide dynamic interconnections between the inputs and outputs. Major classes of switched networks are specified below, based on the number of stages and blocking or nonblocking. We describe the crossbar networks and multiport memory structures first and then the multistage networks. Crossbar networks are mostly used in small or medium-size systems. The multistage networks can be extended to larger systems if the increased latency problem can be suitably addressed.

Network Stages Depending on the interstage connections used, a *single-stage network* is also called a *recirculating network* because data items may have to recirculate through the single stage many times before reaching their destination. A single-stage network is cheaper to build, but multiple passes may be needed to establish certain connections. The crossbar switch and multiport memory organization are both single-stage networks.

A multistage network consists of more than one stage of switch boxes. Such a network should be able to connect from any input to any output. We will study unidirectional multistage networks in Section 7.1.3. The choice of interstage connection patterns determines the network connectivity. These patterns may be the same or different at different stages, depending the class of networks to be designed. The Omega network, Flip network, and Baseline networks are all multistage networks.

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

Examples of blocking networks include the Omega (Lawrie, 1975), Baseline (Wu and Feng, 1980), Banyan (Goke and Lipovski, 1973), and Delta networks (Patel, 1979). Some blocking networks are equivalent after graph transformations. In fact, most multistage networks are blocking in nature. In a blocking network, multiple passes through the network may be needed to achieve certain input-output connections.

A multistage network is called *nonblocking* if it can perform all possible connections between inputs and outputs by rearranging its connections. In such a network, a connection path can always be established between any input-output pair. The Benes networks (Benes, 1965) have such a capability. However, Benes networks require almost twice the number of stages to achieve the nonblocking connections. The Clos networks (Clos, 1953) can also perform all permutations in a single pass without blocking. Certain subclasses of blocking networks can also be made nonblocking if extra stages are added or connections are restricted. The blocking problem can be avoided by using combining networks to be described in the next section.

Crossbar Networks In a *crossbar network*, every input port is connected to a free output port through a crosspoint switch (circles in Fig. 2.26a) without blocking. A crossbar network is a single-stage network built with unary switches at the crosspoints.

Once the data is read from the memory, its value is returned to the requesting processor along the same crosspoint switch. In general, such a crossbar network requires the use of $n \times m$ crosspoint switches. A square crossbar ($n = m$) can implement any of the $n!$ permutations without blocking.

As introduced earlier, a crossbar switch network is a single-stage, nonblocking, permutation network. Each crosspoint in a crossbar network is a unary switch which can be set open or closed, providing a point-to-point connection path between the source and destination.

All processors can send memory requests independently and asynchronously. This poses the problem of multiple requests destined for the same memory module at the same time. In such cases, only one of the requests is serviced at a time. Let us characterize below the crosspoint switching operations.

Crosspoint Switch Design Out of n crosspoint switches in each column of an $n \times m$ crossbar mesh, only one can be connected at a time. To resolve the contention for each memory module, each crosspoint switch must be designed with extra hardware.

Furthermore, each crosspoint switch requires the use of a large number of connecting lines accommodating address, data path, and control signals. This means that each crosspoint has a complexity matching that of a bus of the same width.

For an $n \times n$ crossbar network, this implies that n^2 sets of crosspoint switches and a large number of lines are needed. What this amounts to is a crossbar network requiring extensive hardware when n is very large. So far only relatively small crossbar networks with $n \leq 16$ have been built into commercial machines.

On each row of the crossbar mesh, multiple crosspoint switches can be connected simultaneously. Simultaneous data transfers can take place in a crossbar between n pairs of processors and memories.

Figure 7.6 shows the schematic design of a row of crosspoint switches in a single crossbar network. Multiplexer modules are used to select one of n *read* or *write* requests for service. Each processor sends in an independent request, and the arbitration logic makes the selection based on certain fairness or priority rules.

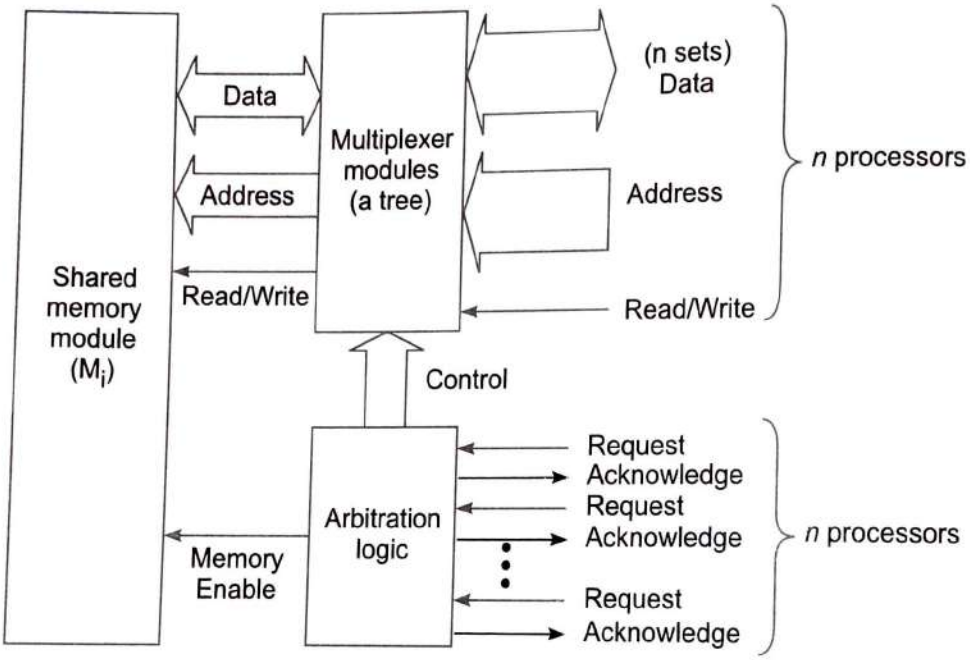


Fig. 7.6 Schematic design of a row of crosspoint switches in a crossbar network

For example, a 4-bit control signal will be generated for $n = 16$ processors. Note that n sets of data, address, and read/write lines are connected to the input of the multiplexer tree. Based on the control signal received, only one out of n sets of information lines is selected as the output of the multiplexer tree.

The memory address is entered for both *read* and *write* access. In the case of *read,* the data fetched from memory are returned to the selected processor in the reverse direction using the data path established. In the case of *write*, the data on the data path are stored in memory.

Acknowledge signals are used to indicate the arbitration result to all requesting processors. These signals initiate data transfer and are used to avoid conflicts. Note that the data path established is bidirectional, in order to serve both *read* and *write* requests for different memory cycles.

Crossbar Limitations A single processor can send many requests to multiple memory modules. For an $n \times n$ crossbar network, at most n memory words can be delivered to at most n processors in each cycle.

The crossbar network offers the highest bandwidth of n data transfers per cycle, as compared with only one data transfer per bus cycle. Since all necessary switching and conflict resolution logic are built into the crosspoint switch, the processor interface and memory port logic are much simplified and cheaper. A crossbar network is cost-effective only for small multiprocessors with a few processors accessing a few memory modules. A single-stage crossbar network is not expandable once it is built.

Redundancy or parity-check lines can be built into each crosspoint switch to enhance the fault tolerance and reliability of the crossbar network.

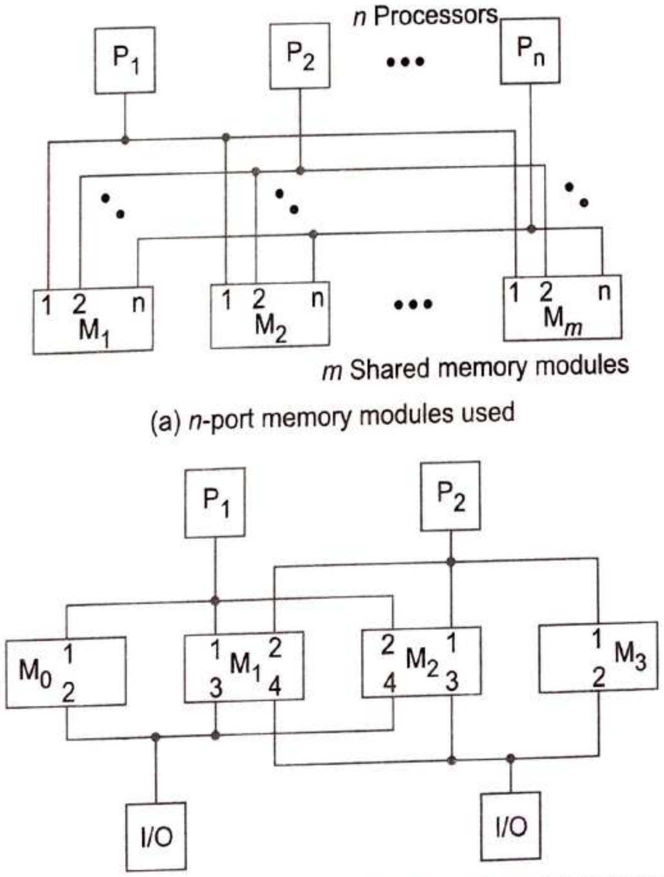
Multiport Memory Because building a crossbar network into a large system is cost prohibitive, some mainframe multiprocessors used a multiport memory organization. The idea is to move all crosspoint arbitration and switching functions associated with each memory module into the memory controller.

Thus the memory module becomes more expensive due to the added access ports and associated logic as demonstrated in Fig. 7.7a. The circles in the diagram represent *n* switches tied to *n* input ports of a memory module. Only one of *n* processor requests can be honored at a time.

The multiport memory organization is a compromise solution between a low-cost, low-performance bus system and a high-cost, high-bandwidth crossbar system. The contention bus is time-shared by all processors and device modules attached. The multiport memory must resolve conflicts among processors.

This memory structure becomes expensive when *m* and *n* become large. A typical mainframe multiprocessor configuration may have $n = 4$ processors and $m = 16$ memory modules. A multiport memory multiprocessor is not scalable because once the ports are fixed, no more processors can be added without redesigning the memory controller.

Another drawback is the need for a large number of interconnection cables and connectors when the configuration becomes large. The ports of each memory module in Fig. 7.7b are prioritized. Some of the processors are CPUs, some are I/O processors, and some are connected to dedicated processors.



(a) *n*-port memory modules used

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

Fig. 7.7 Multiport memory organizations for multiprocessor systems (Courtesy of P. H. Enslow, *ACM Computing Surveys*, March 1977)

For example, the Univac 1100/94 multiprocessor consisted of four CPUs, four I/O processors, and two scientific vector processors connected to four shared-memory modules, each of which was 10-way ported. The access to these ports was prioritized under operating system control. In other multiprocessors, part of the memory module can be made private with ports accessible only to the owner processors.

7.1.3 Multistage and Combining Networks

Multistage networks are used to build larger multiprocessor systems. We describe two multistage networks, the Omega network and the Butterfly network, that have been built into commercial machines. We will study a special class of multistage networks, called combining networks, for resolving access conflicts automatically through the network. The combining network was built into the NYU's Ultracomputer.

Routing in Omega Network We have defined the Omega network in Chapter 2. In what follows, we describe the message-routing algorithm and broadcast capability of Omega network. This class of network was built into the Illinois Cedar multiprocessor (Kuck et al., 1987), into the IBM RP3 (Pfister et al., 1985), and into the NYU Ultracomputer (Gottlieb et al., 1983). An 8-input Omega network is shown in Fig. 7.8.

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. Data routing is controlled by inspecting the destination code in binary. When the ith high-order bit of the destination code is a 0, a 2 × 2 switch at stage i connects the input to the upper output. Otherwise, the input is directed to the lower output.

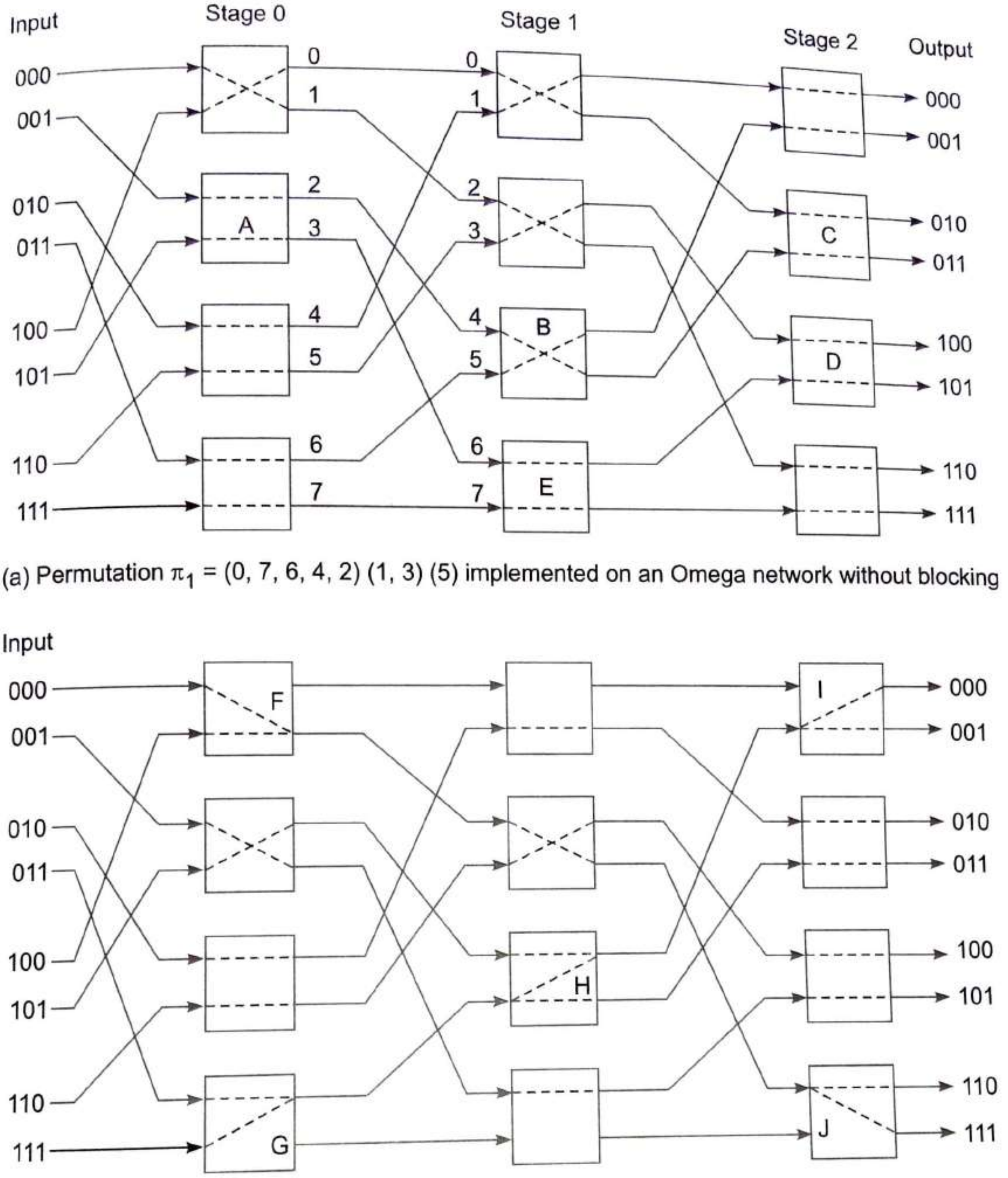
Two switch settings are shown in Figs. 7.8a and b with respect to permutations $\pi_1 = (0, 7, 6, 4, 2)(1, 3)(5)$ and $\pi_2 = (0, 6, 4, 7, 3)(1, 5)(2)$, respectively.

The switch settings in Fig. 7.8a are for the implementation of π_1, which maps $0 \rightarrow 7$, $7 \rightarrow 6$, $6 \rightarrow 4$, $4 \rightarrow 2$, $2 \rightarrow 0$, $1 \rightarrow 3$, $3 \rightarrow 1$, $5 \rightarrow 5$. Consider the routing of a message from input 001 to output 011. This involves the use of switches A, B, and C. Since the most significant bit of the destination 011 is a "zero", switch A must be set straight so that the input 001 is connected to the upper output (labeled 2). The middle bit in 011 is a "one", thus input 4 to switch B is connected to the lower output with a "crossover" connection. The least significant bit in 011 is a "one", implying a flat connection in switch C. Similarly, the switches A, E, and D are set for routing a message from input 101 to output 101. There exists no conflict in all the switch settings needed to implement the permutation π_1 in Fig. 7.8a.

Now consider implementing the permutation π_2 in the 8-input Omega network (Fig. 7.8b). Conflicts in switch settings do exist in three switches identified as F, G, and H. The conflicts occurring at F are caused by the desired routings 000 → 110 and 100 → 111. 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 blocked.

Similarly, we see conflicts at switch G between 011 → 000 and 111 → 011, and at switch H between 101 → 001 and 011 → 000. At switches I and J, broadcast is used from one input to two outputs, which is allowed if the hardware is built to have four legitimate states as shown in Fig. 2.24a. The above example indicates the fact that not all permutations can be implemented in one pass through the Omega network.

The Omega network is a blocking network. In case of blocking, one can establish the conflicting connections in several passes. For the example 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. 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.



(a) Permutation $\pi_1 = (0, 7, 6, 4, 2)\ (1, 3)\ (5)$ implemented on an Omega network without blocking

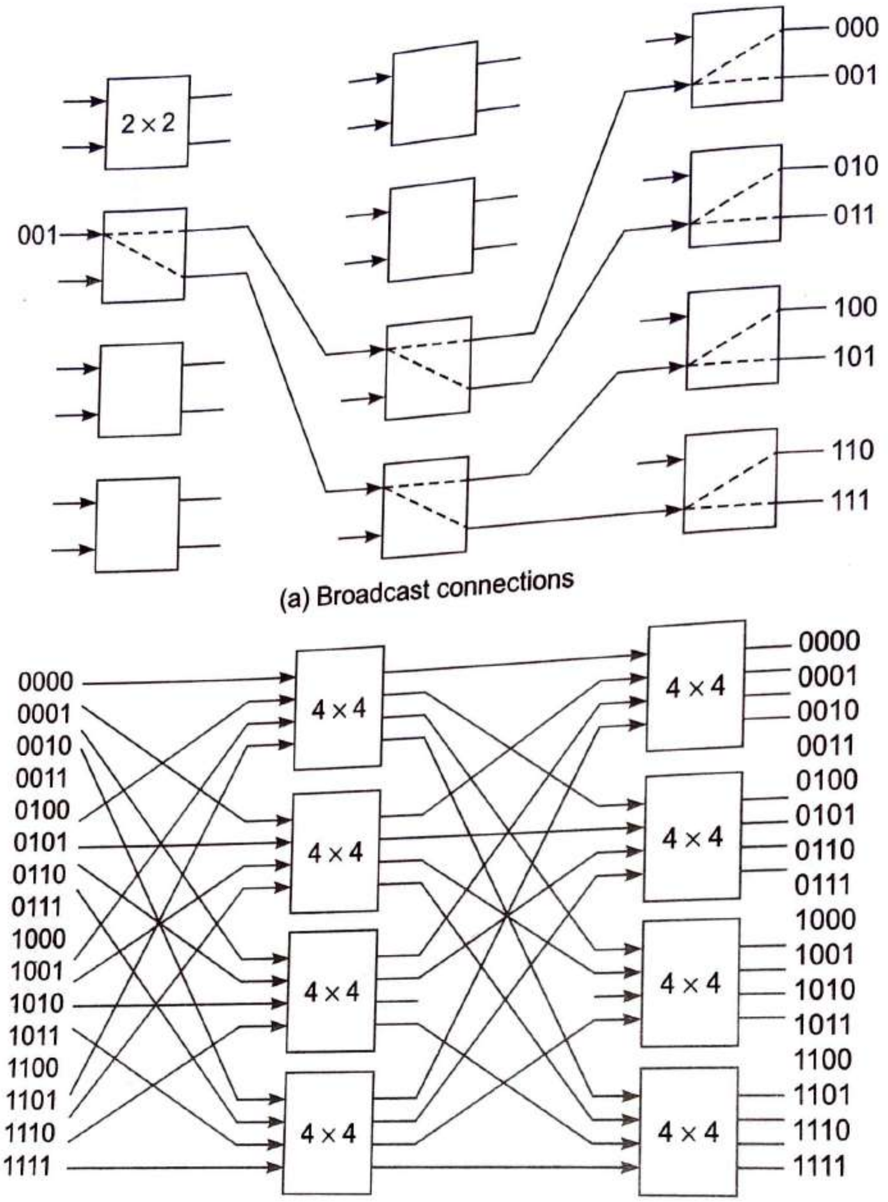
(b) Permutation $\pi_2 = (0, 6, 4, 7, 3)\ (1, 5)\ (2)$ blocked at switches marked F, G, and H

Fig. 7.8 Two switch settings of an 8 × 8 Omega network built with 2 × 2 switches

For $n = 8$, this implies that only $8^4/8! = 4096/40320 = 0.1016 = 10.16\%$ of all permutations are implementable in a single pass through an 8-input Omega network. All others will cause blocking and demand up to three passes to be realized. In general, a maximum of $\log_2 n$ passes are needed for an n-input Omega. Blocking is not a desired feature in any multistage network, since it lowers the effective bandwidth.

The Omega network can also be used to broadcast data from one source to many destinations, as exemplified in Fig. 7.9a, using the upper broadcast or lower broadcast switch settings. In Fig. 7.9a, the message at input 001 is being broadcast to all eight outputs through a binary tree connection.

The two-way shuffle interstage connections can be replaced by four-way shuffle interstage connections when 4 × 4 switch boxes are used as building blocks, as exemplified in Fig. 7.9b for a 16-input Omega network with $\log_4 16 = 2$ stages.



(a) Broadcast connections

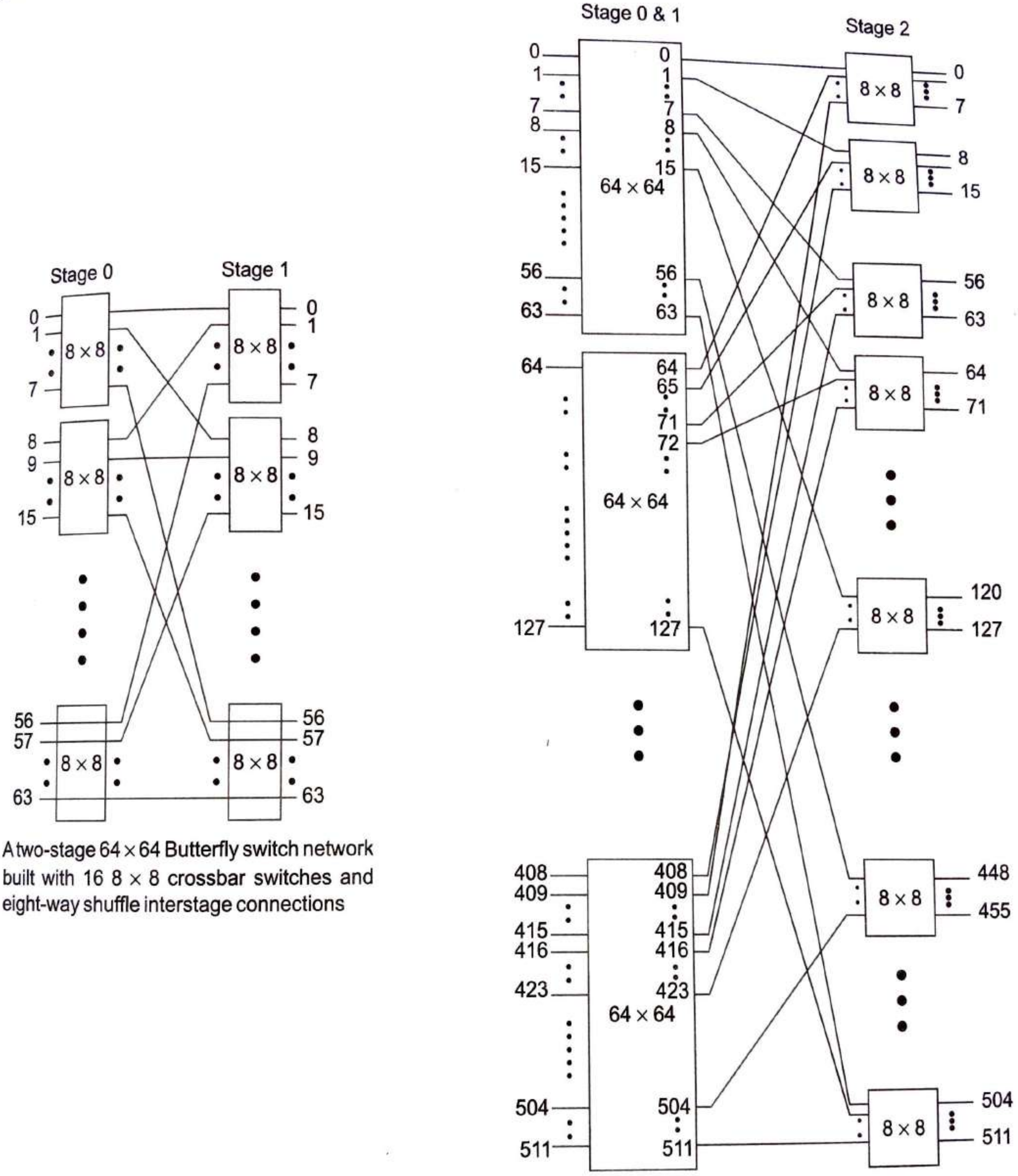
(b) Using four-way shuffle interstage connections

Fig. 7.9 Broadcast capability of an Omega network built with 4 × 4 switches

Note that a four-way shuffle corresponds to dividing the 16 inputs into four equal subsets and then shuffling them evenly among the four subsets. When $k \times k$ switch boxes are used, one can define a k-way shuffle function to build an even larger Omega network with $\log_k n$ stages.

Routing in Butterfly Networks This class of networks is constructed with crossbar switches as building blocks. Figure 7.10 shows two Butterfly networks of different sizes. Figure 7.10a shows a 64-input Butterfly network built with two stages ($2 = \log_8 64$) of 8×8 crossbar switches. The eight-way shuffle function is used to establish the interstage connections between stage 0 and stage 1. In Fig. 7.10b, a three-stage Butterfly network is constructed for 512 inputs, again with 8×8 crossbar switches. Each of the 64×64 boxes in Fig. 7.10b is identical to the two-stage Butterfly network in Fig. 7.10a.

In total, sixteen 8×8 crossbar switches are used in Fig. 7.10a and $16 \times 8 + 8 \times 8 = 192$ are used in Fig. 7.10b. Larger Butterfly networks can be modularly constructed using more stages. Note that no broadcast connections are allowed in a Butterfly network, making these networks a restricted subclass of Omega networks.



(a) A two-stage 64 × 64 Butterfly switch network built with 16 8 × 8 crossbar switches and eight-way shuffle interstage connections

(b) A three-stage 512 × 512 Butterfly switch network built with 192 8 × 8 crossbar switches

Fig. 7.10 Modular construction of Butterfly switch networks with 8 × 8 crossbar switches (Courtesy of BBN Advanced Computers, Inc., 1990)

The Hot-Spot Problem When the network traffic is nonuniform, a *hot spot* may appear corresponding to a certain memory module being excessively accessed by many processors at the same time. For example, a semaphore variable being used as a synchronization barrier may become a hot spot since it is shared by many processors.

Hot spots may degrade the network performance significantly. In the NYU Ultracomputer and the IBM RP3 multiprocessor, a combining mechanism has been added to the Omega network. The purpose 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.

Fetch&Add This atomic memory operation is effective in implementing an N-way synchronization with a complexity independent of N. In a Fetch&Add(x, e) operation, x is an integer variable in shared memory and e is an integer increment. When a single processor executes this operation, the semantics is

$$\begin{array}{l} \text{Fetch\&Add}\ (x, e) \\ \qquad \begin{array}{rcl} \{temp & \leftarrow & x; \\ x & \leftarrow & temp + e; \end{array} \\ \qquad \text{return}\ \ temp\} \end{array} \tag{7.1}$$

When N processes attempt Fetch&Add(x, e) at the same memory word simultaneously, the memory is updated only once following a *serialization principle*. The sum of the N increments, $e_1 + e_2 + \ldots + e_N$, is produced in any arbitrary serialization of the N requests.

This sum is added to the memory word x, resulting in a new value $x + e_1 + e_2 + \ldots + e_N$. The values returned to the N requests are all unique, depending on the serialization order followed. The net result is similar to a sequential execution of N Fetch&Adds but is performed in one indivisible operation. Two simultaneous requests are combined in a switch as illustrated in Fig. 7.11.

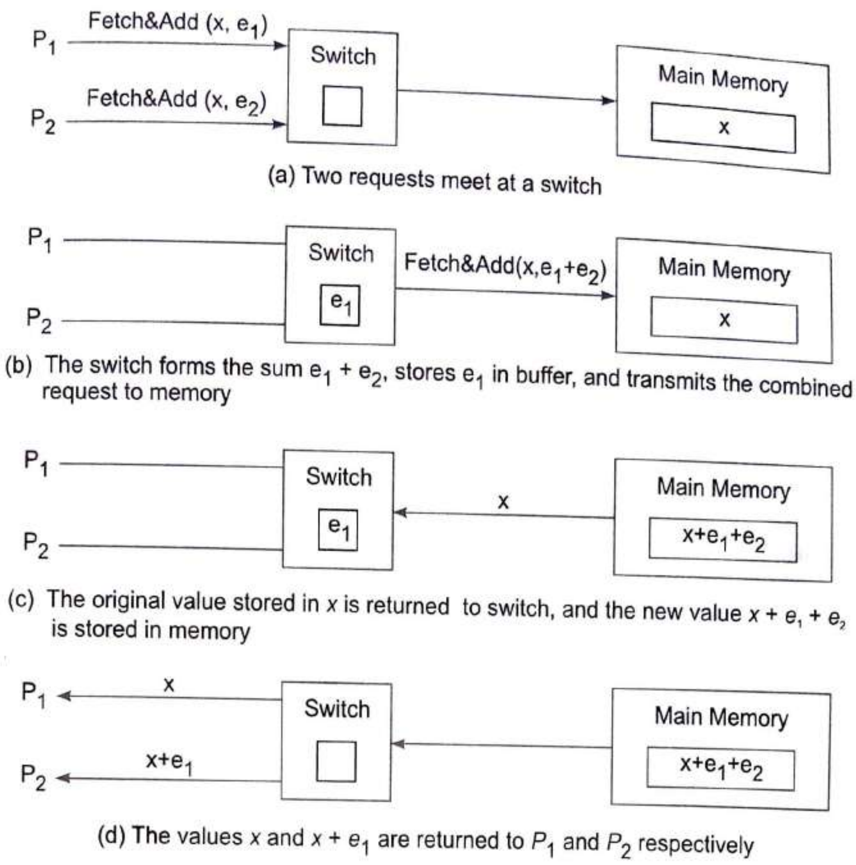
One of the following operations will be performed if processor P_1 executes $\text{Ans}_1 \leftarrow$ Fetch&Add(x, e_1) and P_2 executes $\text{Ans}_2 \leftarrow$ Fetch&Add(x, e_2) simultaneously on the shared variable x. If the request from P_1 is executed ahead of that from P_2, the following values are returned:

$$\begin{array}{lcl} \text{Ans}_1 & \leftarrow & x \\ \text{Ans}_2 & \leftarrow & x + e_1 \end{array} \tag{7.2}$$

If the execution order is reversed, the following values are returned:

$$\begin{array}{lcl} \text{Ans}_1 & \leftarrow & x + e_2 \\ \text{Ans}_2 & \leftarrow & x \end{array} \tag{7.3}$$

Regardless of the executing order, the value $x + e_1 + e_2$ is stored in memory. It is the responsibility of the switch box to form the sum $e_1 + e_2$, transmit the combined request Fetch&Add(x, $e_1 + e_2$), store the value e_1 (or e_2) in a wait buffer of the switch, and 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, as illustrated in Fig. 7.11 in four steps.



(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

Fig. 7.11 Two Fetch&Add operations are combined to access a shared variable simultaneously via a combining network

Applications and Drawbacks The Fetch&Add primitive is very effective in accessing sequentially allocated queue structures in parallel, or in forking out parallel processes with identical code that operate on different data sets.

Consider the parallel execution of N independent iterations of the following Do loop by p processors:

```
Doall N = 1 to 100
        <Code using N>
Endall
```

Each processor executes a Fetch&Add on N before working on a specific iteration of the loop. In this case, a unique value of N is returned to each processor, which is used in the code segment. The code for each processor is written as follows, with N being initialized as 1:

```
n ← Fetch&Add (N, 1)
While (n ≤ 100) Doall
        {Code using n}
        n ← Fetch&Add(N, 1)
Endall
```

The advantage of using a combining network to implement the Fetch&Add operation is achieved at a significant increase in network cost. According to NYU Ultracomputer experience, message queueing and combining in each bidirectional 2 × 2 switch box increased the network cost by a factor of at least 6 or more.

Additional switch cycles are also needed to make the entire operation an atomic memory operation. This may increase the network latency significantly. Multistage combining networks have the potential of supporting large-scale multiprocessors with thousands of processors. The problem of increased cost and latency may be alleviated with the use of faster and cheaper switching technology in the future.

Multistage Networks in Real Systems The IBM RP3 was designed to include 512 processors using a high-speed Omega network for reads or writes and a combining network for synchronization using Fetch&Adds. A 128-port Omega network in the RP3 had a bandwidth of 13 Gbytes/s using a 50-MHz clock.

Multistage Omega networks were also built into the Cedar multiprocessor (Kuck et al., 1986) at the University of Illinois and in the Ultracomputer (Gottlieb et al., 1983) at New York University.

The BBN Butterfly processor (TC2000) used 8 × 8 crossbar switch modules to build a two-stage 64 × 64 Butterfly network for a 64-processor system, and a three-stage 512 × 512 Butterfly switch (see Fig. 7.10) for a 512-processor system in the TC2000 Series. The switch hardware was clocked at 38 MHz with a 1-byte data path. The maximum interprocessor bandwidth for a 64-processor TC2000 was designed at 2.4 Gbytes/s.

The Cray Y-MP multiprocessor used 64-, 128-, or 256-way interleaved memory banks, each of which could be accessed via four ports. Crossbar networks were used between the processors and memory banks in all Cray multiprocessors. The Alliant FX/2800 used crossbar interconnects between seven four-processor (i860) boards plus one I/0 board and eight shared, interleaved cache boards which were connected to the physical memory via a memory bus.

7.2 CACHE COHERENCE AND SYNCHRONIZATION MECHANISMS

Cache coherence protocols for coping with the multicache inconsistency problem are considered below. Snoopy protocols are designed for bus-connected systems. Directory–based protocols apply to network-connected systems. Finally, we study hardware support for fast synchronization. Software-implemented synchronization will be discussed in Chapter 11.

7.2.1 The Cache Coherence Problem

In a memory hierarchy for a multiprocessor system, data inconsistency may occur between adjacent levels or within the same level. For example, the cache and main memory may contain inconsistent copies of the same data object. Multiple caches may possess different copies of the same memory block because multiple processors operate asynchronously and independently.

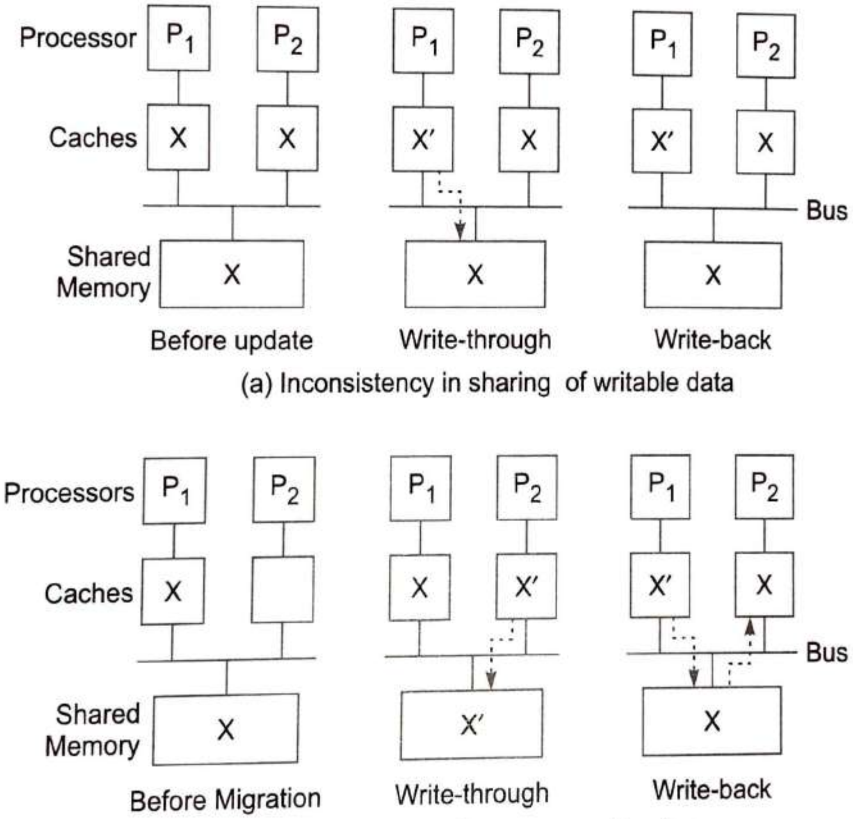
Caches in a multiprocessing environment introduce the *cache coherence problem*. When multiple processors maintain locally cached copies of a unique shared-memory location, any local modification of the location can result in a globally inconsistent view of memory. Cache coherence schemes prevent this problem by maintaining a uniform state for each cached block of data. Cache inconsistencies caused by data sharing, process migration, or I/O are explained below.

Inconsistency in Data Sharing The cache inconsistency problem occurs only when multiple private caches are used. In general, three sources of the problem are identified: *sharing of writable data, process migration*, and *I/O activity*. Figure 7.12 illustrates the problems caused by the first two sources. Consider a multiprocessor with two processors, each using a private cache and both sharing the main memory. Let *X* be a shared data element which has been referenced by both processors. Before update, the three copies of X are consistent.

If processor P_1 writes new data X' into the cache, the same copy will be written immediately into the shared memory under a *write-through* policy. In this case, inconsistency occurs between the two copies (X' and X) in the two caches (Fig. 7.12a).

On the other hand, inconsistency may also occur when a *write-back* policy is used, as shown on the right in Fig. 7.12a. The main memory will be eventually updated when the modified data in the cache are replaced or invalidated.

Process Migration and I/O Figure 7.12b 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. In the middle, a process migrates from processor 2 to processor 1 when using write-through caches.



(a) Inconsistency in sharing of writable data

(b) Inconsistency after process migration

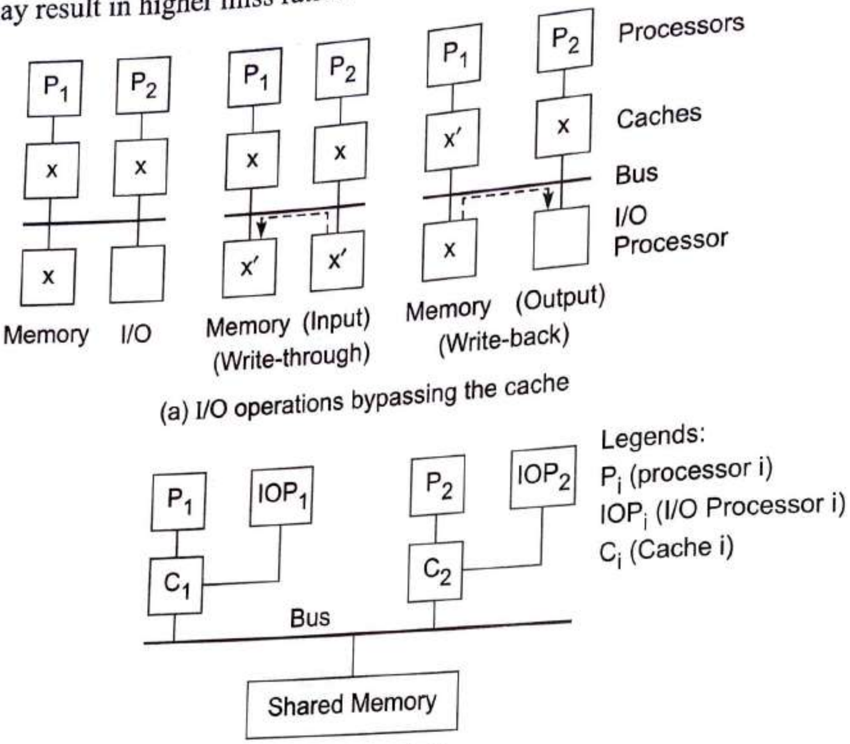
Fig. 7.12 **Cache coherence problems in data sharing and in process migration (Adapted from Dubois, Scheurich, and Briggs 1988)**

In both cases, inconsistency appears between the two cache copies, labeled X and X'. Special precautions must be exercised to avoid such inconsistencies. A coherence protocol must be established before processes can safely migrate from one processor to another.

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

When the I/O processor *loads* a new data X' into the main memory, bypassing the write through caches (middle diagram in Fig. 7.13a), inconsistency occurs between cache 1 and the shared memory. When outputting a data directly from the shared memory (bypassing the caches), the write-back caches also create inconsistency.

One possible solution to the I/O inconsistency problem is to attach the I/O processors (IOP_1 and IOP_2) to the private caches (C_1 and C_2), respectively, as shown in Fig. 7.13b. This way I/O processors share caches with the CPU. The I/O consistency can be maintained if cache-to-cache consistency is maintained via the bus. An obvious shortcoming of this scheme is the likely increase in cache perturbations and the poor locality of I/O data, which may result in higher miss ratios.



(a) I/O operations bypassing the cache

(b) A possible solution

Fig. 7.13 Cache inconsistency after an I/O operation and a possible solution (Adapted from Dubois, Scheurich, and Briggs, 1988)

Two Protocol Approaches Many of the early commercially available multiprocessors used bus-based memory systems. A bus is a convenient device for ensuring cache coherence because it allows all processors in the system to observe ongoing memory transactions. If a bus transaction threatens the consistent state of a locally cached object, the cache controller can take appropriate actions to invalidate the local copy. Protocols using this mechanism to ensure coherence are called *snoopy protocols* because each cache snoops on the transactions of other caches.

On the other hand, scalable multiprocessor systems interconnect processors using short point-to-point links in direct or multistage networks. Unlike the situation in buses, the bandwidth of these networks increases as more processors are added to the system. However, such networks do not have a convenient snooping mechanism and do not provide an efficient broadcast capability. In such systems, the cache coherence problem can be solved using some variant of directory schemes.

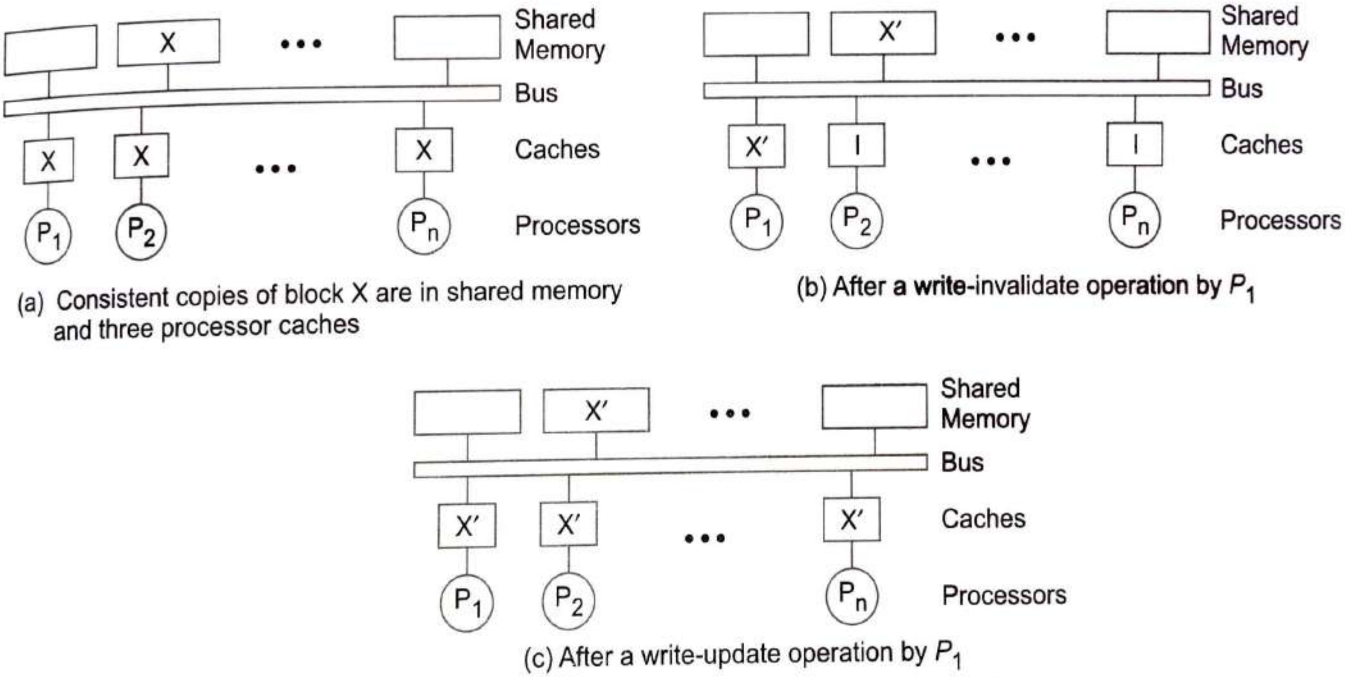
In general, a cache coherence protocol consists of the set of possible states in the local caches, the state in the shared memory, and the state transitions caused by the messages transported through the interconnection network to keep memory coherent. In what follows, we first describe the snoopy protocols and then the directory-based protocols. Other approaches to designing a scalable cache coherence interface will be studied in Chapter 9.

7.2.2 Snoopy Bus Protocols

In using private caches associated with processors tied to a common bus, two approaches have been practiced for maintaining cache consistency: *write-invalidate* and *write-update* policies. Essentially, 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.

Snoopy protocols achieve data consistency among the caches and shared memory through a bus watching mechanism. As illustrated in Fig. 7.14, two snoopy bus protocols create different results. Consider three processors (P_1, P_2, and P_n) maintaining consistent copies of block X in their local caches (Fig. 7.14a) and in the shared-memory module marked X.

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 (denoted I in Fig. 7.14b). Invalidated blocks are sometimes called *dirty*, meaning they should not be used. The *write-update protocol* (Fig. 7.14c) demands the new block content X' be broadcast to all cache copies via the bus. The memory copy is also updated if write-through caches are used. In using write-back caches, the memory copy is updated later at block replacement time.



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

(b) After a write-invalidate operation by P_1

(c) After a write-update operation by P_1

Fig. 7.14 Write-invalidate and write-update coherence protocols for write through caches (1: invalidate)

Write-Through Caches The states of a cache block copy change with respect to *read*, *write*, and *replacement* operations in the cache. Figure 7.15 shows the state transitions for two basic write-invalidate snoopy protocols developed for write-through and write-back caches, respectively. A block copy of a write-through cache i attached to processor i can assume one of two possible cache states: *valid* or *invalid* (Fig. 7.15a).

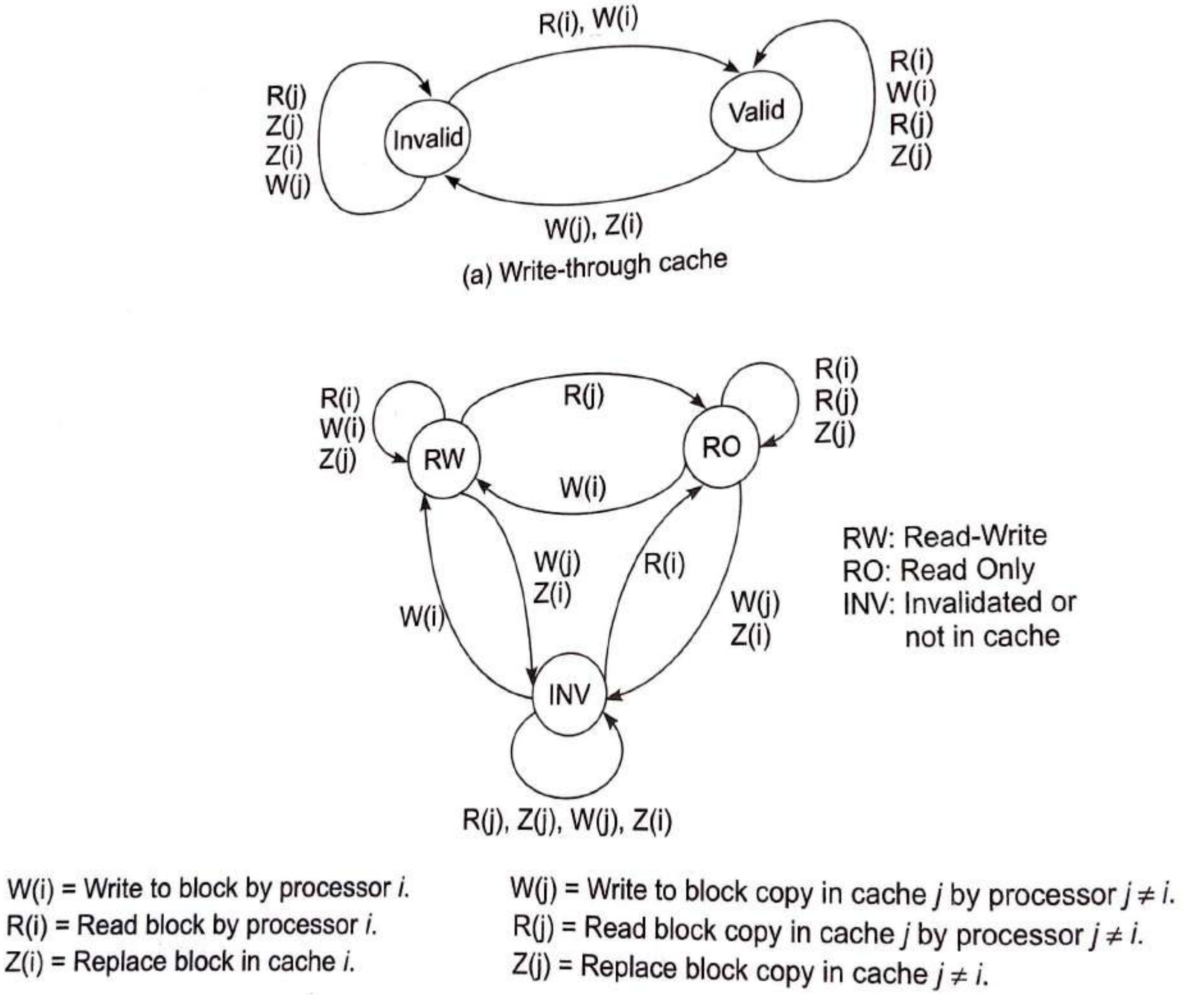
A remote processor is denoted j, where $j \neq i$. For each of the two cache states, six possible events may take place. Note that all cache copies of the same block use the same transition graph in making state changes.

In a *valid* state (Fig. 7.15a), all processors can *read* ($R(i)$, $R(j)$) safely. Local processor i can also *write* ($W(i)$) safely in a *valid* state. The *invalid* state corresponds to the case of the block either being invalidated or being replaced ($Z(i)$ or $Z(j)$).

Wherever a remote processor *writes* ($W(j)$) into its cache copy, all other cache copies become invalidated. 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.

The fraction of *write cycles* on the bus is higher than the fraction of *read cycles* in a write-through cache, due to the need for request invalidations. The *cache directory* (registration of cache states) can be made in dual copies or dual-ported to filter out most invalidations. In case locks are cached, an atomic Test&Set must be enforced.

Write-Back Caches The *valid* state of a write-back cache can be further split into two cache states, labeled RW (*read-write*) and RO (*read-only*) as shown in Fig. 7.15b. 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*.



(a) Write-through 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

Fig. 7.15 Two state-transition graphs for a cache block using write-invalidate snoopy protocols (Adapted from Dubois, Scheurich, and Briggs, 1988)

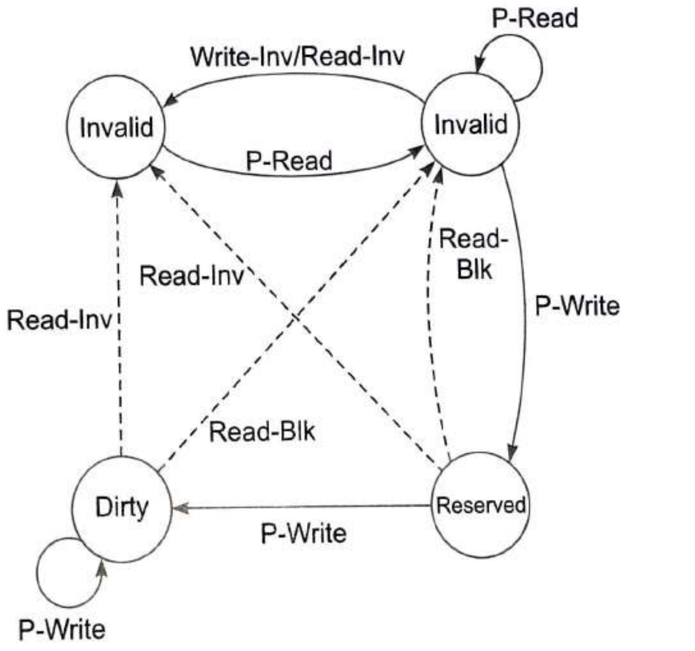
When the memory owns a block, caches can contain only the RO copies of the block. In other words, multiple copies may exist in the RO state and every processor having a copy (called a *keeper* of the copy) can *read* ($R(i)$, $R(j)$) the copy safely.

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. The RW state corresponds to only one cache copy existing in the entire system owned by the local processor i. *Read* ($R(i)$) and *write* ($W(i)$) can be safely performed in the RW state. From either the RO state or the INV state, the cache block becomes uniquely owned when a local *write* ($W(i)$) takes place.

Other state transitions in Fig. 7.15b can be similarly figured out. Before a block is modified, ownership for exclusive access must first be obtained by a *read-only* bus transaction which is broadcast to all caches and memory. If a modified block copy exists in a remote cache, memory must first be updated, the copy invalidated, and ownership transferred to the requesting cache.

Write-once Protocol James Goodman (1983) proposed a cache coherence protocol for bus-based multiprocessors. This scheme combines the advantages of both write-through and write-back invalidations. In order to reduce bus traffic, the very first *write* of a cache block uses a write-through policy.

This will result in a consistent memory copy while all other cache copies are invalidated. After the first *write*, shared memory is updated using a write-back policy. This scheme can be described by the four-state transition graph shown in Fig. 7.16. The four cache states are defined below:



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

Fig. 7.16 Goodman's write-once cache coherence protocol using the write invalidate policy on write-back caches (Adapted from James Goodman 1983, reprinted from Stenstrom, *IEEE Computer*, June 1990)

- *Valid:* The cache block, which is consistent with the memory copy, has been *read* from shared memory and has not been modified.
- *Invalid:* The block is not found in the cache or is inconsistent with the memory copy.
- *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.

- *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).

To maintain consistency, the protocol requires two different sets of commands. The solid lines in Fig. 7.16 correspond to access commands issued by a local processor labeled *read-miss*, *write-hit*, and *write-miss*. Whenever a *read-miss* occurs, the *valid* state is entered.

The first *write-hit* leads to the *reserved state*. The second *write-hit* leads to the *dirty* state, and all future *write-hits* stay in the *dirty* state. Whenever a *write-miss* occurs, the cache block enters the *dirty* state.

The dashed lines correspond to invalidation commands issued by remote processors via the snoopy bus. The *read-invalidate* command reads a block and invalidates all other copies. The *write-invalidate* command invalidates all other copies of a block. The *bus–read* command corresponds to a normal memory *read* by a remote processor via the bus

Cache Events and Actions The memory-access and invalidation commands trigger the following events and actions:

- *Read-miss:* When a processor wants to read a block that is not in the cache, a *read-miss* occurs. A *bus-read* operation will be initiated. If no *dirty* copy exists, then main memory has a consistent copy and supplies a copy to the requesting cache. 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. In all cases, the cache copy will enter the *valid* state after a read-miss.
- *Write-hit:* If the copy is in the *dirty* or *reserved* state, the *write* can be carried out locally and the new state is *dirty*. If the new state is *valid*, a *write-invalidate* command is broadcast to all caches, invalidating their copies. The shared memory is *written through*, and the resulting state is *reserved* after this first *write*.
- *Write-miss*: 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. This is accomplished by sending a *read-invalidate* command which will invalidate all cache copies. The local copy is thus updated and ends up in a *dirty* state.
- *Read-hit*: Read-hits can always be performed in a local cache without causing a state transition or using the snoopy bus for invalidation.
- *Block Replacement:* If a copy is *dirty*, it has to be written back to main memory by block replacement. If the copy is *clean* (i.e., in either the *valid, reserved*, or *invalid* state), no replacement will take place.

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*. Most standard buses cannot support this inhibition operation.

The IEEE Futurebus+ proposed to include this special bus provision. Using a write-through policy after the first *write* and using a write-back policy in all additional *writes* eliminates unnecessary invalidations.

Snoopy cache protocols are popular in bus-based multiprocessors because of their simplicity of implementation. The write-invalidate policies were implemented on the Sequent Symmetry multiprocessor and on the Alliant FX multiprocessor.

Besides the DEC Firefly multiprocessor, the Xerox Palo Alto Research Center implemented another write-update protocol for its Dragon multiprocessor workstation. The Dragon protocol avoids updating memory until replacement, in order to improve the efficiency of intercache transfers.

Multilevel Cache Coherence To maintain consistency among cache copies at various levels, Wilson proposed an extension to the write-invalidate protocol used on a single bus. Consistency among cache copies at the same level is maintained in the same way as described above. Consistency of caches at different levels is illustrated in Fig. 7.3.

An invalidation must propagate vertically up and down in order to invalidate all copies in the shared caches at level 2. Suppose processor P_1 issues a *write* request. The *write* request propagates up to the highest level and invalidates copies in C_{20}, C_{22}, C_{16}, and C_{18}, as shown by the arrows to all the shaded copies.

High-level caches such as C_{20} keep track of dirty blocks beneath them. A subsequent *read* request issued by P_7 will propagate up the hierarchy because no copies exist. When it reaches the top level, cache C_{20} issues a flush request down to cache C_{11} and the dirty copy is supplied to the private cache associated with processor P_7. Note that higher-level caches act as filters for consistency control. An invalidation command or a read request will not propagate down to clusters that do not contain a copy of the corresponding block. The cache C_{21} acts in this manner.

Protocol Performance Issues The performance of any snoopy protocol depends heavily on the workload patterns and implementation efficiency. The main motivation for using the snooping mechanism is to reduce bus traffic, with a secondary goal of reducing the effective memory-access time. The block size is very sensitive to cache performance in write-invalidate protocols, but not in write-update protocols.

For a uniprocessor system, bus traffic and memory-access time are mainly contributed by cache misses. The miss ratio decreases when block size increases. However, as the block size increases to a *data pollution* point, the miss ratio starts to increase. For larger caches, the data pollution point appears at a larger block size.

For a system requiring extensive process migration or synchronization, the write-invalidate protocol will perform better. However, a cache miss can result for an invalidation initiated by another processor prior to the cache access. Such *invalidation misses* may increase bus traffic and thus should be reduced.

Extensive simulation results have suggested that bus traffic in a multiprocessor may increase when the block size increases. Write-invalidate also facilitates the implementation of synchronization primitives. Typically, the average number of invalidated cache copies is rather small (one or two) in a small multiprocessor.

The write-update protocol requires a bus broadcast capability. This protocol also can avoid the ping-pong effect on data shared between multiple caches. Reducing the sharing of data will lessen bus traffic in a write-update multiprocessor. However, write–update cannot be used with long write bursts. Only through extensive program traces (trace-driven simulation) can one reveal the cache behavior, hit ratio, bus traffic, and effective memory-access time.

7.2.3 Directory-Based Protocols

A write-invalidate protocol may lead to heavy bus traffic caused by *read-misses*, resulting from the processor updating a variable and other processors trying to read the same variable. On the other hand, the write-update protocol may update data items in remote caches which will never be used by other processors. In fact, these problems pose additional limitations in using buses to build large multiprocessors.

When a multistage or packet switched network is used to build a large multiprocessor with hundreds of processors, the snoopy cache protocols must be modified to suit the network capabilities. Since broadcasting is expensive to perform in such a network, consistency commands will be sent only to those caches that keep a copy of the block. This leads to *directory-based protocols* for network-connected multiprocessors.

Directory Structures In a multistage or packet switched network, cache coherence is supported by using cache directories to store information on where copies of cache blocks reside. Various directory-based protocols differ mainly in how the directory maintains information and what information it stores.

Tang (1976) proposed the first directory scheme, which used a *central directory* containing duplicates of all cache directories. This central directory, providing all the information needed to enforce consistency, is usually very large and must be associatively searched, like the individual cache directories. Contention and long search times are two drawbacks in using a central directory for a large multiprocessor.

A distributed-directory scheme was proposed by Censier and Feautrier (1978). Each memory module maintains a separate directory which records the state and presence information for each memory block. The state information is local, but the presence information indicates which caches have a copy of the block.

In Fig. 7.17, a *read-miss* (thin lines) in cache 2 results in a request sent to the memory module. The memory controller retransmits the request to the dirty copy in cache 1. This cache *writes back* its copy. The memory module can supply a copy to the requesting cache. In the case of a *write-hit* at cache 1 (bold lines), a command is sent to the memory controller, which sends invalidations to all caches (cache 2) marked in the presence vector residing in the directory D_1.

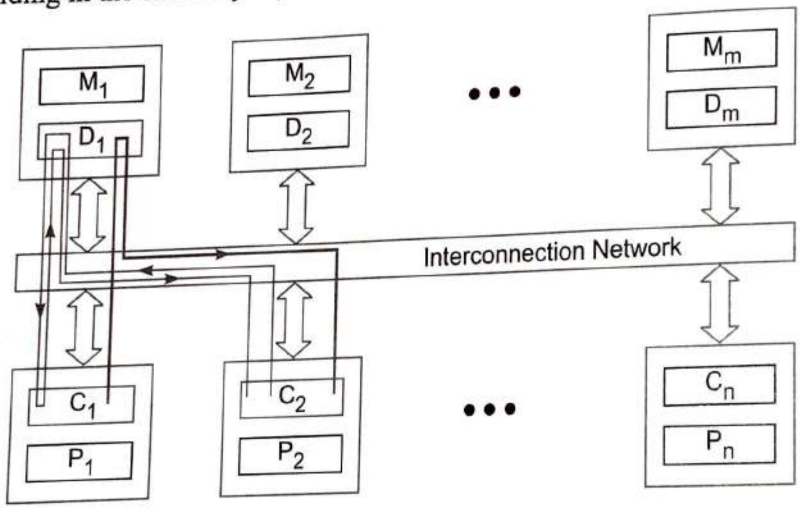


Fig. 7.17 Basic concept of a directory-based cache coherence scheme (Courtesy of Censier and Feautrier, *IEEE Trans. Computers,* Dec. 1978)

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

Different types of directory protocols fall under three primary categories: *full map directories, limited directories,* and *chained directories*. Full-map directories store enough data associated with each block in global memory so that every cache in the system can simultaneously store a copy of any block of data. That is, each directory entry contains *N* pointers, where *N* is the number of processors in the system.

Limited directories differ from full-map directories in that they have a fixed number of pointers per entry, regardless of the system size. Chained directories emulate the full-map schemes by distributing the directory among the caches. The following descriptions of the three classes of cache directories are based on the original classification by Chaiken, Fields, Kwihara, and Agarwal (1990):

Full-Map Directories The full-map protocol implements directory entries with one bit per processor and a dirty bit. Each bit represents the status of the block in the corresponding processor's cache (present or absent). If the dirty bit is set, then one and only one processor's bit is set and that processor can write into the block.

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. The cache coherence protocol must keep the state bits in the memory directory and those in the cache consistent.

Figure 7.18a illustrates three different states of a full-map directory. In the first state, location X is missing in all of the caches in the system. The second state results from three caches (C1, C2, and C3) requesting copies of location X. Three pointers (processor bits) are set in the entry to indicate the caches that have copies of the block of data. In the first two states, the dirty bit on the left side of the directory entry is set to clean (C), indicating that no processor has permission to write to the block of data. The third state results from cache C3 requesting write permission for the block. In the final state, the dirty bit is set to dirty (D), and there is a single pointer to the block of data in cache C3.

Let us examine the transition from the second state to the third state in more detail. Once processor P3 issues the write to cache C3, the following events will take place:

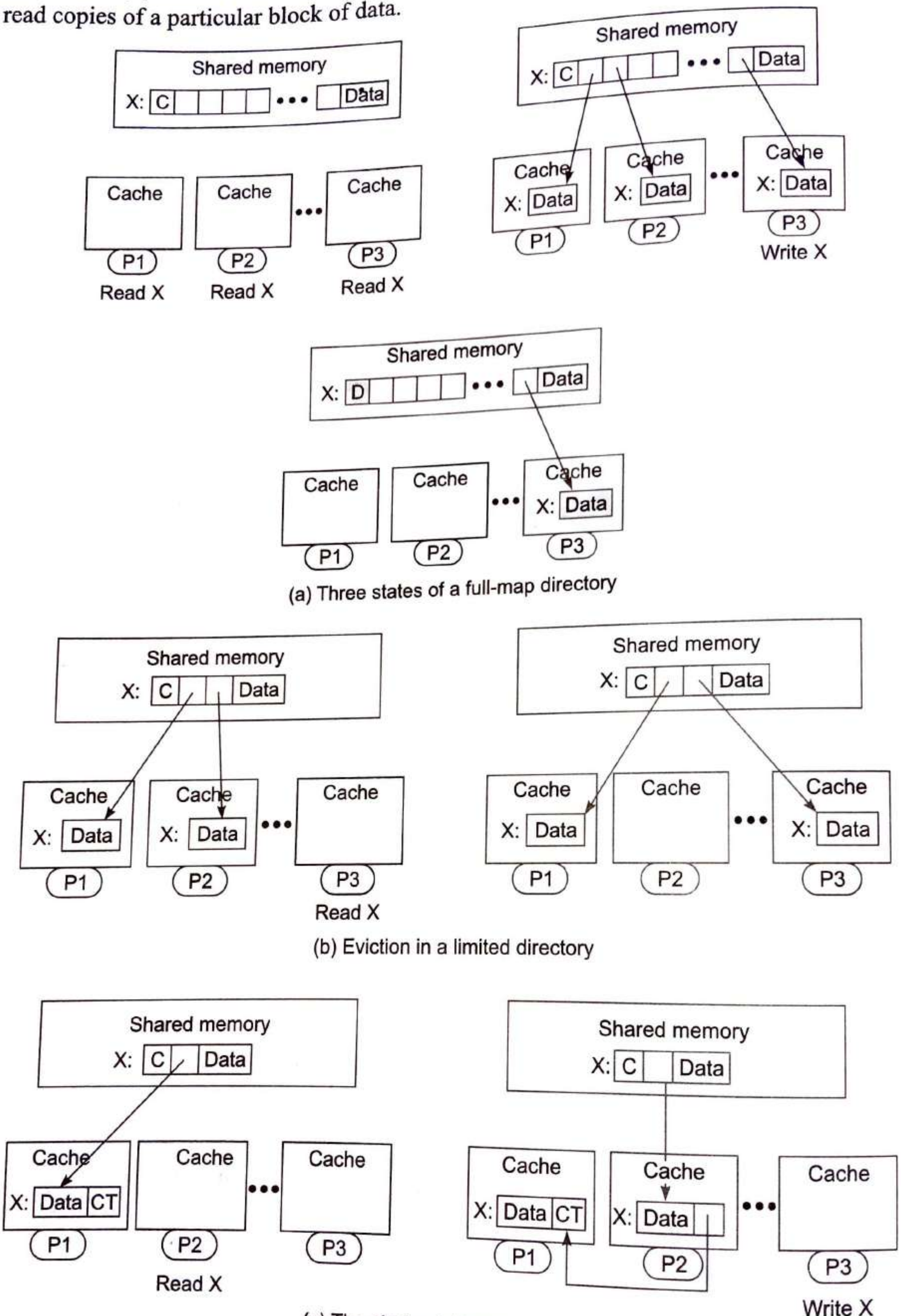
(1) Cache C3 detects that the block containing location X is valid but that the processor does not have permission to write to the block, indicated by the block's write-permission bit in the cache.
(2) Cache C3 issues a write request to the memory module containing location X and stalls processor P3.
(3) The memory module issues invalidate requests to caches Cl and C2.
(4) Caches Cl and C2 receive the invalidate requests, set the appropriate bit to indicate that the block containing location X is invalid, and send acknowledgments back to the memory module.
(5) The memory module receives the acknowledgments, sets the dirty bit, clears the pointers to caches Cl and C2, 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.

The memory module waits to receive the acknowledgments before allowing processor P3 to complete its write transaction. By waiting for acknowledgments, the protocol guarantees that the memory system ensures sequential consistency. The full-map protocol provides a useful upper bound for the performance of centralized directory-based cache coherence. However, it is not scalable due to excessive memory overhead.

Because the size of the directory entry associated with each block of memory is proportional to the number of processors, the memory consumed by the directory is proportional to the size of memory $O(N)$ multiplied by the size of the directory $O(N)$. Thus, the total memory overhead scales as the square of the number of processors $O(N^2)$.

Limited Directories Limited directory protocols are designed to solve the directory size problem. Restricting the number of simultaneously cached copies of any particular block of data limits the growth of the directory to a constant factor.

A directory protocol can be classified as $Dir_i\ X$ using the notation from Agarwal et al (1988). The symbol i stands for the number of pointers, and X is NB for a scheme with no broadcast. A full-map scheme without broadcast is represented as $Dir_N\,NB$. A limited directory protocol that uses $i < N$ pointers is denoted $Dir_i\,NB$. The limited directory protocol is similar to the full-map directory, except in the case when more than i caches request read copies of a particular block of data.



(a) Three states of a full-map directory

(b) Eviction in a limited directory

(c) The chained directory

Fig. 7.18 Three types of cache directory protocols (Courtesy of Chaiken et al., *IEEE Computer*, June 1990)

Figure 7.18b shows the situation when three caches request read copies in a memory system with a $Dir_2\ NB$ protocol. In this case, we can view the two-pointer directory as a two-way set-associative cache of pointers to shared copies. When cache C3 requests a copy of location X, the memory module must invalidate the copy in either cache Cl or cache C2. This process of pointer replacement is called *eviction*. Since the directory acts as a set-associative cache, it must have a pointer replacement policy.

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.

Directory pointers in a $Dir_i\ NB$ protocol encode binary processor identifiers, so each pointer requires $\log_2 N$ bits of memory, where N is the number of processors in the system. Given the same assumptions as for the full-map protocol, the memory overhead of limited directory schemes grows as $O(N \log_2 N)$.

These protocols are considered scalable with respect to memory overhead because the resource required to implement them grows approximately linearly with the number of processors in the system. $Dir_i\ B$ protocols allow more than i copies of each block of data to exist, but they resort to a broadcast mechanism when more than i cached copies of a block need to be invalidated. However, point-to-point interconnection networks do not provide an efficient systemwide broadcast capability. In such networks, it is difficult to determine the completion of a broadcast to ensure sequential consistency.

Chained Directories Chained directories realize the scalability of limited directories without restricting the number of shared copies of data blocks. This type of cache coherence scheme is called a *chained* scheme because it keeps track of shared copies of data by maintaining a chain of directory pointers.

The simpler of the two schemes implements a singly linked chain, which is best described by example (Fig. 7.18c). Suppose there are no shared copies of location X. If processor P1 reads location X, the memory sends a copy to cache C1, along with a *chain termination* (CT) pointer. The memory also keeps a pointer to cache C1. Subsequently, when processor P2 reads location X, the memory sends a copy to cache C2, along with the pointer to cache C1. The memory then keeps a pointer to cache C2.

By repeating the above step, all of the caches can cache a copy of the location X. If processor P3 writes to location X, it is necessary to send a data invalidation message down the chain. To ensure sequential consistency, the memory module denies processor P3 write permission until the processor with the chain termination pointer acknowledges the invalidation of the chain. Perhaps this scheme should be called a *gossip* protocol (as opposed to a snoopy protocol) because information is passed from individual to individual rather than being spread by covert observation.

The possibility of cache block replacement complicates chained-directory protocols.

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 P_i reads location Y, it must first 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.

The second scheme can be implemented by a less complex protocol than the first. In either case, sequential consistency is maintained by locking the memory location while invalidations are in progress. Another solution to the replacement problem is to use a doubly linked chain. This scheme maintains forward and backward chain pointers for each cached copy so that the protocol does not have to traverse the chain when there is a cache replacement. The doubly linked directory optimizes the replacement condition at the cost of a larger average message block size (due to the transmission of extra directory pointers), twice the pointer memory in the caches, and a more complex coherence protocol.

Although the chained protocols are more complex than the limited directory protocols, they are still scalable in terms of the amount of memory used for the directories. The pointer sizes grow as the logarithm of the number of processors, and the number of pointers per cache or memory block is independent of the number of processors.

Cache Design Alternatives The relative merits of physical address caches and virtual address caches have to be judged based on the access time, the aliasing problem, the flushing problem, OS kernel overhead, special tagging at the process level, and cost/performance considerations. Beyond the use of private caches, three design alternatives are suggested below.

Each of the design alternatives has its own advantages and shortcomings. There exists insufficient evidence to determine whether any of the alternatives is always better or worse than the use of private caches. More research and trace data are needed to apply these cache architectures in designing high-performance multiprocessors.

Shared Caches An alternative approach to maintaining cache coherence is to completely eliminate the problem by using *shared caches* attached to shared-memory modules. No private caches are allowed in this case. This approach will reduce the main memory access time but contributes very little to reducing the overall memory-access time and to resolving access conflicts.

Shared caches can be built as second-level caches. Sometimes, one can make the second-level caches partially shared by different clusters of processors. Various cache architectures are possible if private and shared caches are both used in a memory hierarchy. The use of shared cache alone may be against the scalability of the entire system. Tradeoffs between using private caches, caches shared by multiprocessor clusters, and shared main memory are interesting topics for further research.

Noncacheable Data Another approach is not to cache shared writable data. Shared data are *noncacheable*, and only instructions or private data are *cacheable* in local caches. Shared data include locks, process queues, and any other data structures protected by critical sections.

The compiler must tag data as either *cacheable* or *noncacheable*. Special hardware tagging must be used to distinguish them. Cache systems with cacheable and noncacheable blocks demand more support from hardware and compilers.

Cache Flushing A third approach is to use *cache flushing* every time a synchronization primitive is executed. This may work well with transaction processing multiprocessor systems. Cache flushes are slow unless special hardware is used. This approach does not solve I/O and process migration problems.

Flushing can be made very selective by the compiler in order to increase efficiency. Cache flushing at synchronization, I/O, and process migration may be carried out unconditionally or selectively. Cache flushing is more often used with virtual address caches.

7.2.4 Hardware Synchronization Mechanisms

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. Synchronization enforces correct sequencing of processors and ensures mutually exclusive access to shared writable data. Synchronization can be implemented in software, firmware, and hardware through controlled sharing of data and control information in memory.

Multiprocessor systems use hardware mechanisms to implement low-level or primitive synchronization operations, or use software (operating system) level synchronization mechanisms such as *semaphores* or *monitors*. Only hardware synchronization mechanisms are studied below. Software approaches to synchronization will be treated in Chapter 10.

Atomic Operations Most multiprocessors are equipped with hardware mechanisms for enforcing atomic operations such as memory *read*, *write*, or *read-modify-write* operations which can be used to implement some synchronization primitives. Besides atomic memory operations, some interprocessor interrupts can be used for synchronization purposes. For example, the synchronization primitives, Test&Set (*lock*) and Reset (*lock*), are defined below:

```
Test&Set  (lock)
              $temp \leftarrow lock$;   $lock \leftarrow 1$;
              return $temp$                                    (7.4)
Reset  (lock)
              $lock \leftarrow 0$
```

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. This synchronization primitive may tie up some bus cycles while a processor enters busy-waiting on the *spin lock*. To avoid spinning, interprocessor interrupts can be used.

A lock tied to an interrupt is called a *suspend lock*. Using such a lock, a process does not relinquish the processor while it is waiting. Whenever the process fails to open the lock, it records its status and disables all interrupts aiming at the lock. When the lock is open, it signals all waiting processors through an interrupt. A similar primitive, Compare&Swap, was implemented in IBM 370 mainframes.

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

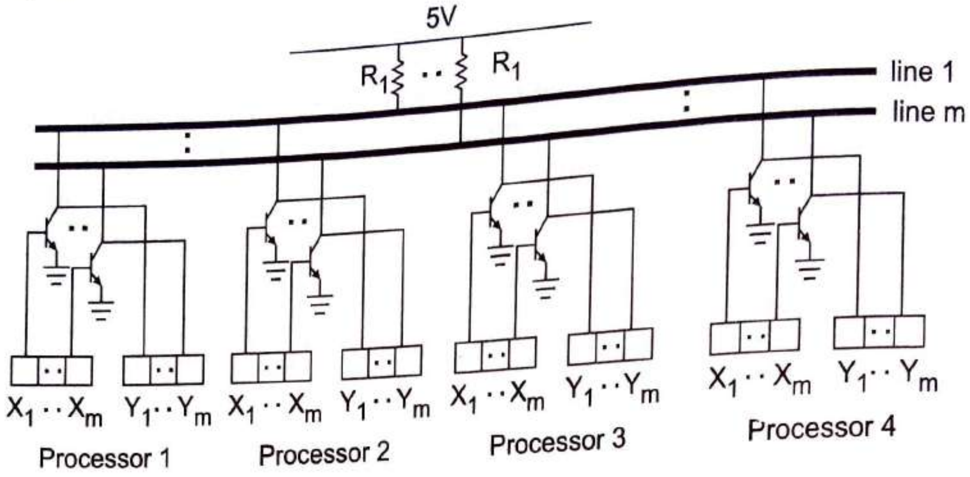
Wired Barrier Synchronization A wired-NOR logic is shown in Fig. 7.19 for implementing a barrier mechanism for fast synchronization. Each processor uses a dedicated control vector $X = (X_1, X_2, ..., X_m)$ and accesses a common monitor vector $Y = (Y_1, Y_2, ..., Y_m)$ in shared memory, where m corresponds to the barrier lines used.

The number of barrier lines needed for synchronization depends on the multiprogramming degree and the size of the multiprocessor system. Each control bit X_i is connected to the base (input) of a probing transistor. The monitor bit Y_i checks the collector voltage (output) of the transistor.

Each barrier line is wired-NOR to n transistors from n processors. Whenever bit X_i is raised to high (1), the corresponding transistor is closed, pulling down (0) the level of barrier line i. The wired-NOR connection implies that line i will be high (1) only if control bits X_i from all processors are low (0).

This demonstrates the ability to use the control bit X_i to signal the completion of a process on processor i. The bit X_i is set to 1 when a process is initiated and reset to 0 when the process finishes its execution.

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. This timing is watched by all processors through snooping on the Y_i bits. Thus only one barrier line is needed to monitor the initiation and completion of a single synchronization involving many concurrent processes.



(a) Barrier lines and interface logic

Step 1: Forking (use of one barrier line)

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

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

	Process 1	Process 2	Process 3	Process 4
X	0	1	0	1
Y	0	0	0	0

Step 3: All processes reach the synchronization point

	Process 1	Process 2	Process 3	Process 4
X	0	0	0	0
Y	1	1	1	1

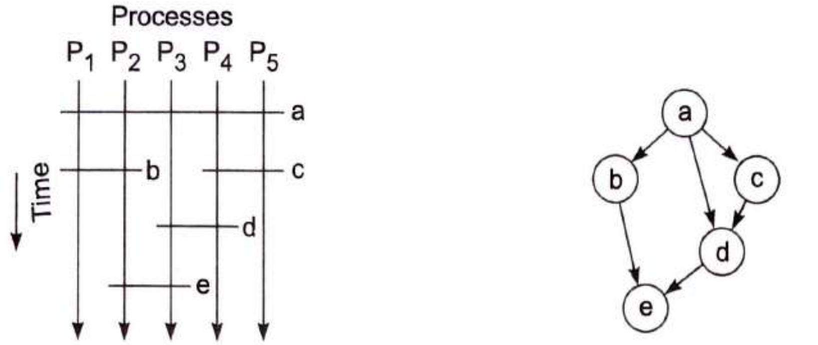
(b) Synchronization steps

Fig. 7.19 The synchronization of four independent processes on four processors using one wired-NOR barrier line (Adapted from Hwang and Shang, *Proc. Int. Conf. Parallel Processing*, 1991)

Multiple barrier lines can be used simultaneously to monitor several synchronization points. Figure 7.19 shows the synchronization of four processes residing on four processors using one barrier line. Note that other barrier lines can be used to synchronize other processes at the same time in a multiprogrammed multiprocessor environment.

Example 7.2 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. 7.20.



(a) Synchronization patterns

(b) Precedence graph

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

	Processor 1	Processor 2	Processor 3	Processor 4	Processor 5
X	11000	11001	10011	10110	10100
Y	00000	00000	00000	00000	00000
Step 1: Synchronization at barrier a					
X	01000	01001	00011	00110	00100
Y	10000	10000	10000	10000	10000
Step 2a: Synchronization at barrier b					
X	00000	00001	00011	00110	00100
Y	11000	11000	11000	11000	11000
Step 2b: Synchronization at barrier c					
X	00000	00001	00011	00010	00000
Y	11100	11100	11100	11100	11100
Step 3: Synchronization at barrier d					
X	00000	00001	00001	00000	00000
Y	11110	11110	11110	11110	11110
Step 4: Synchronization at barrier e					
X	00000	00000	00000	00000	00000
Y	11111	11111	11111	11111	11111

(c) Synchronization steps

Fig. 7.20 The synchronization of five partially ordered processes using wired-NOR barrier lines (Adapted from Hwang and Shang, *Proc. Int. Conf. Parallel Processing*, 1991)

Each node was on a separate board. For numeric nodes, the processor and floating-point units were on the same i860 chip. The local memory took up most of the board space. The external I/O interface was implemented only on the boundary nodes with a computational array. The message I/O intcrface was required for message passing between local nodes and the mesh network. The *mesh-connected router* is shown in Fig. 7.25.

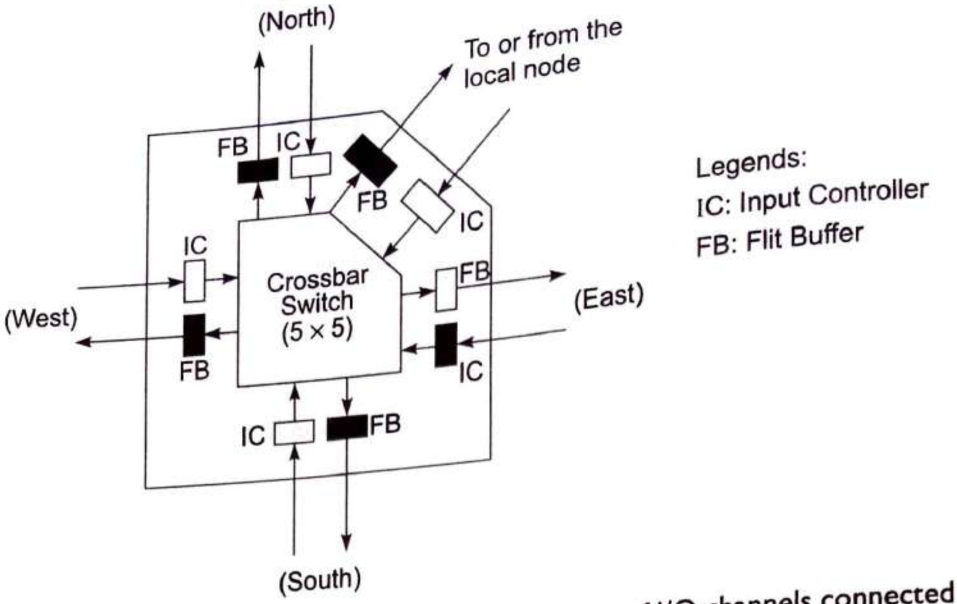


Fig. 7.25 The structure of a mesh-connected router with four pairs of I/O channels connected to neighboring routers

Each router had 10 I/O ports, 5 for input and 5 for output. Four pairs of I/O channels were used for mesh connection to the four neighbors at the north, south, east, and west nodes.

Flow control digits (flits) buffers were used at the end of input channels to hold the incoming flits. The concept of flits will be clarified in the next section. Besides four pairs of external channels, a fifth pair was used for internal connection between the router and the local node. A 5 × 5 crossbar switch was used to establish a connection between any input channel and any output channel.

The functions of the hardware router included pipelined message routing at the flit level and resolving buffer or channel deadlock situations to achieve deadlock-free routing. In the next section, we will explain various routing mechanisms and deadlock avoidance schemes.

All the I/O channels shown in Figs. 7.24 and 7.25 are *physical channels* which allow only one message (flit) to pass at a time. Through time-sharing, one can also implement *virtual channels* to multiplex the use of physical channels as described in the next section.

7.4 MESSAGE-PASSING MECHANISMS

Message passing in a multicomputer network demands special hardware and software support. In this section, we study the store-and-forward and wormhole routing schemes and analyze their communication latencies. We introduce the concept of virtual channels. Deadlock situations in a message-passing network are examined. We show how to avoid deadlocks using virtual channels.

Both deterministic and adaptive routing algorithms are presented for achieving deadlock-free message routing. We first study deterministic dimension-order routing schemes such as E-cube routing for hypercubes and X-Y routing for two-dimensional meshes. Then we discuss adaptive routing using virtual channels or virtual subnets. Besides one-to-one unicast routing, we will consider one-to-many multicast and one-to-all broadcast operations using virtual subnets and greedy routing algorithms.

7.4.1 Message-Routing Schemes

Message formats are introduced below. Refined formats led to the improvement from store-and-forward to wormhole routing in two generations of multicomputers. A handshaking protocol is described for asynchronous pipelining of successive routers along a communication path. Finally, latency analysis is conducted to show the time difference between the two routing schemes presented.

Message Formats Information units used in message routing are specified in Fig. 7.26. A *message* is the logical unit for internode communication. It is often assembled from an arbitrary number of fixed-length packets, thus it may have a variable length.

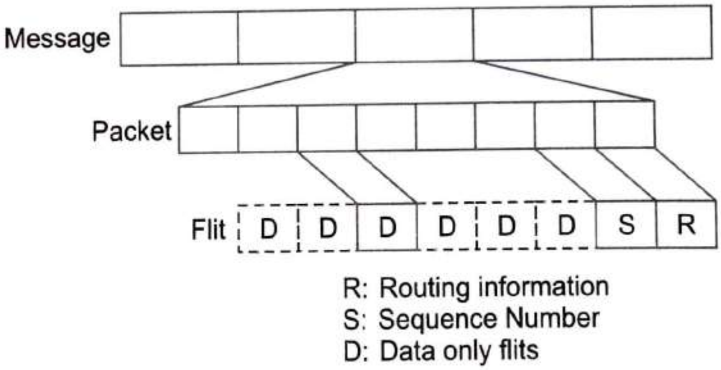


Fig. 7.26 The format of message, packets, and flits (control flow digits) used as information units of communication in a message-passing network

A *packet* is the basic unit containing the destination address for routing purposes. Because different packets may arrive at the destination asynchronously, a sequence number is needed in each packet to allow reassembly of the message transmitted.

A packet can be further divided into a number of fixed-length *flits* (flow control digits). Routing information (destination) and sequence number occupy the header flits. The remaining flits are the data elements of a packet.

In multicomputers with store-and-forward routing, packets are the smallest unit of information transmission. In wormhole-routed networks, packets are further subdivided into flits. The flit length is often affected by the network size.

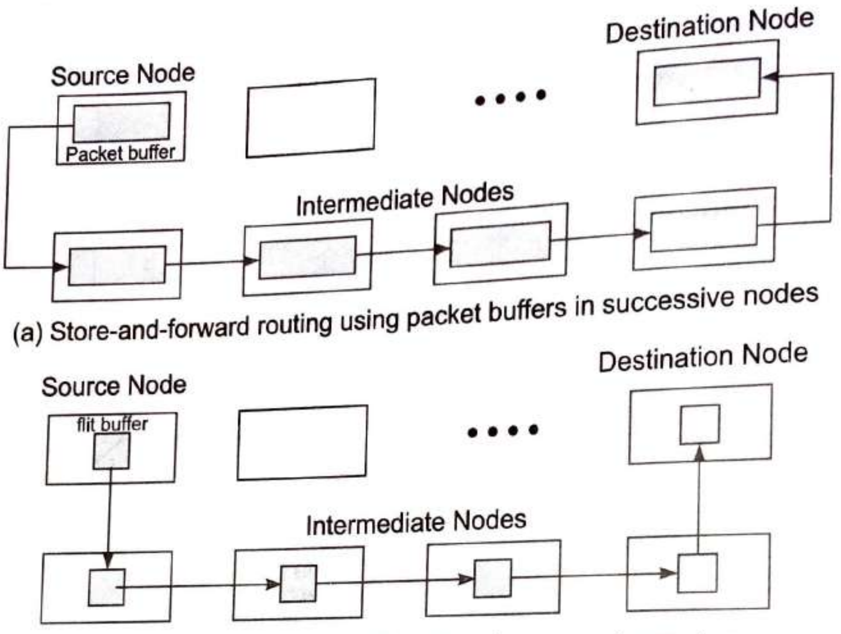
The packet length is determined by the routing scheme and network implementation. Typical packet lengths range from 64 to 512 bits. The sequence number may occupy one to two flits depending on the message length. Other factors affecting the choice of packet and flit sizes include channel bandwidth, router design, network traffic intensity, etc.

Store-and-Forward Routing Packets are the basic unit of information flow in a *store-and-forward* network. The concept is illustrated in Fig. 7.27a. Each node is required to use a packet buffer. A packet is transmitted from a source node to a destination node through a sequence of intermediate nodes.

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 a packet buffer in the receiving node are both available.

The latency in store-and-forward networks is directly proportional to the distance (the number of hops) between the source and the destination. This routing scheme was implemented in the first generation of multicomputers.

Wormhole Routing By subdividing the packet into smaller flits, latter generations of multicomputers implement the *wormhole routing* scheme, as illustrated in Fig. 7.27b. Flit buffers are used in the hardware routers attached to nodes. The transmission from the source node to the destination node is done through a sequence of routers.



(a) Store-and-forward routing using packet buffers in successive nodes

(b) Wormhole routing using flit buffers in successive routers

Fig. 7.27 Store-and-forward routing and wormhole routing (Courtesy of Lionel Ni, 1991)

All the flits in the same packet are transmitted in order as inseparable companions in a pipelined fashion. 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).

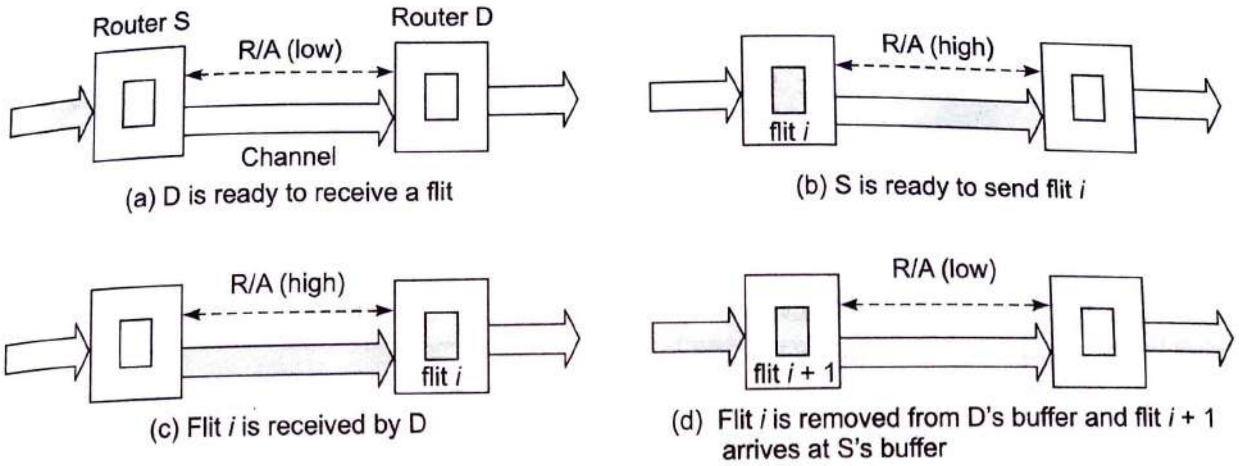
Only the header flit knows where the train (packet) is going. All the data flits (box cars) must follow the header flit. Different packets can be interleaved during transmission. However, the flits from different packets cannot be mixed up. Otherwise they may be towed to the wrong destinations.

We prove below that wormhole routing has a latency almost independent of the distance between the source and the destination.

Asynchronous Pipelining. The pipelining of successive flits in a packet is done asynchronously using a handshaking protocol as shown in Fig. 7.28. Along the path, a 1-bit *ready/request* (R/A) line is used between adjacent routers.

When the receiving router (D) is ready (Fig. 7.28a) to receive a flit (i.e. the flit buffer is available), it pulls the R/A line low. When the sending router (S) is ready (Fig. 7.28b), it raises the line high and transmits flit *i* through the channel.

While the flit is being received by D (Fig. 7.28c), the R/A line is kept high. After flit *i* is removed from D's buffer (i.e. is transmitted to the next node) (Fig. 7.28d), the cycle repeats itself for the transmission of the next flit *i* + 1 until the entire packet is transmitted.



(a) D is ready to receive a flit

(b) S is ready to send flit i

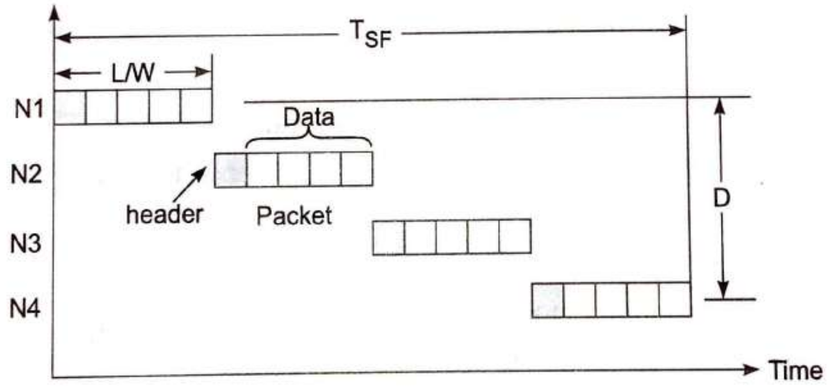
(c) Flit i is received by D

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

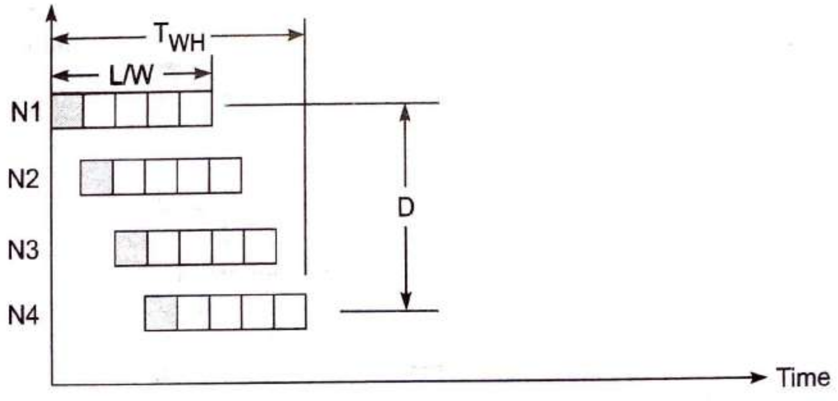
Fig. 7.28 Handshaking protocol between two wormhole routers (Courtesy of Lionel Ni, 1991)

Asynchronous pipelining can be very efficient, and the clock used can be faster than that used in a synchronous pipeline. However, the pipeline can be stalled if flit buffers or successive channels along the path are not available during certain cycles. Should that happen, the packet can be buffered, blocked, dragged, or detoured. We will discuss these flow control methods in Section 7.4.3.

Latency Analysis A time comparison between store-and-forward and wormhole-routed networks is given in Fig. 7.29. 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).



(a) Store-and-forward routing



(a) Wormhole routing

Fig. 7.29 Time comparison between the two routing techniques

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

$$T_{SF} = \frac{L}{W}(D+1) \tag{7.5}$$

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

$$T_{WH} = \frac{L}{W} + \frac{F}{W} \times D \tag{7.6}$$

Equation 7.5 implies that T_{SF} is directly proportional to D. In Eq. 7.6, $T_{WH} = L/W$ if $L >> F$. Thus the distance D has a negligible effect on the routing latency.

We have ignored the network startup latency and block time due to resource shortage (such as channels being busy or buffers being full, etc.) The channel propagation delay has also been ignored because it is much smaller than the terms in T_{SF} or T_{WH}.

According to the estimate given in Table 7.1, a typical first generation value of T_{SF} is between 2000 and 6000 μs, while a typical value of T_{WH} is 5 μs or less. Current systems employ much faster processors, data links and routers. Both the latency figures above would therefore be smaller, but wormhole routing would still have much lower latency than packet store-and-forward routing.

7.4.2 Deadlock and Virtual Channels

The communication channels between nodes in a wormhole-routed multicomputer network are actually shared by many possible source and destination pairs. The sharing of a physical channel leads to the concept of virtual channels.

We introduce below the concept and explain its applications in avoiding deadlocks in this section and in facilitating network partitioning for multicasting in Section 7.4.4.

Virtual Channels A virtual channel is a logical link between two nodes. It is formed by a flit buffer in the source node, a physical channel between them, and a flit buffer in the receiver node. Figure 7.30 shows the concept of four virtual channels sharing a single physical channel.

Four flit buffers are used at the source node and receiver node, respectively. One source buffer is paired with one receiver buffer to form a virtual channel when the physical channel is allocated for the pair.

In other words, the physical channel is time-shared by all the virtual channels. Besides the buffers and channel involved, some channel states must be identified with different virtual channels. The source buffers hold flits awaiting use of the channel. The receiver buffers hold flits just transmitted over the channel. The channel (wires or fibers) provides a communication medium between them.

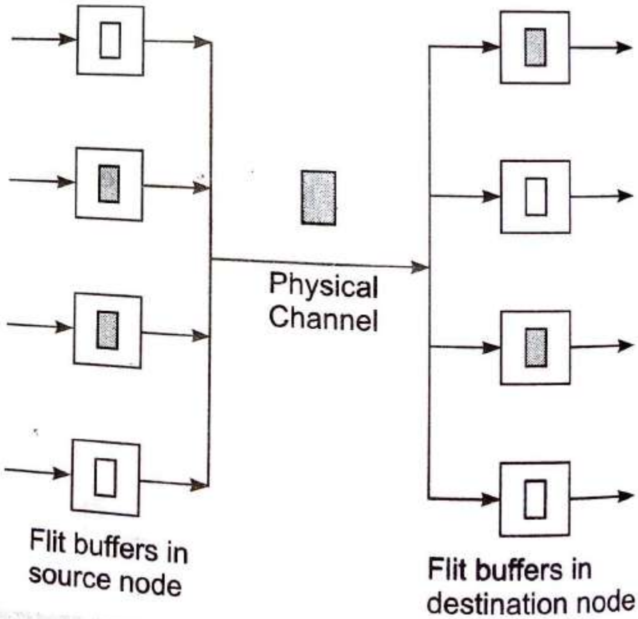
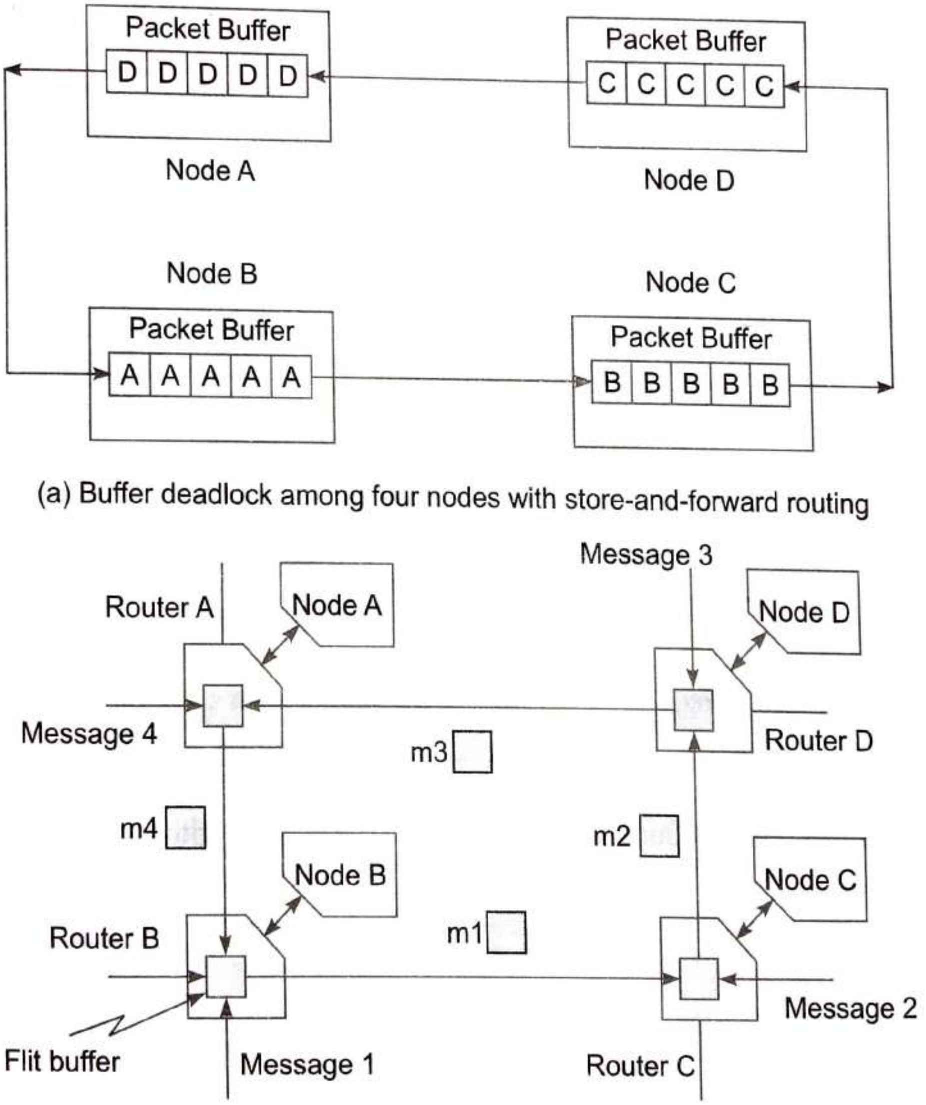


Fig. 7.30 Four virtual channels sharing a physical channel with time multiplexing on a flit-by-flit basis

Comparing the setup in Fig. 7.30 with that in Fig. 7.28, the difference lies in the added buffers at both ends. The sharing of a physical channel by a set of virtual channels is conducted by time-multiplexing on flit-by-flit basis.

Example 7.3 The deadlock situations caused by circular waits at buffers or at channels

As illustrated in Fig. 7.31, two types of deadlock situations are caused by a circular wait at buffers or channels. A *buffer deadlock* is shown in Fig. 7.31a for a store-and-forward network. A circular wait situation results from four packets occupying four buffers in four nodes. Unless one packet is discarded or misrouted, the deadlock cannot be broken. In Fig. 7.31b, a *channel deadlock* results from four messages being simultaneously transmitted along four channels in a mesh-connected network using wormhole routing.



(a) Buffer deadlock among four nodes with store-and-forward routing

(b) Channel deadlock among four nodes with wormhole routing; shaded boxes are flit buffers

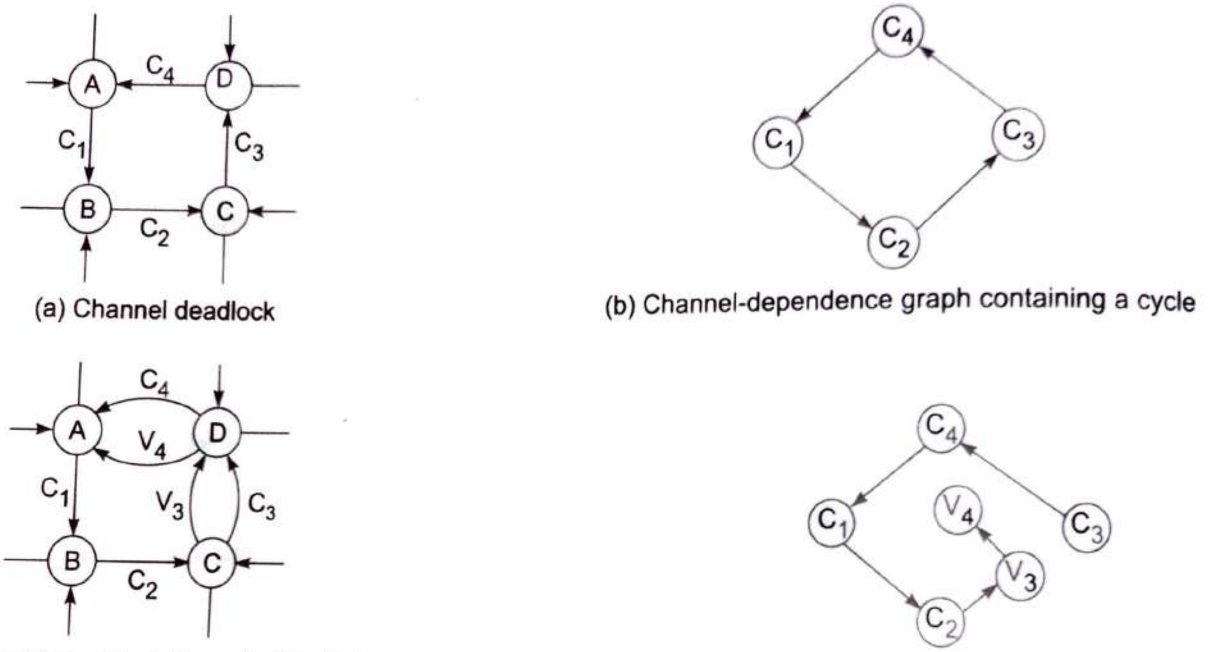
Fig. 7.31 Deadlock situations caused by a circular wait at buffers or at communication channels

Four flits from four messages occupy the four channels simultaneously. If none of the channels in the cycle is freed, the deadlock situation will continue. Circular waits are further illustrated in Fig. 7.32 using a *channel-dependence graph*.

The channels involved are represented by nodes, and directed arrows are used to show the dependence relations among them. A deadlock avoidance scheme is presented using virtual channels.

Deadlock Avoidance By adding two virtual channels, V_3 and V_4 in Fig. 7.32c, one can break the deadlock cycle. A modified channel-dependence graph is obtained by using the virtual channels V_3 and V_4, after the use of channel C_2, instead of reusing C_3 and C_4.

The cycle in Fig. 7.32b is being converted to a spiral, thus avoiding a deadlock. Channel multiplexing can be done at the flit level or at the packet level if the packet length is sufficiently short. Virtual channels can be implemented with either *unidirectional channels* or *bidirectional channels*.



(a) Channel deadlock

(b) Channel-dependence graph containing a cycle

(c) Adding two virtual channels (V_3, V_4)

(d) A modified channel-dependence graph using the virtual channels

Fig. 7.32 Deadlock avoidance using virtual channels to convert a cycle to a spiral on a channel-dependence graph

The use of virtual channels may reduce the effective channel bandwidth available to each request. There exists a tradeoff between network throughput and communication latency in determining the degree of using virtual channels. High-speed multiplexing is required for implementing a large number of virtual channels.

7.4.3 Flow Control Strategies

In this section, we examine various strategies developed to control smooth network traffic flow without causing congestion or deadlock situations. When two or more packets collide at a node when competing for buffer or channel resources, policies must be set regarding how to resolve the conflict.

Based on these policies, we describe below deterministic and adaptive routing algorithms developed for one-to-one i.e. unicast communication.

Packet Collision Resolution In order to move a flit between adjacent nodes in a pipeline of channels, three elements must be present: (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, they may request the same receiver buffer or the same outgoing channel. Two arbitration decisions must be made: (i) Which packet will be allocated the channel? and (ii) What will be done with the packet being denied the channel? These decisions lead to the four methods illustrated in Fig. 7.33 for coping with the packet collision problem.

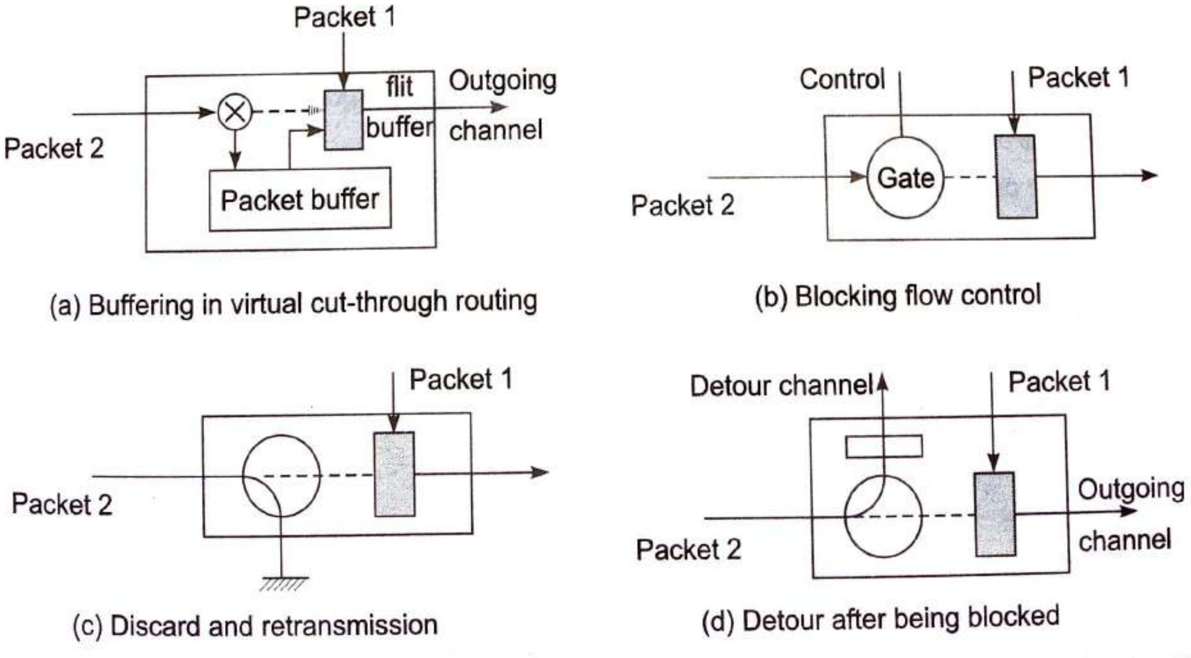
Figure 7.33 illustrates four methods for resolving the conflict between two packets competing for the use of the same outgoing channel at an intermediate node. Packet 1 is being allocated the channel, and packet 2 being denied. A *buffering* method has been proposed with the *virtual cut-through routing* scheme devised by Kermani and Kleinrock (1979).

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. However, it requires the use of a large buffer to hold the entire packet.

Furthermore, the packet buffers along the communication path should not form a cycle as shown in Fig. 7.31a. The packet buffer however may cause significant storage delay. The virtual cut-through method offers a compromise by combining the store-and-forward and wormhole routing schemes. When collisions do not occur, the scheme should perform as well as wormhole routing. In the worst case, it will behave like a store-and-forward network.

Pure wormhole routing uses a blocking policy in case of packet collision, as illustrated in Fig. 7.33b. The second packet is being blocked from advancing; however, it is not being abandoned. Figure 7.33c shows the *discard* policy, which simply drops the packet being blocked from passing through.

The fourth policy is called *detour* (Fig. 7.33d). The blocked packet is routed to a detour channel. The blocking policy is economical to implement but may result in the idling of resources allocated to the blocked packet.



(a) Buffering in virtual cut-through routing

(b) Blocking flow control

(c) Discard and retransmission

(d) Detour after being blocked

Fig. 7.33 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)

The discard policy may result in a severe waste of resources, and it demands packet retransmission and acknowledgment. Otherwise, a packet may be lost after discarding. This policy is rarely used now because of its unstable packet delivery rate. The BBN Butterfly network had used this discard policy.

Detour routing offers more flexibility in packet routing. However, the detour may waste more channel resources than necessary to reach the destination. Furthermore, a re-routed packet may enter a cycle of *livelock*, which wastes network resources. Both the Connection Machine and the Denelcor HEP had used this detour policy.

In practice, some multicomputer networks use hybrid policies which may combine the advantages of some of the above flow control policies.

Dimension-Order Routing Packet routing can be conducted deterministically or adaptively. In *deterministic routing*, the communication path is completely determined by the source and destination addresses. In other words, the routing path is uniquely predetermined in advance, independent of network condition.

Adaptive routing may depend on network conditions, and alternate paths are possible. In both types of routing, deadlock-free algorithms are desired. Two such deterministic routing algorithms are given below, based on a concept called *dimension order routing*.

Dimension-order routing requires the selection of successive channels to follow a specific order based on the dimensions of a multidimensional network. 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. For hypercube (or n-cube) networks, the scheme is called *E-cube routing* as originally proposed by Sullivan and Bashkow (1977). These two routing algorithms are described below by presenting examples.

E-cube Routing on Hypercube Consider an n-cube with $N = 2^n$ nodes. Each node b is binary-coded as $b = b_{n-1}b_{n-2} \dots b_1b_0$. Thus the source node is $s = s_{n-1} \dots s_1s_0$ and the destination node is $d = d_{n-1} \dots d_1d_0$. We want to determine a route from s to d with a minimum number of steps.

We denote the n dimensions as $i = 1, 2, \dots, n$, where the ith dimension corresponds to the $(i-1)$st bit in the node address. Let $v = v_{n-1} \dots v_1v_0$ be any node along the route. The route is uniquely determined as follows:

1. Compute the direction bit $r_i = s_{i-1} \oplus d_{i-1}$ for all n dimensions ($i = 1, \dots, n$). 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 $i \leq n$, go to step 2, else done.

Example 7.4 E-cube routing on a four-dimensional hypercube

The above E-cube routing algorithm is illustrated with the example in Fig. 7.34. Now $n = 4$, $s = 0110$, and $d = 1101$. Thus $r = r_4r_3r_2r_1 = 1011$. 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 $r_4 = 1$.

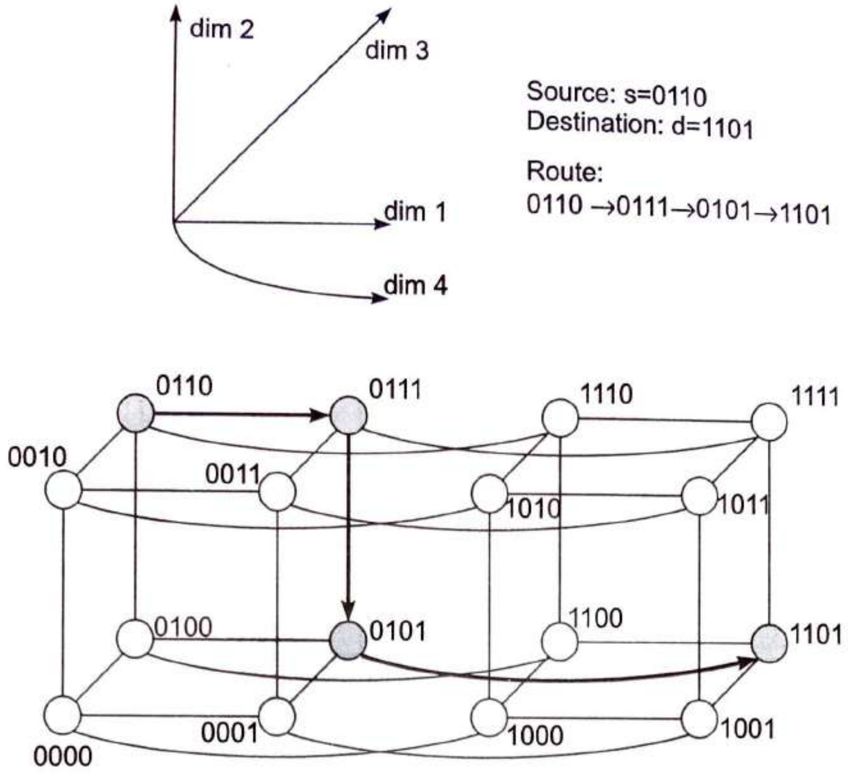


Fig. 7.34 E-cube routing on a hypercube computer with 16 nodes

The route selected is shown in Fig. 7.34 by arrows. Note that the 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.

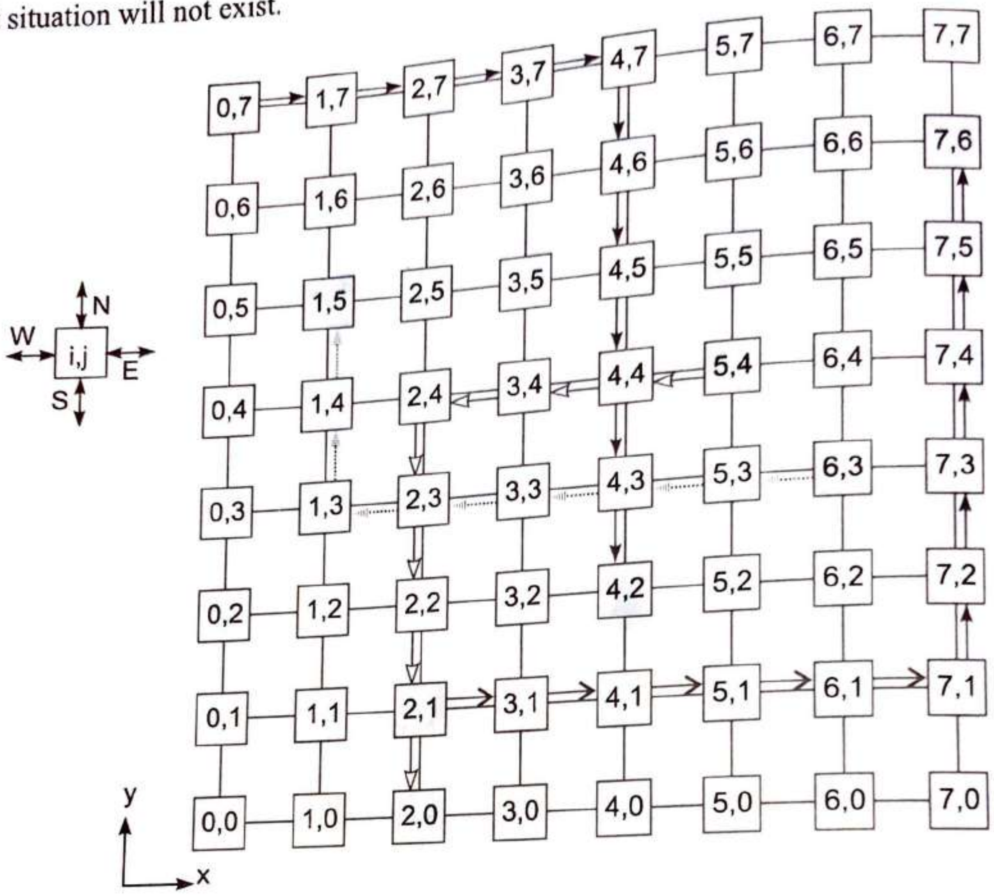
X-Y Routing on a 2D Mesh The same idea is applicable to mesh-connected networks. X-Y routing is illustrated by the example in Fig. 7.35. From any source node $s = (x_1y_1)$ to any destination node $d = (x_2y_2)$, route from s along the X-axis first until it reaches the column Y_2, where d is located. 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 7.5 X-Y routing on a 2D mesh-connected multicomputer

Four (source, destination) pairs are shown in Fig. 7.35 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 the Y-dimension, a deadlock or circular wait situation will not exist.



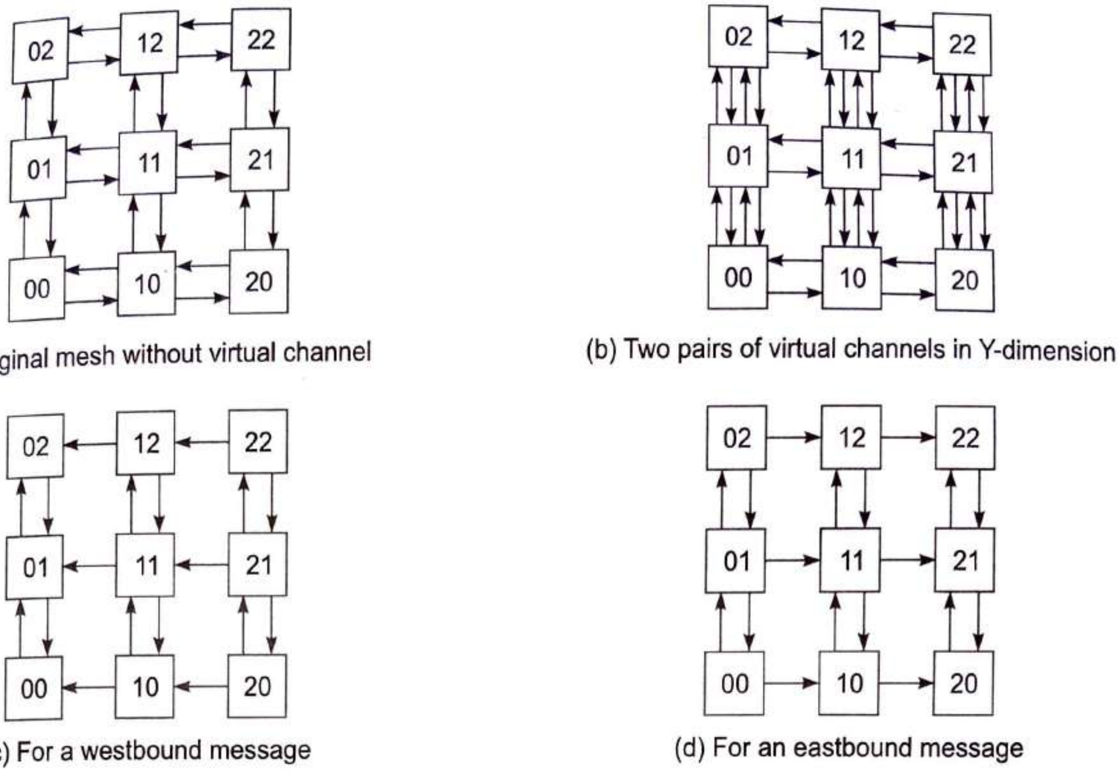
Four (source; destination) pairs: (2,1;7,6)⟶ (0,7;4,2)⟶ (5,4;2,0)⟶ (6,3;1,5)- - -

Fig. 7.35 X-Y routing on a 2D mesh computer with 8 × 8 = 64 nodes

It is left as an exercise for the reader to prove that both E-cube and X-Y schemes result in deadlock-free routing. Both can be applied in either store-and-forward or wormhole-routed networks, resulting in a minimal route with the shortest distance between source and destination.

However, the same dimension order routing scheme cannot produce minimal routes for torus networks. Nonminimal routing algorithms, producing deadlock-free routes, allow packets to traverse through longer paths, sometimes to reduce network traffic or for other reasons.

Adaptive Routing The main purpose of using adaptive routing is to achieve efficiency and avoid deadlock. The concept of virtual channels makes adaptive routing more economical and feasible to implement. We have shown in Fig. 7.32 how to apply virtual channels for this purpose. The idea can be further extended by having virtual channels in all connections along the same dimension of a mesh-connected network (Fig. 7.36).



(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

Fig. 7.36 Adaptive X-Y routing using virtual channels to avoid deadlock; only westbound and eastbound traffic are deadlock-free (Courtesy of Lionel Ni, 1991)

Example 7.6 Adaptive X-Y routing using virtual channels

This example uses two pairs of virtual channels in the Y-dimension of a mesh using X-Y routing.

For westbound traffic, the *virtual network* in Fig. 7.36c can be used to avoid deadlock because all eastbound X-channels are not in use. Similarly, the virtual network in Fig. 7.36d 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 deadlockfree multicast routing in the next section.

7.4.4 Multicast Routing Algorithms

Various communication patterns are specified below. Routing efficiency is defined. The concept of virtual networks and network partitioning are applied to realize the complex communication patterns with efficiency.

Communication Patterns Four types of communication patterns may appear in multicomputer networks. What we have implemented in previous sections is the one-to-one unicast pattern with one source and one destination.

A *multicast* pattern corresponds to one-to-many communication in which one source sends the same message to multiple destinations.

A *broadcast* pattern corresponds to the case of one-to-all communication. The most generalized pattern is the many-to-many *conference* communication.

In what follows, we consider the requirements for implementing multicast, broadcast, and conference communication patterns. Of course, all patterns can be implemented with multiple unicasts sequentially, or even simultaneously if resource conflicts can be avoided. Special routing schemes must be used to implement these multi-destination patterns.

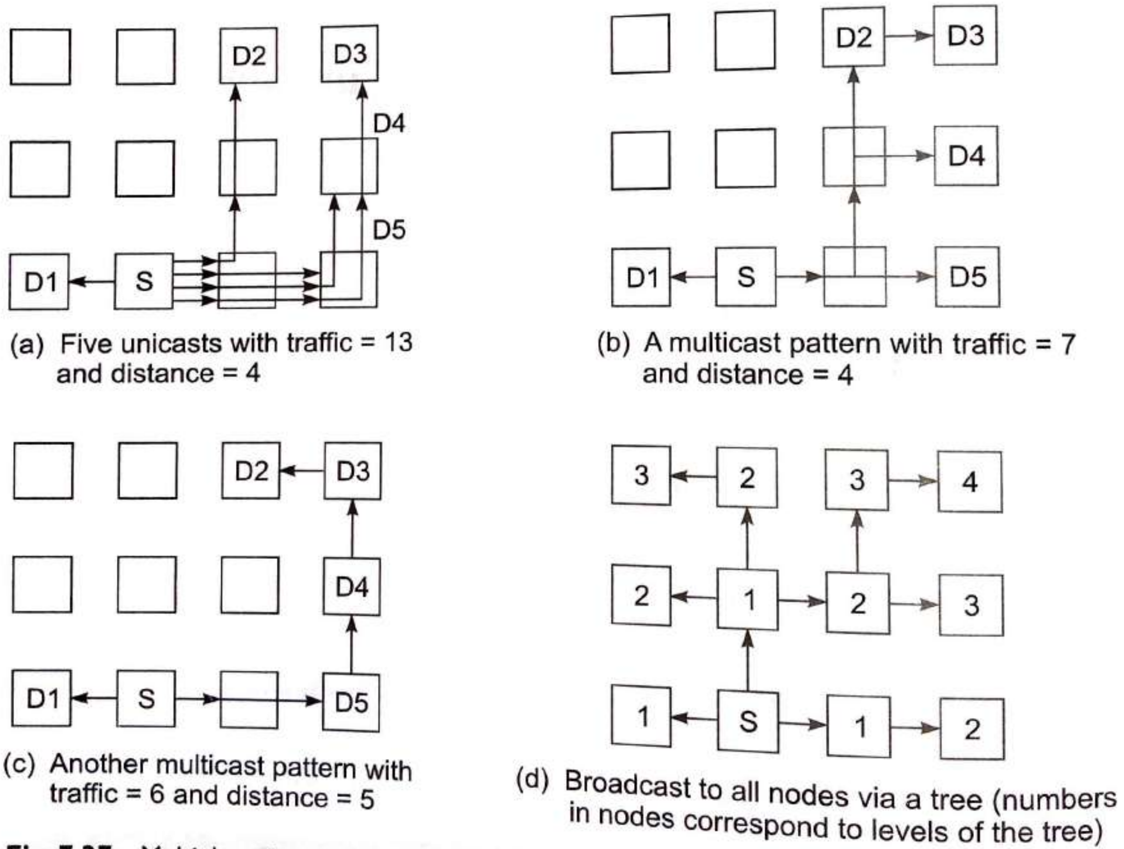
Routing Efficiency Two commonly used efficiency parameters are *channel bandwidth* and *communication latency*. The channel bandwidth at any time instant (or during any time period) indicates the effective data transmission rate achieved to deliver the messages. The latency is indicated by the packet transmission delay involved.

An optimally routed network should achieve both maximum bandwidth and minimum latency for the communication patterns involved. However, these two parameters are not totally independent. Achieving maximum bandwidth may not necessarily achieve minimum latency at the same time, and vice versa.

Depending on the switching technology used, latency is the more important issue in a store-and-forward network, while in general the bandwidth affects efficiency more in a wormhole-routed network.

Example 7.7 Multicast and broadcast on a mesh-connected computer

Multicast routing is implemented on a 3 × 3 mesh in Fig. 7.37. The source node is identified as S, which transmits a packet to five destinations labeled D_i for $i = 1, 2, ..., 5$.



(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)

Fig. 7.37 Multiple unicasts, multicast patterns, and a broadcast tree on a 3 × 4 mesh computer

This five-destination multicast can be implemented by five unicasts, as shown in Fig. 7.37a. 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.

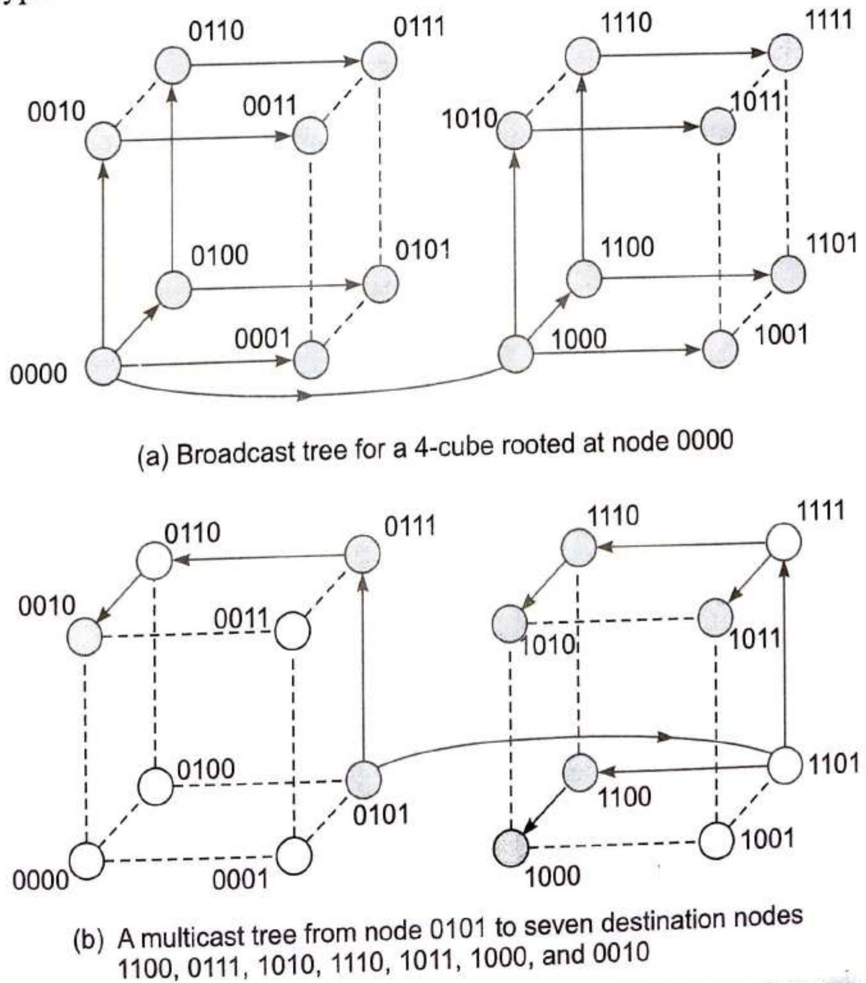
A multicast can be implemented by replicating the packet at an intermediate node, and multiple copies of the packet reach their destinations with significantly reduced channel traffic.

Two multicast routes are given in Figs. 7.37b and 7.37c, resulting in traffic of 7 and 6, respectively. On a wormhole-routed network, the multicast route in Fig. 7.37c is better. For a store-and-forward network, the route in Fig. 7.37b 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 Fig. 7.37d. 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 7.8 Multicast and broadcast on a hypercube computer

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 Fig. 7.38a for a 4-cube rooted at node 0000. Again, minimum traffic should result with a broadcast tree for a hypercube.



(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

Fig. 7.38 Broadcast tree and multicast tree on a 4-cube using a greedy algorithm (Lan, Esfahnian, and Ni, 1990)

A greedy multicast tree is shown in Fig. 7.38b 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.

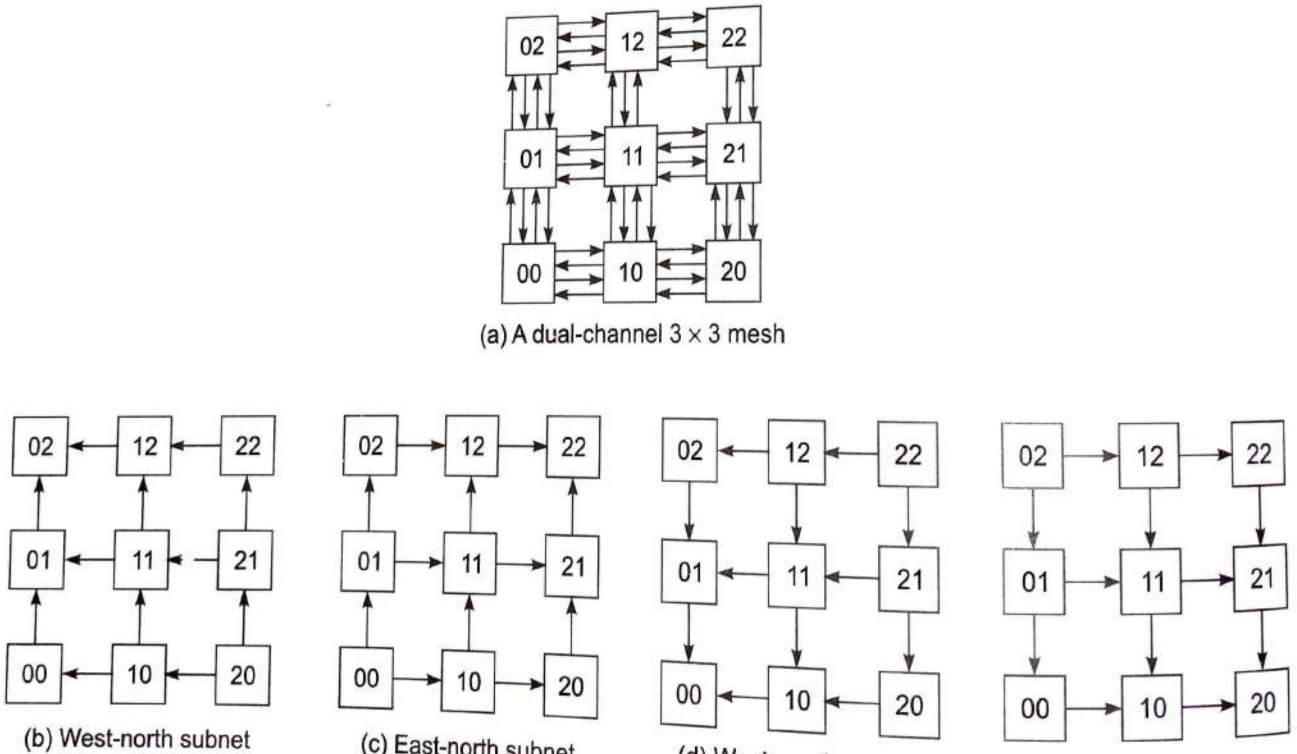
Extending the multicast tree, one should compare the reachability via all dimensions before selecting certain dimensions to obtain a minimum cover set for the remaining nodes. In case of a tie between two dimensions, selecting any one of them is sufficient. Therefore, the tree may not be uniquely generated.

It has been proved that this greedy multicast algorithm requires the least number of traffic channels compared with multiple unicasts or a broadcast tree. To implement multicast operations on wormhole-routed networks, the router in each node should be able to replicate the data in the flit buffer.

In order to synchronize the growth of a multicast tree or a broadcast tree, all outgoing channels at the same level of the tree must be ready before transmission can be pushed one level down. Otherwise, additional buffering is needed at intermediate nodes.

Virtual Networks Consider a mesh with dual virtual channels along both dimensions as shown in Fig. 7.39a.

These virtual channels can be used to generate four possible virtual networks. For west-north traffic, the virtual network in Fig. 7.39b should be used.



(a) A dual-channel 3 × 3 mesh

(b) West-north subnet

(c) East-north subnet

(d) West-south subnet

(e) East-south subnet

Fig. 7.39 Four virtual networks implementable from a dual-channel mesh

Similarly, one can construct three other virtual nets for other traffic orientations. Note that no cycle is possible on any of the virtual networks. Thus deadlock can be completely avoided when X-Y routing is implemented on these networks.

If both pairs between adjacent nodes are physical channels, then any two of the four virtual networks can be simultaneously used without conflict. 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.

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.

Obviously, adding channels to the network will increase the adaptivity in making routing decisions. However, the increased cost can be appreciable and thus prevent the use of redundancy.

Network Partitioning 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 Fig. 7.40.

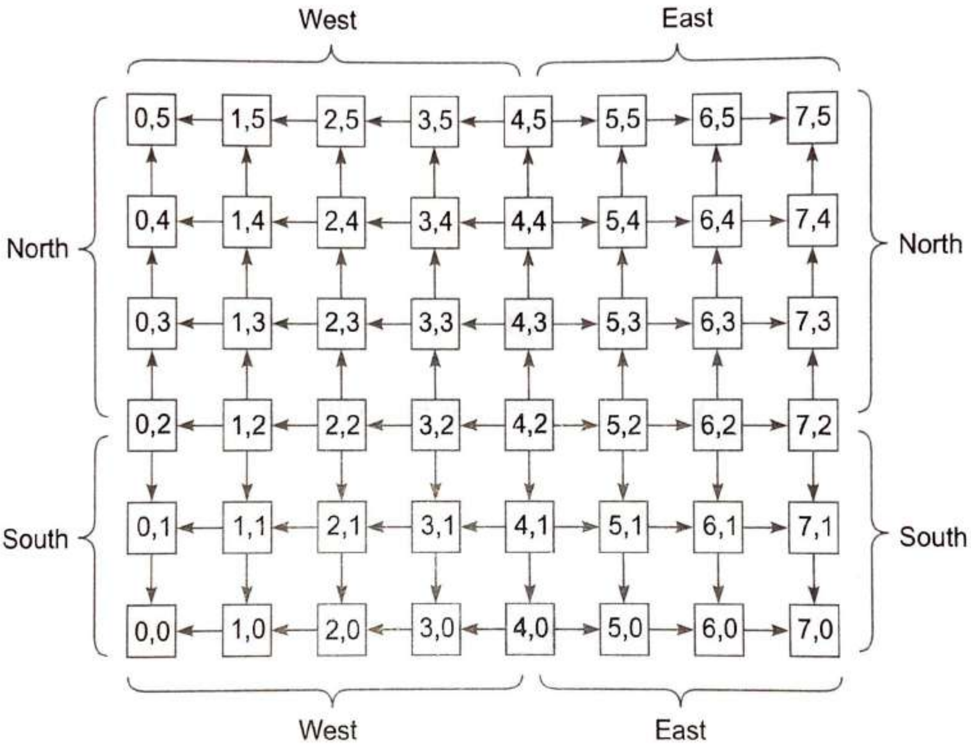


Fig. 7.40 Partitioning of a 6 × 8 mesh into four subnets for a multicast from source node **(4,2)**. Shaded nodes are along the boundary of adjacent subnets (Courtesy of Lin, McKinly, and Ni, 1991)

Suppose source node (4, 2) wants to transmit to a subset of nodes in the 6×8 mesh. The mesh is partitioned into four logical subnets. 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.

Nodes in the fifth column and third row are along the boundary between subnets. Essentially, the traffic is being directed outward from the center node (4, 2). There is no deadlock if an X-Y multicast is performed in this partitioned mesh.

Similarly, one can partition a binary n-cube into 2^{n-1} subcubes to provide deadlock-free adaptive routing. Each subcube has $n + 1$ levels with 2^n virtual channels per level for the bidirectional network. The number

7.4 MULTIPROCESSOR OPERATING SYSTEMS

In this section, we discuss the operating system requirements for multiprocessors. First, a classification of multiprocessor operating systems is presented. We then discuss other system software supports needed for multiprocessing.

7.4.1 Classification of Multiprocessor Operating Systems

There is conceptually little difference between the operating system requirements of a multiprocessor and those of a large computer system utilizing multiprogramming. However, there is the additional complexity in the operating system when multiple processors must work simultaneously. This complexity is also a result of the operating system being able to support multiple asynchronous tasks which execute concurrently.

The functional capabilities which are often required in an operating system for a multiprogrammed computer include the resource allocation and management schemes, memory and dataset protection, prevention of system deadlocks and abnormal process termination or exception handling. In addition to these capabilities, multiprocessor systems also need techniques for efficient utilization of resources and, hence, must provide input-output and processor load-balancing schemes. One of the main reasons for using a multiprocessor system is to provide some effective reliability and graceful degradation in the event of failure. Hence, the operating system must also be capable of providing system reconfiguration schemes to support graceful degradation. These extra capabilities and the nature of the multiprocessor execution environment places a much heavier burden on the operating system to support automatically the exploitation of parallelism in the hardware and the programs being executed. An operating system which performs poorly will negate other advantages which are associated with multiprocessing. Hence it is of utmost importance that the operating system for a multiprocessing computer be designed efficiently.

The presence of more than one processing unit in the system introduces a new dimension into the design of the operating system. The influence of a large number of processors on the design of an operating system is still a research problem. The modularity of processors and the interconnection structure among them affect the system development. Furthermore, communication schemes, synchronization mechanisms, and placement and assignment policies dominate the efficiency of the operating system. We introduce below only the basic configurations that have appeared in existing multiprocessor systems.

There are basically three organizations that have been utilized in the design of operating systems for multiprocessors, namely, *master-slave configuration*, *separate supervisor* for each processor, and *floating supervisor control*. For most multiprocessors, the first operating system available assumed the master-slave mode. This mode, in which the supervisor is always run on the same processor, is certainly the simplest to implement. Furthermore, it may often be designed by making relatively simple extensions to a uniprocessor operating system that includes full multiprogramming capabilities. Although the master-slave type of system is simple, it is normally quite inefficient in its control and utilization of the total system resources. The other two operating modes are superior to the master-slave in performance.

In a *master-slave* mode, one processor, called the master, maintains the status of all processors in the system and apportions the work to all the slave processors. An example of the master-slave mode is in the Cyber-170, where the operating is executed by one peripheral processor P_0. All the other processors (central and peripheral) are treated as slaves to P_0. Another example is found in the DEC ...10, in which there are two identical processors. One of the processors is designated as master and the other as slave. The operating system runs only on the master, with the slave treated as a schedulable resource.

Since the supervisor routine is always executed in the same processor, a slave ...quest via a trap or supervisor call instruction for an executive service must be ... to the master, which acknowledges the request and performs the appropriate ...vice. The supervisor and its associated procedures need not be reentrant ... there is only one processor that uses them. There are other characteristics ... the master-slave operating system. Table conflicts and lock-out problems for ...tem control tables are simplified by forcing a single processor to run the executive. However, this operating system mode causes the entire system to be very susceptible to catastrophic failures which require operation intervention to restart ... master processor when an irrecoverable error occurs. In addition to the ...flexibility of the overall system, the utilization of the slave processors may become appreciably low if the master cannot dispatch processes fast enough to keep the ...es busy. The master-slave mode is most effective for special applications where the work load is well defined or for asymmetrical systems in which the slaves have less capability than the master processor. It is the mode sometimes used if there are very few processors involved.

When there is a *separate supervisor system* (kernel) running in each processor, the operating system characteristics are very different from the master-slave systems. This is similar to the approach taken by computer networks, where each processor contains a copy of a basic kernel. Resource sharing occurs at a higher level, for example, via a shared file structure. Each processor services its own needs. However, since there is some interaction between the processors, it is necessary for some of the supervisory code to be reentrant or replicated to provide separate copies for each processor. Although each supervisor has its own set of private tables, some tables are common and shared by the whole system. This creates table access problems. The method used in accessing the shared resources depends on the degree of coupling among the processors. The separate supervisor operating system is not as sensitive to a catastrophic failure as a master-slave system. Also, each processor has its own set of input-output devices and files, and any reconfiguration of I/O usually requires manual intervention and possibly manual switching.

Unfortunately, the replication of the kernel in the processors would demand a lot of memory which may be underutilized, especially when compared with the utilization of the shared data structures. A static form of caching could be used to buffer frequently used portions of the operating system code, while the infrequently used code could be centralized in a shared memory. Unfortunately, the determination of which portions of operating system are frequently executed is relatively difficult to make and is likely to be dependent of the application workload.

The *floating supervisor control* scheme treats all the processors as well as other resources symmetrically or as an anonymous pool of resources. This is the most difficult mode of operation and the most flexible. In this mode, the supervisor routine floats from one processor to another, although several of the proce be executing supervisory service routines simultaneously. This type of sys attain better load balancing over all types of resources. Conflicts in service are resolved by priorities that are either set statistically or under dynamic Since there is a considerable degree of sharing, most of the supervisory code reentrant. In this system, table-access conflicts and table lock-out delays can avoided. It is important to control these accesses in such a way that system in is protected. This mode of operation has the advantages of providing gr degradation and better availability of a reduced capacity system. Furtherm provides true redundancy and makes the most efficient use of available res Examples of operating systems that execute in this mode are the MVS and the IBM 3081 and the Hydra on the C.mmp.

Most operating systems, however, are not pure examples of any of the classes discussed above. The only generalization that is possible is that th system produced is usually of the master-slave type and the ultimate being so is the fioating supervisor control. In Table 7.3, we summarize the major ch teristics, advantages, and shortcomings of the above three types of oper systems for multiprocessor computers.

7.4.2 Software Requirements for Multiprocessors

One of the issues often raised in a discussion on multiprocessor software i question of how it differs from uniprocessor software. In particular, how d software written to execute on multiple processors differ from that written execute on the more familiar multiprogrammed uniprocessor environme There are basically two sources of differences. These are the architectural attrib that are unique to the multiprocessor, and a new programming style peculiar parallel applications. Such differences would warrant that the hardware a software of the system should provide facilities that are different from those fo in conventional multiprogrammed uniprocessor environments. A multip grammed uniprocessor can simulate the multiple processor environment creating multiple "virtual processors" for the users. For example, a Unix u routinely requests the concurrent execution of multiple programs with the outp of one program "piped" as the input to the other. In this case, each program m be thought of as executing on a virtual processor. At this level of program executi there are few differences between a multiprogrammed uniprocessor system and multiprocessor system. However, the presence of multiple processors and oth replicated components usually increases the amount of management softw that must be provided.

An architectural attribute that may affect programming in a multiprocess system is nonhomogeneity. If the central processors are nonhomogeneous, that functionally different, they must be treated differently by software. For examp if one processor possesses emulation capability not possessed by another, so programs can only run to completion on the processor with the emulation cap

Table ·3 Operating system configurations for a multiprocessor computer

Master-slave operating system

1. The executive routine is always executed in the same processor. If the slave needs service that must be provided by the supervisor, then it must request that service and wait until the current program in the master processor is interrupted and the supervisor is dispatched. The supervisor and the routines that it uses do not have to be reentrant since there is only the one processor using them.
2. Having a single processor executing the supervisor simplifies the table conflict and lock-out problem in control tables. The overall system is comparatively inflexible. This type of system requires comparatively simple software and hardware.
3. The entire system is subject to catastrophic failures that require operator intervention to restart when the processor designated as the master has a failure or irrecoverable error.
4. Idle time on the slave system can build up and become quite appreciable if the master cannot execute the dispatching routines fast enough to keep the slave(s) busy.
5. This type of operating system is most effective for special applications where the work load is well defined or for asymmetrical systems in which the slaves have less capability than the master processor.

Separate supervisor in each processor.

1. Each processor services it own needs. In effect, each processor (supervisor) has its own set of I/O equipment, files, etc.
2. It is necessary for some of the supervisory code to be reentrant or replicated to provide separate copies for each processor.
3. Each processor (actually each supervisor) has its own set of private tables, although some tables must be common to the entire system, and that creates table-access control problems.
4. The separate supervisor operating system is as sensitive as is the master-slave system; however, the restart of an individual processor that has failed will probably be quite difficult.
5. Because of the point immediately above, the reconfiguration of I/O usually requires manual intervention and possibly manual switching.

Floating-supervisor operating system.

1. The "master" floats from one processor to another, although several of the processors may be executing supervisor service routines at the same time.
2. This type of system can attain better load balancing over all types of resources.
3. Conflicts in service requests are resolved by priorities that can be set statically or under dynamic control.
4. Most of the supervisory code must be reentrant since several processors can execute the same service routine at the same time.
5. Table-access conflicts and table lock-out delays can occur, but there is no way to avoid this with multiple supervisors; the important point is that they must be controlled in such a way that system integrity is protected.

bility. Hence, software resource managers must provide appropriate dispatching mechanisms for such programs. Another example of software complexity occurs in a system with asymmetric main memory. In this case, not all processors can access all memory. This complicates the operating system software for resource management.

There is a second potential source of difference between multiprocessor and uniprocessor software. This is in the programming style peculiar to parallel applications. The basic unit of a program in execution is that of a process, an independent schedulable entity (a sequential program) that runs a processor and uses hardware and software resources. It may also execute concurrently with other processes, delayed (at least logically) only when it needs to wait to inter one or more other processes. Hence, a parallel program can be said to two or more interacting processes.

The potential of multiprocessing is achieved by enhancing its capab parallel processing. Parallel processing can be indicated in a program expl implicitly. For explicit parallelism, users must be provided with progr abstractions that permit them to indicate explicit parallelism when de program. Implicit parallelism is detected by the compiler. In this case, the c scans the source program and recognizes the program flow. From this flow and other conditions, it detects nontrivial units of program statements whi be identified as a process. Some of these units may be independent and can concurrently with other processes.

In a multiprocessor system, synchronization takes on increased impo as it could create too high a penalty. This could significantly degrade system formance if the synchronization mechanisms are not efficient and the algor that use them are not properly designed. In some processors, the synchroniz primitives are not implemented directly in hardware or microcode. There software alternatives must be provided. For example, the PDP-11 processors for the C.mmp have been implemented with the semaphore-synchroniz primitive in software, thereby taking a significant number of instructions. In environment where processes need to synchronize often, this may be a m bottleneck.

Program-control structures are provided to aid the programmer in develo efficient parallel algorithms. Three basic nonsequential program-control struc have been identified. These control structures are characterized by the fact that programmer need only focus on a small program and not on the overall contr the computation. The first example is the *message-based* organization which used in the Cm* operating system. In this organization, computation is perform by multiple homogeneous processes that execute independently and interac messages. The grain size of a typical process depends on the system.

The second example of a control structure is the *chore* structure. In t structure, all codes are broken into small units. The process that executes the u of code (and the code itself) is called a *chore*. An important characteristic of a ch is that once it begins execution, it runs to completion. Hence, to avoid long wa chores are basically small. They have relatively very little input and they refere only a few different objects. Moreover, they do not block and are noninterrupti As part of its output, one chore might request the execution of a small set additional chores. Examples of systems that use this structure are the Plurib and the BCC-500.

Consider the memory-management portion of the operating system, whi controls swapping between the main memory and a fixed-head disk. Samp chores may include (*a*) the disk command to request the transfer of a page of da between the disk and the memory, and (*b*) acknowledging completion of a disk sector transmission and arranging for any subsequent action.

The third nonsequential control structure is that of *production systems*, oft often used in artificial intelligence systems. Productions are expressions of th antecedent, consequent⟩. Whenever the boolean antecedent evaluates to the consequent may be performed. In contrast to chores, production consequents may or may not include code which might block. In a production system, scheduling strategies are often required (*a*) to control the selection of antecedents to be evaluated next, (*b*) to order (if necessary) the execution of selected antecedents, (*c*) to select the subset of runnable consequents to be executed, and order (if necessary) the execution of the selected consequents. Note that by features of all three control structures, they are all compatible with parallel execution.

The high degree of concurrency in a multiprocessor can increase the complexity of fault handling, especially in the recovery step. In a uniprocessor, it is always possible to eliminate parallelism by disabling interrupts and, if necessary, halting activity. Software is needed to establish effective error recovery capability. This software, even with the aid of hardware mechanisms, may be quite complex. Understanding the behavior of running processes in a multiprocessor system is more complex than in uniprocessor environments. Although parallel programs may not be too complex to implement, there is a natural problem of nondeterminism in multiprocessors. Some efforts have been made to prove the correctness of parallel programs by researchers but extending these proofs to complex programs is still a formidable task.

4.3 Operating System Requirements

The basic goals for an operating system are to provide programmer interface (environment) to the machine, manage resources, provide mechanisms (system defined) to implement policies (user definable), and facilitate matching applications to the machine. It must also help achieve reliability. But this and other desirable attributes incur a cost that may be unacceptable. Guidelines should be established for trading performance for desirable attributes. The degree of transparency of the detailed machine that should be made available to the programmer should also be determined.

There are different levels of interaction in the specification of an operating system for multiprocessing systems. Asynchronous supervisor processes share the specification of the address-space management, process management, and synchronization levels. Efficient operating systems are designed to have a modular structure and hierarchical organization. This makes the detection and localization of errors easier. The classic functions of an operating system include the creation of objects such as processes and their domains, which include the memory segments. The management and sharing of segments, as was discussed in Chapter 2, are also important operating system functions. Other functions are the management of process communications through mailboxes or message buffers. Messages are used to define the interface between processes and help to reduce the number of ways an error can be propagated through the system.

In a multiprocessor system, processes can execute concurrently until they need to interact. Planned and controlled interaction is referred to as process communication or process synchronization. Process communication mu place through shared or global variables. Cooperating processes must com cate to synchronize or limit their concurrency. The relationship betwe cooperating processes regarding a resource falls into one of two fundam categories. They are either competitors or producers-consumers. Since p communication takes place through shared memory, competitors acces memory to seize and release permanent or *reusable* resources. Produ consumers access this memory to pass temporary or *consumable* resource as messages and signals.

In systems with multiple concurrent processes, the presence of resource as unit record peripherals and tape drives which must not be used simultane by several processes (if program operation is to be correct) introduces the rec ment for exclusive access to these devices. This requirement may also be imp on shared objects such as a data segment during updating. Processes de exclusive access to a resource may compete for it. The same competition a concerning access to what are called *virtual* resources, such as system table communication buffers between cooperating processes. Since it must be guaran that both processes do not access the buffer simultaneously, exclusive acce the buffer must be ensured. This exclusiveness of access is called *mutual exclu* between processes. A request for mutual exclusion on the use of a resource imp the desire to reserve or release the resource. Process cooperation and compet may both be implemented if a mechanism is provided for process coordinatio synchronization. This mechanism will be discussed in Section 8.1.

The above requirements for processor cooperation and competition h obvious implications on short- and medium-term scheduling of the mul processors. If a desired resource or object is not available, the process reques it must be suspended, blocked, or retry until it becomes available. There are two levels of exclusiveness. One consists of the requirements for referral of acce a data structure (virtual resource) which may often be of short duration. The o is the requirement for, perhaps, a substantial delay until the physical resou such as the processor or tape unit, becomes available. If the delay is short not worthwhile to shift the attention of the processor from the process whi running on it to another process. If the delay exceeds the time required to sw the processor, the ability to shift attention may be vital for efficient utilizatio the processor.

The sharing of the multiple processors may be achieved by placing the seve processes together in shared memory and providing a mechanism for rap switching the attention of a processor from one process to another. This operat is often called *context switching*. Sharing of the processors introduces three s ordinate problems:

1. The protection of the resources of one process from willful or accidental dama by other processes
2. The provision for communication among processes and between user proces and supervisor processes

This chapter covers interprocess synchronization mechanisms, system deadlocks, protection schemes, multiprocessor scheduling, and parallel algorithms. These are important topics in developing a sophisticated operating system for a multiprocessor system. The parallel algorithms form the basis in using MIMD computers. For other related issues that have not been covered below, readers are advised to check the attached bibliographic notes.

8.1 INTERPROCESS COMMUNICATION MECHANISMS

Various interprocess communication schemes have been proposed by computer designers. This section enumerates some of the process-synchronization mechanisms implementable at the instruction level. High-level mechanisms such as the **P** and **V** primitives and conditional critical regions are then presented. Examples are given on the producer-consumer processes and the reader-writer problem using these mechanisms. The extension of conditional critical regions to monitors is also discussed.

8.1.1 Process Synchronization Mechanisms

Cooperating processes in a multiprocessor environment must often communicate and synchronize. Execution of one process can influence the other via communication. Interprocess communication employs one of two schemes: use of shared variables or message passing. Often the processes that communicate do so via a synchronization mechanism. A process executes with unpredictable speed and generates actions or events which must be recognized by another cooperating process. The set of constraints on the ordering of these ev stitutes the set of synchronization required for the operating proc synchronization mechanism is used to delay execution of a process in satisfy such constraints.

Two types of synchronization are commonly employed when using variables. These are *mutual exclusion* and *condition synchronization*. W that mutual exclusion ensures that a physical or virtual resource is held indi Another situation occurs in a set of cooperating processes when a share object is in a state that is inappropriate for executing a given operation process which attempts such an operation should be delayed until the state data object changes to the desired value as a result of other processes executed. This type of synchronization is sometimes called condition synch ization. The mutual-exclusive execution of a critical section, S, whose ac controlled by a variable gate can be enforced by an entry protocol deno MUTEXBEGIN (gate) and an exit protocol denoted by MUTEXEND Alternatively, the effect of the entry and exit protocols can be expressed as gate **do** S.

There are certain problems associated with implementing the MUTEXBE MUTEXEND construct. Execution of the MUTEXBEGIN statement s detect the status of the critical section. If it is busy, the process attempting enter the critical section must wait. This can be done by setting an indica show that a process is currently in the critical section. Execution of the MUTE END statement should reset the status of the critical section to idle and pr a mechanism to schedule the waiting process to use the *critical section*. One implementation is the use of the LOCK and UNLOCK operations correspond to MUTEXBEGIN and MUTEXEND respectively. For consider that there is a single *gate* that each process must pass through to a CS and also leave it. If a process attempting to enter the CS finds the gate locked (open) it locks (closes) it as it enters the CS in one indivisible operation that all other processes attempting to enter the CS will find the gate locked completion, the process unlocks the gate and exits from the CS. Assuming that variable *gate* = 0(1) means that the gate is open (closed), the access to controlled by the gate can be written as

```
LOCK (gate)
execute critical section
UNLOCK (gate)
```

The LOCK (x) operation may be implemented as follows:

```
        var x: shared integer;
LOCK (x): begin
            var y: integer;
              y ← x;
            while y = 1 do y ← x; // wait until gate is open
              x ← 1; // set gate to unavailable status //
            end
```

The UNLOCK(x) operation may be implemented as

```
UNLOCK(x): x ← 0;
```

The LOCK mechanism as shown is not satisfactory because two or more processes may find $x = 0$ before one reaches the $x \leftarrow 1$ statement. This can be remedied if the processor has an instruction that both tests and sets (modifies) a word. Such an instruction, called TEST_AND_SET(x) and available on the IBM S/370, tests and sets a shared variable x in a single *read-modify-write* memory cycle to produce a variable y. The read-modify-write operation must take place in one cycle so that the memory location, x, is not accessed and modified by another processor before the current processor completes the test-and-set operation. The indivisibility is usually accomplished by the requesting processor which holds the bus until the cycle is completed. Therefore the set of operations $\{y \leftarrow x; x \leftarrow 1\}$ is indivisible in the following definition of TEST_AND_SET(x):

```
                  var x: shared integer;
TEST_AND_SET(x): begin
                      var y: integer;
                      y ← x;
                      If y = 0 then x ← 1;
                      end
```

The LOCK operation may be rewritten as

```
      var x: shared integer;
LOCK(x): begin
         var y: integer;
           Repeat {y ← TEST_AND_SET(x)} until y = 0;
         end
```

An important property of locks is that a process does not relinquish the processor on which it is executing while it is waiting for a lock held by another process. Thus, it is able to resume execution very quickly when the lock becomes available. However, this property may create problems for the error-recovery mechanism of the system when the processor which is executing the lock fails. The error-recovery procedure has to be sophisticated enough to ensure that deadlocks are not introduced as a result of the recovery process itself.

Another instruction used to enforce mutual exclusion of access to a shared variable in memory location *m*_addr is the *compare-and-swap* (CAS) instruction. This instruction is available on the IBM 370/168. A typical syntax of this instruction uses the two additional operands *r*_old and *r*_new, which are processor registers

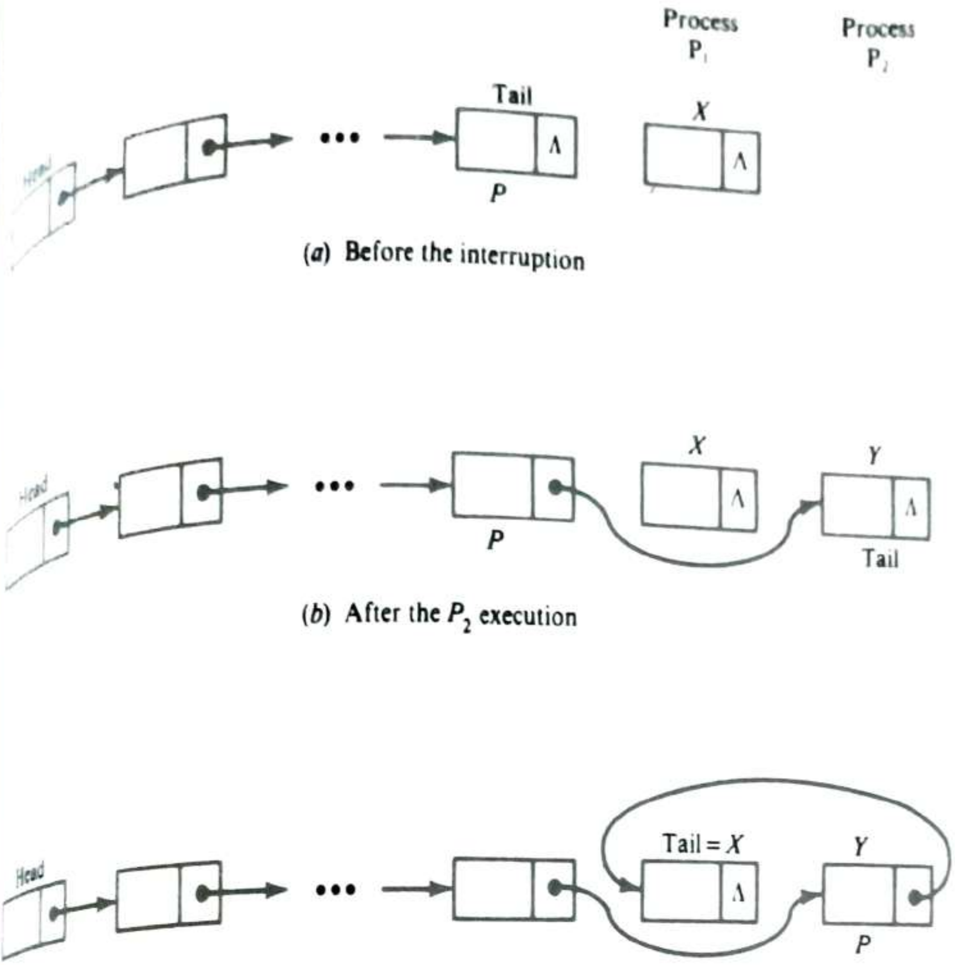
(CAS *r*_old, *r*_new, *m*_addr). The action of the CAS instruction is defined as follows

```
        var m_addr: shared address;
        var r_old, r new: registers;
        var z: CAS flag;
      CAS: if r_old = m_addr then
             {m_addr ← r_new; z ← 1}
        else
             {r_old ← m addr; z ← 0}
```

Notice that associated with the CAS instruction is a processor flag z. The flag is set if the comparison indicates equality. Again, the execution of the CAS instruction (that is, the IF statement) is an indivisible operation. We illustrate the use of the CAS instruction with a shared singly linked queue data structure (Figure 8.1) which is accessed concurrently by the two processes P_1 and P_2. The two operations which can be performed on the queue are ENQUEUE(X) and DEQUEUE. ENQUEUE(X) adds a node X to the "TAIL" of the queue and DEQUEUE returns a pointer to the deleted "HEAD" of the queue. HEAD and TAIL are shared global variables. Assuming that the queue is never empty (for simplicity), the ENQUEUE(X) primitive for a nonconcurrent system can be described as

```
Procedure ENQUEUE(X);
var P: pointer;       //P is local to each invocation //
begin
LINK(X) ← Λ;          //terminate last node's link//
P ← TAIL;
TAIL ← X;
LINK(P) ← X;          //attach new node to queue//
end
```

Suppose process P_1 requests to enqueue node X. While P_1 is executing the primitive, it gets interrupted by P_2, which requests to enqueue node Y to the same queue. Assume that the interruption occurs at the end of statement $P \leftarrow$ TAIL. Figure 8.1*a* illustrates the state of the queue at the time of interruption. If P_1 executes the procedure to completion after P_2 returns control to P_1, node X will be attached to the queue. However, node Y, which was added by P_2, would have been detached from the queue unintentionally. This error occurs because pointer P was not updated to point to the last node attached by process P_2. We can avoid this problem by using the CAS instruction to update P to point to the last attached node. This can be accomplished by replacing the TAIL ← X statement with



(*a*) Before the interruption

(*b*) After the P_2 execution

(*c*) Followed by the P_1 execution

Figure 8.1 Interleaved execution of ENQUEUE by process P_1 and process P_2.

“repeat CAS P, X, TAIL until TAIL = X”. The modified ENQUEUE(X) primitive is shown below:

```
Procedure ENQUEUE(X);
var P: pointer;
begin
  LINK(X) ← Λ;
  P ← TAIL;
  repeat CAS P,X,TAIL until TAIL = X;
  LINK(P) ← X;
  end
```

The CAS instruction ensures that the logical state (P) of the interrupted program is maintained on resumption of the interrupted program. Otherwise updates the state P to the most recent value of TAIL.

Figure 8.1*b* shows the outcome of the execution of the primitive by P_2 foll by the completion of the execution of the primitive by P_1 (Figure 8.1*c*). The instruction is more useful than the test-and-set instruction. An extension o CAS instruction is the *compare double and swap*, also available on the IBM 37 There are other variations which enforce mutual exclusion. For example Honeywell 60/66 has the *load-accumulator-and-clear-memory-location* (L instruction.

The LOCK instruction using TAS has a drawback in that processes attem to enter critical sections are busy accessing and testing common variables is called *busy-wait* or *spin-lock*, which results in performance degradation process cannot normally be context-swapped off its processor while it is wa Hence, the processor is said to be *locked out*. Such lock-out is only permitte supervisor mode. In general, LOCK and UNLOCK primitives are not us allowed to be executed in user mode because the user process may be swap out while holding a critical section. On the other hand, if the user makes a su visor call each time it attempts to access a critical section, the overhead w greatly increased. Hence the CAS instruction was provided as an excellent mec ism of letting the user do some synchronization in user mode.

The performance degradation due to spin-locks is two-fold. When a proce is spinning, it actively consumes memory bandwidth that might otherwise h been used more constructively. If the spinning period is too long, a processo not effectively utilized during that period. A number of methods have been propo to reduce the degradation due to spin-locks. The first method is aimed at redu the request rate to memory and, hence, the degree of memory conflicts. Th accomplished by delaying the reissuance of the lock request for an interval Thus, the LOCK(x) primitive, for example, can be modified as

```
LOCK(x): begin
           y ← TEST-AND-SET(x);
           while y ≠ 0 do
             begin
               PAUSE(T)  //
               y ← TEST-AND-SET(x);
             end
           end
```

Note that the processor issuing the request may not be released unless *T* is l enough. The choice for *T* depends on the granularity of the resource bein quested.

The second method is directed at relieving the processor of performing lock access by incorporating a separate mechanism which processes lock reque This can be accomplished in one of several ways. For example, the mech can continuously access the lock until it is available, as in the HEP mac concurrently, the processor can execute another ready-to-run process in its local memory. When the processor is signaled by the mechanism that the lock has been allocated, it immediately resumes execution of the waiting process. While resumption is immediate because the process was not swapped out. The busy wait can be avoided if, in the first access to the lock, it is found busy. In this case, the process requesting it is blocked. When the lock becomes available, the mechanism is signaled so that the blocked process can be readied to resume execution. The latter scheme seems adequate for a mutually exclusive access to a resource with large granularity.

The distribution of locks in memory is an important factor in the performance of concurrent processes accessing lockable resources. For example, if all locks are stored in one memory module, the contention for these locks can become excessive. In a multiprocessor with private caches, the accesses to locks by the processors can cause excessive overhead because of consistency checks. However, contention for these locks can be partly relieved by distributing the locks into many blocks of memory.

Two primitive operations can be defined to *block* a process attempting to enter a busy critical section and *wake_up* the blocked process when the critical section becomes free. These primitives are

wake-up (p): if process p is logically blocked (that is, dormant), change its state to active; else set up a *wake-up waiting switch* (wws) to remember the wake-up call

block (p): if process p's wws is set, reset it and continue execution of the process; else change p's state to dormant

Using these operations requires a wake-up list which is updated dynamically. In order to prevent loss of information, wake-up signals that occur while a process is executing must be saved and a process should not be allowed to become dormant until all its wake-up signals have been serviced. The wake-up waiting switch (wws) is the mechanism used to save wake-up signals. A process identification tag is appended to the wake-up signal, which is used to route the signal to the appropriate receiving process. Hardware or software mechanisms may be used to implement the wws, which stores the wake-up signals on arrival until they are acknowledged. This may result in a potential race condition if the mechanism is improperly designed. Note that the blocking and unblocking operations constitute an overhead which may be significant if designed improperly.

Lock conflicts are resolved in the implementation of the busy-wait because a request that finds the lock busy waits until the lock is released. Serialization is thus enforced. Another synchronization primitive was proposed to permit some form of concurrency of access to a memory location while still enforcing some serialization. The format of this primitive is *fetch-and-add* (X, e), where X is a shared integer variable and e is an integer expression. Let the value of X be denoted by Y. We abbreviate the primitive as $\mathbf{F\&A}(X, e)$, which is defined to return the old value of X and replace the contents of X by the sum $Y + e$ in one indivisible operation. If several fetch-and-add operations are initiated simultaneo different processors on the shared variable X, the effect of these opera exactly what it would be if they occurred in some unspecified serial orde is, X is modified by the appropriate total increment and each operation the intermediate value of X corresponding to its position in the serial As an example, consider the two processors P_i and P_j which issue:

$$S_i \leftarrow \mathbf{F\&A}(X, e_i),\ S_j \leftarrow \mathbf{F\&A}(X, e_j)$$

respectively. Then S_i and S_j may contain Y and $Y + e_i$, respectively, or S_j may contain $Y + e_j$ and Y, respectively, depending on the priorities of P_j and the order of arrival of their requests. In either case, X becomes $Y +$

The fetch-and-add primitive can be implemented within the processor me switch, as shown in Figure 8.2 for the example with two simultaneous fetch- adds directed at the same memory location X. In the example, it is assumed P_i's request has a higher priority than P_j's. The switch forms $e_i + e_j$ and tran $\mathbf{F\&A}(X, e_i + e_j)$ to the memory. At the same time, e_i is stored in the sw register. On receipt of Y from memory as a result of $\mathbf{F\&A}(X, e_i + e_j)$, the sw transmits Y and $Y + e_i$ in response to requests $\mathbf{F\&A}(X, e_i)$ and F&A(respectively.

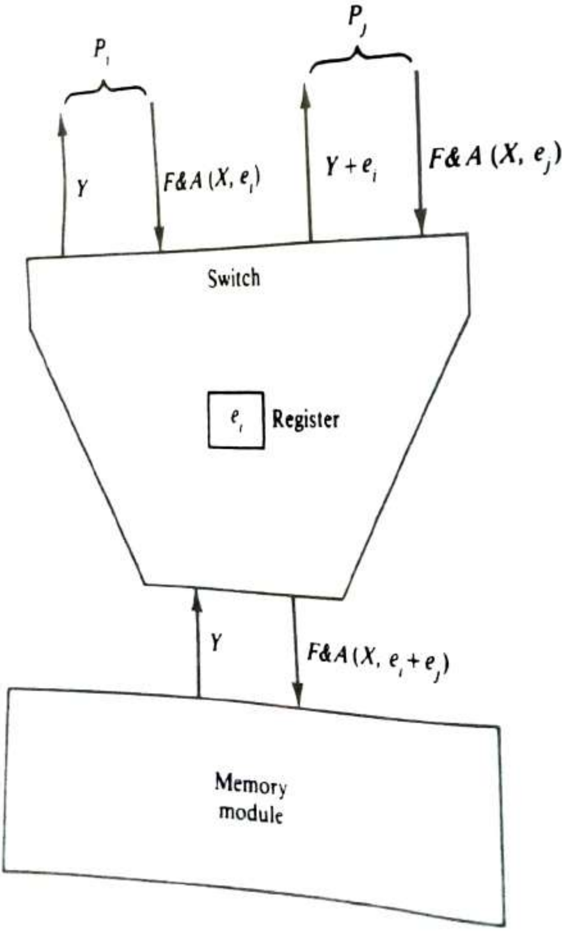


Figure 8.2 Implementatio *Fetch-and-Add* primitive (

We illustrate a simple application of **F&A** in a multiprocessor environment. Suppose it is required that all processors which intend to access a given resource must first indicate their intentions by incrementing a common counter, X, in memory. An accurate count of these requests can be maintained if each processor indicates its intention by the statement **F&A**(X, 1). Hence if two or more processors execute the statement simultaneously, X will contain the correct count of requests at the completion of all instructions. Note that the value returned to each processor as a result of execution of **F&A**(X, 1) can be used as its position in the request queue.

Another synchronization primitive uses the semaphore, which consists of a counter, a process routine queue, and the two functions **P** and **V**. This is described in the next subsection. Although simple, semaphores are known to be sufficient solutions to synchronization problems for permanent-resource competitors and temporary-resource producers-consumers. However, semaphores are often very inconvenient in representing communication between processes. For this reason, most operating systems also provide other process-communication mechanisms. Two examples are events and messages.

Event primitives are typically provided by the two functions *wait* and *signal*. A process can wait on an event or a combination of events to be true. When another process signals an event, all processes waiting on that event are placed on the ready queue. Other variations are also possible. One potential problem with events is that a process has the possibility of waiting on an event that either never becomes true or was signaled earlier. A slight variation of waiting on an event, used especially in real-time systems, is waiting on a timing queue administered by the operating system for a specified time period to elapse.

Messages provide an even more flexible and direct method of interprocess communication, especially for producer-consumer relationships. Typical primitives are the functions *send* and *receive*, which allow a string of characters to be passed between processes. Implementation variations are numerous. For example, send may or may not wait for an acknowledgement. Receive usually waits if no message has been sent. The Intel iAPX 432 multiprocessor system uses the send and receive primitives.

8.1.2 Synchronization with Semaphores

Dijkstra invented the two operations **P** and **V**, which can be shared by many processes and which implement the mutual-exclusion mechanism efficiently. The P and V operations are called primitives and are assumed indivisible. They operate on a special common variable called a *semaphore*, which indicates the number of processes attempting to use the critical section:

```
var s: semaphore
```

Then the primitive **P**(s) acts as an open bracket or MUTEXBEGIN of a critical section, that is, it acts to acquire permission to enter. The $V(s)$ primitive is MUTEXEND and records the termination of a critical section:

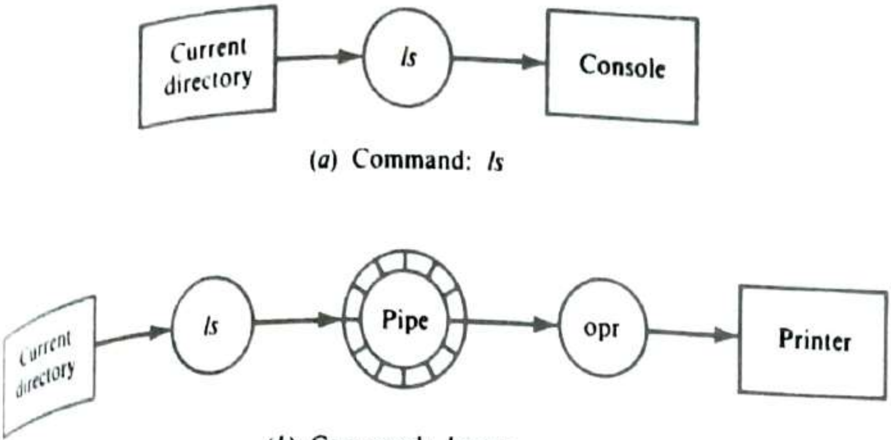
```
P(s) MUTEXBEGIN (s)
     s ← s − 1;
     If s < 0 then
     begin
     Block the process executing the P(s) and put it
     in a FIFO queue associated with the semaphore s;
     Resume the highest priority ready-to-run process;
     end
     MUTEXEND
V(s) MUTEXBEGIN (s)
     s ← s + 1;
     If s ≤ 0 then
     begin
     If an inactive process associated with semaphore s exists, then
     wake up the highest priority blocked process associated with s
     and put it in a ready list.
     end
     MUTEXEND
```

The semaphore s is usually initialized to 1. When s can take values of 0 or 1, it is called a *binary semaphore*, since it acts as a lock bit, allowing only one process at a time within an associated critical section. If s takes any integer value, it is called a *counting semaphore*. Notice that the **P**(s) and **V**(s) operations are modifying and testing its status. P(s) and V(s) can be implemented in hardware or in software using locks.

One common use of synchronization mechanisms is to permit concurrent processes to exchange data during execution. The data or messages to be exchanged are usually stored in a circular buffer which is used to synchronize the speeds of the sending and receiving processes. Such a circular buffer is usually called a message buffer or *mailbox*.

For example, the Unix operating system provides an elegant form of message buffers called *pipes*. These are used as channels to stream data from one process to another. The typing of the "ls" command on the console in a Unix environment causes the files in the current directory to be "listed" on the console by running the "ls" process. If the user wishes to print the listing of the files on the printer, the two concurrent processes "ls" and "opr" may be used and are specified as "ls | opr." The "|" symbol specifies that a pipe should channel the output of "ls" to become the input to "opr." The "ls" process *produces* the list as an output to a pipe or buffer from which it is *consumed* and printed by the "opr" process, as illustrated in Figure 8.3.

Whenever a process produces sequences of output which are consumed by another process as input, there is said to be a producer-consumer relationship



(a) Command: ls

(b) Command: ls | opr

Figure 8.3 Flow of data between two processes in UNIX.

message buffer may be considered to consist of a finite number of identical slots which are used for communication between the producer and consumer processes. If the number of slots is finite, the buffer is arranged as a circular buffer.

To demonstrate the communication between the producer and consumer processes, consider a finite buffer BUFFER of size n arranged as a circular queue in which the slot positions are named $0, 1, \ldots, n-1$. There are the two pointers c and p, which correspond to the "head" and "tail" of a circular queue, respectively, as shown in Figure 8.4. The consumei consumes the message from the head c by updating c and then retrieving the message. Hence, c points to an empty slot before each consumption. The producer adds a message to the buffer by updating p before the add operation. Therefore, pointers p and c move counterclockwise and there can be a maximum of n message slots for consumption. Initially, $p = c = 0$, which indicates that the buffer is empty. Let the variables *empty* and *full* be used to indicate the number of empty slots and occupied slots, respectively. The *empty* variable is used to inform the producer of the number of available slots, while the

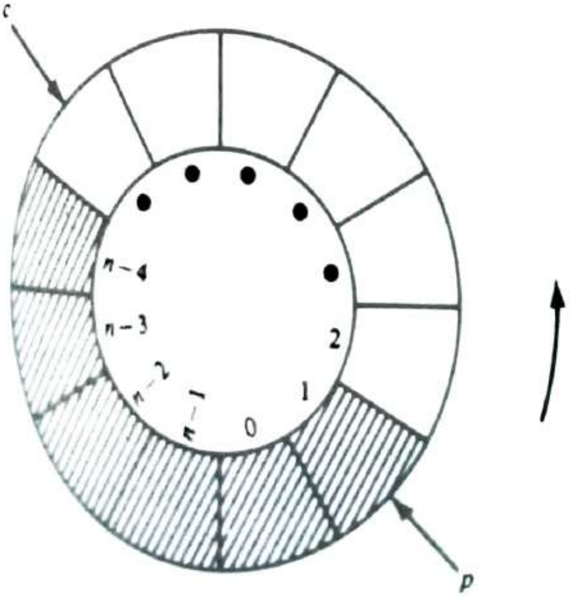


Figure 8.4 A circular message buffer with producer pointer p and consumer pointer c.

full variable informs the consumer of the number of messages needed to be consumed. The concurrent program below illustrates the actions of the producer and consumer processes. The producer or consumer will be suspended when *empty* = 0 or *full* = 0, respectively.

Example 8.1

```
    shared record
           begin
           var p, c: integer;
           var empty, full: semaphore;
           var BUFFER [0:n − 1]: message;
            nd
    initial empty = n, full = 0, p = 0, c = 0;
  cobegin
  Producer: begin
             var m: message;
               Cycle
                 begin
                   Produce a message m;
                   P(empty);
                   p ← (p + 1) mod n;
                   BUFFER [p] ← m; // place message in buffer//
                   V(full)
                 end
             end
  Consumer: begin
             var m: message;
               Cycle
                 begin
                   P (full);
                   c ← (c + 1) mod n;
                   m ← BUFFER [c]; // remove message from buffe
                   V (empty);
                   Consume message m;
                 end
               end
  coend
```

The **P** and **V** operations may be extended for ease of problem formulation and clarity of solutions. The extended primitives **PE** and **VE** developed by Agerwala (1977) are indivisible and each operates on a set of semaphores which must be initialized to nonnegative values.

```
PE($s_1$, $s_2$, ..., $s_n$, $\bar{s}_{n+1}$, ..., $\bar{s}_{n+m}$):
  MUTEXBEGIN
  if for all i, $1 \le i \le n$, $s_i > 0$ and for all j, $1 \le j \le m$, $s_{n+j} = 0$
    then for all i, $1 \le i \le n$, $s_i \leftarrow s_i - 1$
    else the process is blocked and put in a set of queues associated
         with the set of semaphores $s_1, \ldots, s_n$;
  MUTEXEND
VE($s_1$, $s_2$, ..., $s_n$):
  MUTEXBEGIN
  for all i, $1 \le i \le n$, $s_i \leftarrow s_i + 1$;
    wake-up highest priority process
    associated with set of semaphores ($s_1, \ldots, s_n$);
  MUTEXEND
```

There is no association between s_i and $\bar{s}_j$. The $\bar{s}_j$ symbol is used for convenience to represent the semaphore s_j where $j > n$. The following examples are used to illustrate the application of the extended primitives.

Example 8.2: *N* processes, equal priority, *m* resources Each of N processes requires exclusive access to a subset of m distinct resources. The processes are granted access without any consideration of priorities. If two processes use disjoint subsets of resources, they may execute simultaneously. The solution is given below:

```
              var $r_1, r_2, \ldots, r_m$: semaphore
            initial $r_1 = r_2 = \cdots = r_m = 1$;
Process i:  begin
              PE ($r_a, r_b, \ldots, r_x$);
              Use resource a, b, ..., x;
              VE ($r_a, r_b, \ldots, r_x$)
            end
```

Semaphore r_i is associated with resource i. If the PE primitive is completed successfully by a process, it indicates that resources $a, b, \ldots, x$ are available and hence are allocated to the process.

Consider the application of this example in which processes X, Y, Z compete for card reader R, printer P, and tape unit T, as shown in Figure 8.5. Each process requires two of the resources simultaneously: X requires R and P, Y requires R and T, and Z requires P and T. The “Dining Philosophers’ Problem” can be expressed as a special case of the above example (see Problems 8.6 and 8.7).

Example 8.3: *N* processes, *N* priorities, one resource Process i has higher priority than process $i + 1$, for $1 \le i \le N - 1$. The processes request access to a resource and are allocated the resource in a mutually exclusive manner

```
        var $r_R$, $r_P$, $r_T$: semaphore;
        initial $r_R = r_P = r_T = 1$;

        cobegin
Process X: begin PE($r_R$, $r_P$); use resource R and P; VE($r_R$, $r_P$); end
Process Y: begin PE($r_R$, $r_T$); use resource R and T; VE($r_R$, $r_T$); end
Process Z: begin PE($r_P$, $r_T$); use resource P and T; VE($r_P$, $r_T$); end

        coend
```

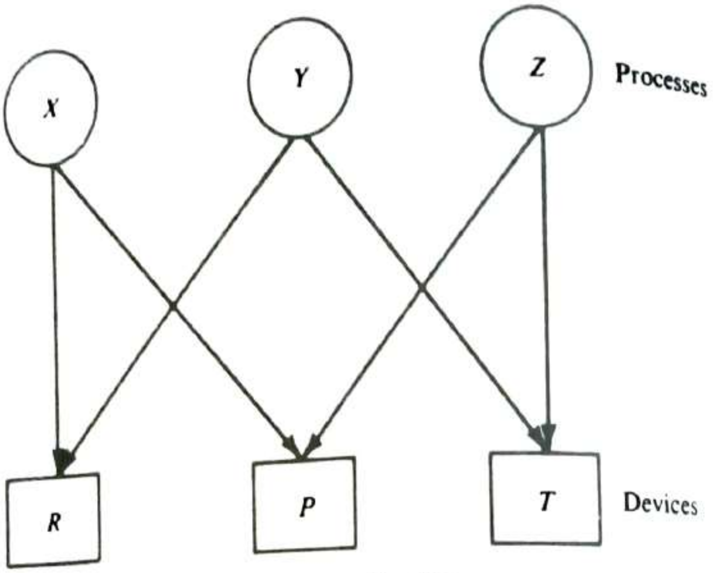


Figure 8.5 Example of multiple resource allocation.

based on the priorities. A request by a process is not honored until all higher priority requests have been granted. The resource is used nonpreemptively

```
        var $s_1, \ldots, s_n$, R: semaphore
        initial $s_1 = s_2 = \cdots = s_n = 0$;
        initial R = 1;
    Process i; begin
            VE ($s_i$); // register request of process i //
            PE (R,$\bar{s}_1, \ldots, \bar{s}_{i-1}$); // check to see if resource is available
                    // and if there are any outstanding requests //
                    // made by higher priority processes //
            PE ($s_i$); // if not, grant resource to process i //
                    // and withdraw outstanding request //
            Use resource;
            VE (R); // Return or deallocate resource //
            end
```

Note that the requesting process cannot be blocked on **PE**(s_i) since a **VE**(s_i) was executed earlier to register request. This example may be used in servicing prioritized interrupts. In this case the processes represent the interrupts and R represents the processor which services the interrupts.

The above example can be clarified further by considering a two-processor system that runs a supervisor process and a user process. If these two processes compete for a certain resource in the system *simultaneously*, the supervisor process should be given higher priority. The program segments of the supervisor and user processes that access the shared resource are shown below:

```
var    s, u, R: semaphore
initial s = u = 0;
initial R = 1;
Supervisor: begin                  User: begin
                VE(s);                       VE(u);
                PE(R);                       PE(R, s);
                PE(s);                       PE (u);
                Use Resource;                Use Resource;
                VE(R);                       VE(R);
                end                          end
```

Notice that the constructs differ mainly in the second **PE** statements. Since the supervisor process is of a higher priority than the user process, it only checks to see if the resource is available [**PE**(R)], whereas the user process also checks to see if there is an outstanding request from the supervisor [**PE**(R, s)]. Since we are considering simultaneous execution of the user and supervisor codes, the execution of the **PE**(R, s) statement will find $s = 1$, which was set by the **VE**(s) operation in the supervisor process. Hence, the user process will be blocked until the resource is released by the supervisor.

Although, semaphores can be implemented using locks, they are more commonly accessed by system calls to the supervisor. The supervisor maintains two sets of lists or queues: *blocked* and *ready*. Descriptors for processes that are blocked on a semaphore are added to a block queue associated with that semaphore. For the generalized **P** and **V**, the set of blocked queues may be quite complex. However, execution of a **PE** or **VE** operation causes a trap to a supervisor routine which completes the operation. The ready list contains descriptors of processes that are ready to be assigned to a processor for execution. In a multiprocessor, with master-slave operating system, a single processor may be responsible for maintaining the ready list.and assigning processes to the slave processors. The ready list may be shared in a multiprocessor with a distributed supervisor. In this case, the ready list may be accessed concurrently. Therefore, mutual exclusion must be ensured and can be accomplished by spin-locks, since enqueue and dequeue operations are fast on the ready list. Moreover, a processor that is attempting to access the ready list cannot execute any other process.

Semaphores are quite general and can be used to program almost any kin synchronization. However, the use of the P and V primitives in a parallel algori makes the algorithm rather unstructured and prone to error. For example, omi a P or V, or accidentally invoking a P on one semaphore and a V on another have disastrous effects, since mutual exclusion would no longer be ensu Also, when using semaphores, a programmer can forget to include in criti sections all statements that reference the shared modifiable objects. This could cause errors in execution. Another problem with using semaphores is t both condition synchronization and mutual exclusion are implemented using t same pair of primitives. This makes it difficult to identify the purpose of a gi P or V operation without a detailed trace of other effects on the semaphore.

8.1.3 Conditional Critical Sections and Monitors

Conditional critical section (CCS) was proposed by Hoare (1972) and Hanse (1972) to overcome most of the difficulties encountered with **P** and Vs. This is structured and highly user-oriented tool for specifying communication amon concurrent processes. Their use allows direct expression of the fact that a proce has to wait until an arbitrary condition on the shared variables holds. Interproce communication in a system of concurrent processes is done by means of a shared variable v, which is composed of the component variables $v_1, v_2, \ldots, v_n$, as defined by:

$$\textbf{var } v: \textbf{shared record } v_1, \ldots, v_n: \langle \text{type} \rangle \textbf{ end}$$

The variable v is used to name a given resource. The global state of a system of processes is determined by the values of the shared variable v and the program counters of the single processes. The variables in v may only be accessed within CCS statements that name v. A CCS statement is of the form

$$\textbf{csect } v \textbf{ do await } C: S$$

where C is a boolean expression and S is a statement list. Note that variables local to the executing process may also appear in the CCS statement:

A CCS statement delays the executing process until the condition, C, is true. S is then executed. The evaluation of C and execution of S are uninterruptible by other CCS statements that name the same resource. Thus C is guaranteed to be true when execution of S begins. Mutual exclusion is provided by guaranteeing that execution of different CCS statements, each naming the same resource but not overlapped. Condition synchronization is provided by explicit boolean conditions in CCS statements.

We illustrate the use of the conditional critical sections by two applications. The first example is a solution to the producer-consumer problem. Assume that the two classes of processes (producers and consumers) communicate via a bounded circular buffer as in Figure 8.4. Access to this buffer must be mutually exclusive. Seven shared variables which are associated with the critical sections are used to indicate the global status of the system of processes.

Example 8.4 The variables *p* and *c* are as in Example 8.1. Variables *empty* and *full* are also integer variables denoting the number of slots empty or occupied respectively. Variables *np* and *nc* indicate the number of producers and consumers respectively, which are working on the buffer.

```
var v: shared record
              begin
              var p, c, empty, full, np, nc: integer;
              var BUFFER [0: n - 1]: message;
              end
initial empty = n, full = 0, p = 0, c = 0;
Procedure Enqueue (m: message)
  begin
    csect v do await empty > 0 and np = 0:
        begin
          np ← np + 1;
          empty ← empty - 1;
        end
    p ← (p + 1) mod n;
    BUFFER [p] ← m;
    csect v do full ← full + 1;
  end
Procedure Dequeue (m: message)
  begin
    csect v do await full > 0 and nc = 0:
        begin
          nc ← nc + 1;
          full ← full - 1;
        end
    c ← (c + 1) mod n;
    m ← BUFFER [c];
    csect v do empty ← empty + 1;
end
```

The second example on the use of the conditional critical section is the solution of the reader-and-writer problem. Improper reading and writing of shared variables is the classic cause of difficulty in finding operating system bugs. The basic problem is that two sets of processes executing concurrently may interleave read and write operations in such a way that improper decisions are made and the shared variables are left in an improper state. This kind of bug is insidious, for it may only show up infrequently—and then the symptoms occur rarely or never repeat since they depend on a particular concurrency relationship.

In the reader-and-writer problem, there are reader and writer processes which share a common data segment. Any number of readers may access the segment simultaneously, but a writer must have exclusive access to it. To prevent a writer from waiting indefinitely long, it is necessary that no more readers be allowed to acquire the resource from the moment that a writer first wants to acquire it until the time it actually does acquire it. The variable *aw* indicates the number of writers that want to acquire the resource; *nw* and *nr* indicate the number of writers and readers, respectively, that have acquired the resource.

Example 8.5

```
        var v shared record aw, nw, nr   : integer end
        Initial aw = nw = nr = 0;
    Reader: begin
            csect v do await aw = 0: nr ← nr + 1;
            read segment;
            csect v do nr ← nr − 1;
            end
    Writer: begin
            csect v do
              begin
              aw ← aw + 1;
              await nr = 0 and nw = 0: nw ← nw + 1;
              end
              write to segment;
              csect v do begin
                nw ← nw − 1;
                aw ← aw − 1;
                end
              end
          end
```

All the process-synchronization methods we have discussed are logically equivalent in performing the synchronization or scheduling problem. However, some of them implement the solution to certain problems in a more complicated and inefficient manner than others. Therefore, they are not practically equivalent.

Monitors—*extension of conditional critical sections* A monitor is a shared data structure and a set of functions that access the data structure to control the synchronization of concurrent processes. This general definition includes semaphores, events, and messages as specific implementations. The notion of a monitor is not more powerful than these other techniques—just more general. While a process is a useful abstraction for multiprogramming, a monitor is a useful abstraction for process communication. Consequently, a programmer can ignore the implementation details of the resource when using it and can ignore how it is used when programming the monitor that implements it.

To assure the correctness of a program, it is useful to associate data structures with the operations performed on them. A monitor provides a body in which

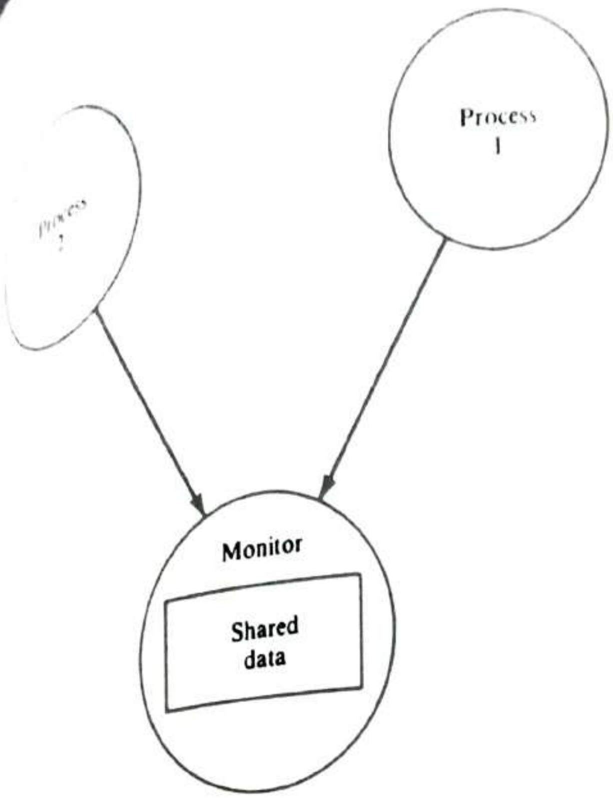


Figure 8.6 **Monitor representation.**

associate shared data structures with their critical sections. By so doing, the data structures are no longer shared or global, but local or hidden within the body of a monitor. In addition, process functions no longer contain critical sections. Instead, the critical sections are centralized and protected within the monitor functions. The restricted access to shared data structures provided by a monitor is even more attractive if it can be checked by a compiler. Many high-level languages today provide the means for controlling the scope of variable names.

Monitors provide support for processes to form a multiprogramming system. While a process is active in the sense that it performs a job, a monitor is passive in the sense that it only executes when called by a process. A monitor is necessary only when two or more processes communicate to ensure that they communicate properly. Figure 8.6 is a representation of two processes communicating through shared data encapsulated by a monitor.

A monitor consists of a set of *permanent variables* used to store the resource's state, and some procedures, which implement operations on the resource. A monitor also has initialization code for the permanent variables. This code is executed once before any procedure body is executed. The values of the permanent variables are retained between activations of monitor procedures and may be accessed only from within the monitor. Monitor procedures can have parameters and local variables, each of which takes on new values for each procedure activation. The structure of a monitor with name *mname* and procedures OP1, . . . , OPN is shown below.

```
mname monitor;
   var declarations of permanent variables
   procedure OP1 (parameters)
      var declarations of variables local to OP1
      begin
         code to implement OP1
      end
        ⋮
      procedure OPN (parameters)
      var declarations of variables local to OPN
      begin
         code to implement OPN
      end
   begin
      code to initialize permanent variables
   end
```

The procedure OPJ within monitor *mname* can be invoked by executing

call *mname* · OPJ (arguments).

The execution of the procedures in a given monitor is guaranteed to be mu exclusive. This ensures that the permanent variables are never accessed c currently.

Pushing the monitor concept to its logical limit suggests that systems sho be designed as collections of processes and monitors only. In this case, every d *structure* is local to either a process or monitor. This decomposition is valuab large systems since it simplifies the problems of program validation and m tenance. If a data structure changes, it is clear which functions are affected, the addition of a new process or monitor does not require the revalidation unchanged components.

The current state of a monitor is defined by the monitor image, that is memory associated with the monitor program. A monitor image repres either permanent or temporary resources that are the elements of process in action. A process image is that portion of memory belonging to a process defining its states. The process image changes with the execution of a progr associated with the process. In the absence of process activity, process ima and monitor images differ significantly. In this idle state, process images are no importance and may vanish. However, monitor images—at least those resenting permanent resources – must remain and resume a nonassigned state

Monitor functions are reentrant but contain nonreentrant sections— critical sections. Indeed, monitor functions must be designed to protect against th Monitor functions need not be considered as part of the monitor image. In fa if two different monitor variables are accessed in the same way, a single copy o program may be shared between the two monitors.

object

8.3 MULTIPROCESSOR SCHEDULING STRATEGIES

In this section, we discuss the processor management techniques used processor systems. The introduction of multiple processors complica scheduling problem. Deterministic and probabilistic models have been evaluate some scheduling schemes. Generally, finding an optimal algori the processor scheduling problem in multiprocessors is computationally able. However, some dynamic-scheduling algorithms are close to optimal

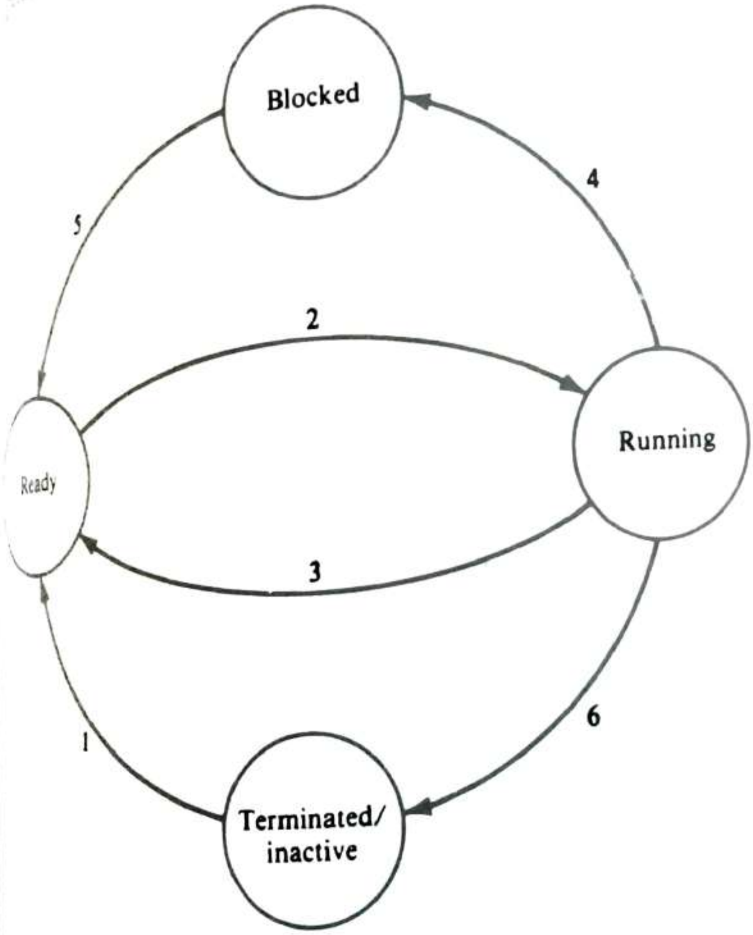
8.3.1 Dimensions of Multiple Processor Management

Multiprocessor management and scheduling have been a fertile source of esting problems for researchers in the field of computer engineering. Rep components, particularly those that are nonhomogeneous (heterogene asymmetric, increase the amount of management that must be provided by the operating system or the application or both. In its most general form problem involves the scheduling of a set of processes on a set of processors arbitrary characteristics in order to optimize some objective function involves the selection of a process for execution from a set of processes.

Basically, there are two resource-allocation decisions that are made in processing systems. One is where to locate code and data in physical memory *placement decision*; and the other is on which processor to execute each process *assignment decision*. These decisions are often trivial for a uniprocessor system which assignments are dictated. Furthermore, the physical memory address is accessible to the single processor in a uniprocessor system, hence the ques of accessibility never occurs and memory-contention problems can be minimi by interleaving. Oftentimes, assignment decision is called processor managem It describes the managing of the processor as a shared resource among users and internal processes. As a result, processor management consists basic kinds of scheduling: long-term external load scheduling and short internal process scheduling.

In general, active processes undergo different state transitions in the cours their lifetimes in the system. A process is in the *run* state if it is using a proces A suspended process may enter the pool of *blocked processes*. A process is blo if it cannot run because it is waiting for some external response, such as a wak signal, which may arrive to unblock it. The unblocking operation changes status of the process to a *ready-to-run* or *ready* state, where it is eventually schedu ssor. Figure 8.12 illustrates the possible state transition experienced by a The scheduler at this level performs the *short-term process-scheduling* n of selecting a process from the set of ready-to-run processes. The selected s assigned to run on a processor. The *medium* and *long-term load-scheduling* n is used to select and activate a new process to enter the processing nment. The activation of the new process causes it to be put on the ready ng-term load-scheduling also acts to control the degree of multiprocessing the number of active processes) in the system which, if excessive, may *thrashing* (see Chapter 2).

The process scheduler or dispatcher performs its function each time the running ss is blocked or preempted. Its purpose is to select the next running process the set of ready queues. The process scheduler resides in the kernel and can nsidered a monitor for the ready queues. Since it is probably the most frely executed program in th system, it should be fairly efficient to minimize ting system overhead.



Transition	Event
1	Activate process
2	Run process
3	Preempt process
4	Block process
5	Wakeup process
6	Terminate process

8.12 States of a process and their state transitions.

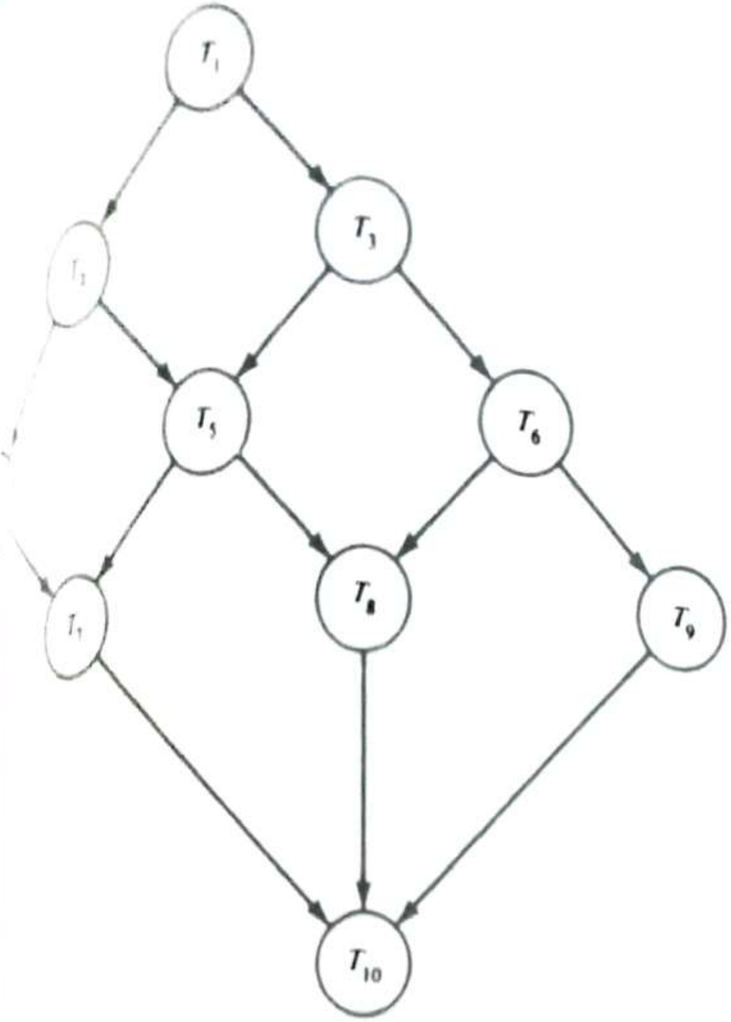
The ready list can either be local or global. A local ready list may be asso with each multiprogrammed processor which has a local memory. Thus a pr once activated, may be bound to a processor. The local ready list reduc access time of the list and, hence, the overhead encountered by the dispa However, the local ready list concept discourages process migration. More under light system load, the processor utilization may not be equally distrib among all processors. To permit process migration, a global ready list w resides in the shared memory may be used. This has the disadvantage of requ more overhead in saving and restoring process states by the process sched However, the standard deviation of the processor utilization is small.

The general objectives of many theoretical scheduling algorithms are develop processor assignments and scheduling techniques that use minim numbers of processors to execute parallel programs in the least time. In addi some develop algorithms for processor assignment to minimize the execution t of the parallel program when processors are available.

There are basically two types of models of processor scheduling, determinis and nondeterministic. In *deterministic* models, all the information required express the characteristics of the problem is known before a solution to problem, that is, a *schedule*, is attempted. Such characteristics are the execu time of each task and the relationship between the tasks in the system. The object of the resultant schedules is to optimize one or more of the evaluation criteria. F example, in deterministic models, the execution time of each process can either interpreted as the maximum processing time or as the expected processing tim In the former case, the time to complete the schedule would be consider the maximum time to complete the system of processes, and in the latt case, the length of the schedule represents a rough estimate of the mean leng of the computation. The motivation for this objective is that, in many case a poor schedule can lead to an unacceptable response time or utilization system resources.

Deterministic models are not very realistic and do not take into consideratio the irregular and unpredictable demands made on the multiprocessor syste Hence, stochastic models are often formulated to study the dynamic-schedulin techniques that take place in the system. In stochastic models, the execution tim of a process is a random variable t with a given *cumulation distribution functio* (**cdf**) F.

Processor scheduling implies that processes or tasks are to be assigned to particular processor for execution at a particular time. Since many tasks can be candidates for execution, it is necessary to represent the collection of tasks in manner which conveniently represents the relationships (if any) among the task Generally, we will refer to a set of related processes as a *task system* or a job. A job, which consists of a set of processes, is represented as a precedence graph, a shown in Figure 8.13. The nodes in the graph are tasks which may represe independent operations or parts of a single program which are related to eac other in time. The collection of nodes represents a set of processes $T = \{T_1, \ldots, T_n$ and the directed edge between nodes implies that a partial ordering or precedenc



...13 Representation of a task system, $T = (T_1, \ldots, T_{10})$.

...xists between the processes. Therefore, if $T_i < T_j$, process T_i must be ...ted before T_j can be initiated. Processes with no predecessors are called ...rocesses (e.g., T_1), and those with no successors are called *final* processes ... The individual nodes within a graph can be related to each other in a ... of ways.

...xample, it is possible for all processes in a graph to be independent of ...her. In this case, there is no precedence relation or partial ordering between ...esses and all the processes can be scheduled concurrently, provided there ...gh processors available. The *width* of the task graph G, denoted by ... the maximum size of any independent subset of processes. In Figure ... $T_1 < T_3$, $T_4 < T_7$, and $T_5 < T_7$. The width of the graph is 3. ...ed with each node is a second attribute which refers to the time required ...othetical processor to execute the code represented by the node. Sometimes this attribute is called the *weight* of the node. In a deterministic model, this ... a constant for each node, whereas, in a stochastic model, it is a random ... with a mean and standard deviation or a known distribution.

... computation graph and a multiprocessor system with p processors, a ...ment or a schedule must be developed such that it gives a description of ... to be run and in what order as a function of time. The schedule must

A number of measures have been developed to evaluate the effective
processor schedules. Some of these measures are (*a*) response or completion
(*b*) speed-up ratio, and (*c*) processor utilization. The objectives for multipro
resource management and scheduling are the same performance objectiv
for their uniprocessor counterparts, namely, maximizing throughput, minim
response time, or completing processing of tasks in order of priority. Consequ
the multiprocessor schedulers have much in common with single proc
schedulers.

8.3.2 Deterministic Scheduling Models

Deterministic schedules are usually displayed with timing diagrams called G
charts. We define some measures of performance based on Gantt charts. The
time of a process is equal to the time its execution is completed. The flow tim
a schedule is defined as the sum of the flow times of all processes in the sche
For example, the flow times of processes T_1 and T_4 in Figure 8.15 are seven
two, respectively, while the flow time of the schedule is 25.5. The *mean flow ti*
obtained by dividing the flow time of a schedule by the number of processe
the schedule. The *utilization* (or fractional busy time) of processors P_1, P_2, an
is 0.93, 1.00, and 0.86, respectively. These utilization values are obtained by divi
the time during which the processor was busy by the total time during whic
was available for execution. The *idle time* of P_1, P_2, and P_3 is 0.5, 0.0, and
respectively.

Figure 8.16 shows a process system schedule for a given program graph on
processors. The numbers associated with each node in the process graph repre
the execution time of the process. Figure 8.16*b* gives the optimal schedule for
graph using two processors. Note that this schedule is achieved by keepin
processor idle even when there is a process to execute. Figure 8.16*c* shows t
activating the schedulable process as soon as possible does not necessarily achi
an optimum schedule. The total execution time ET_1 of the process graph *G* o
uniprocessor is the sum of the numbers (weights) associated with each no
Hence, $\mathrm{ET}_1 = 27$. From Figure 8.16*b*, the execution time on the two-proces
system is $\mathrm{ET}_2 = 15$. Therefore the speedup, $S_p = \mathrm{ET}_1/\mathrm{ET}_2 = \frac{27}{15} = 1.8$, for $p =$

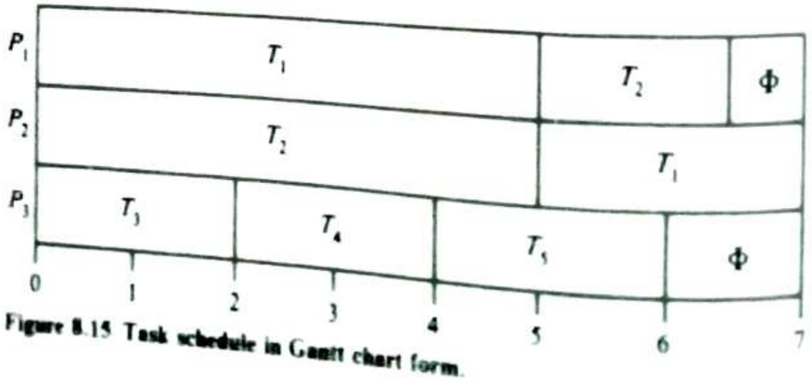
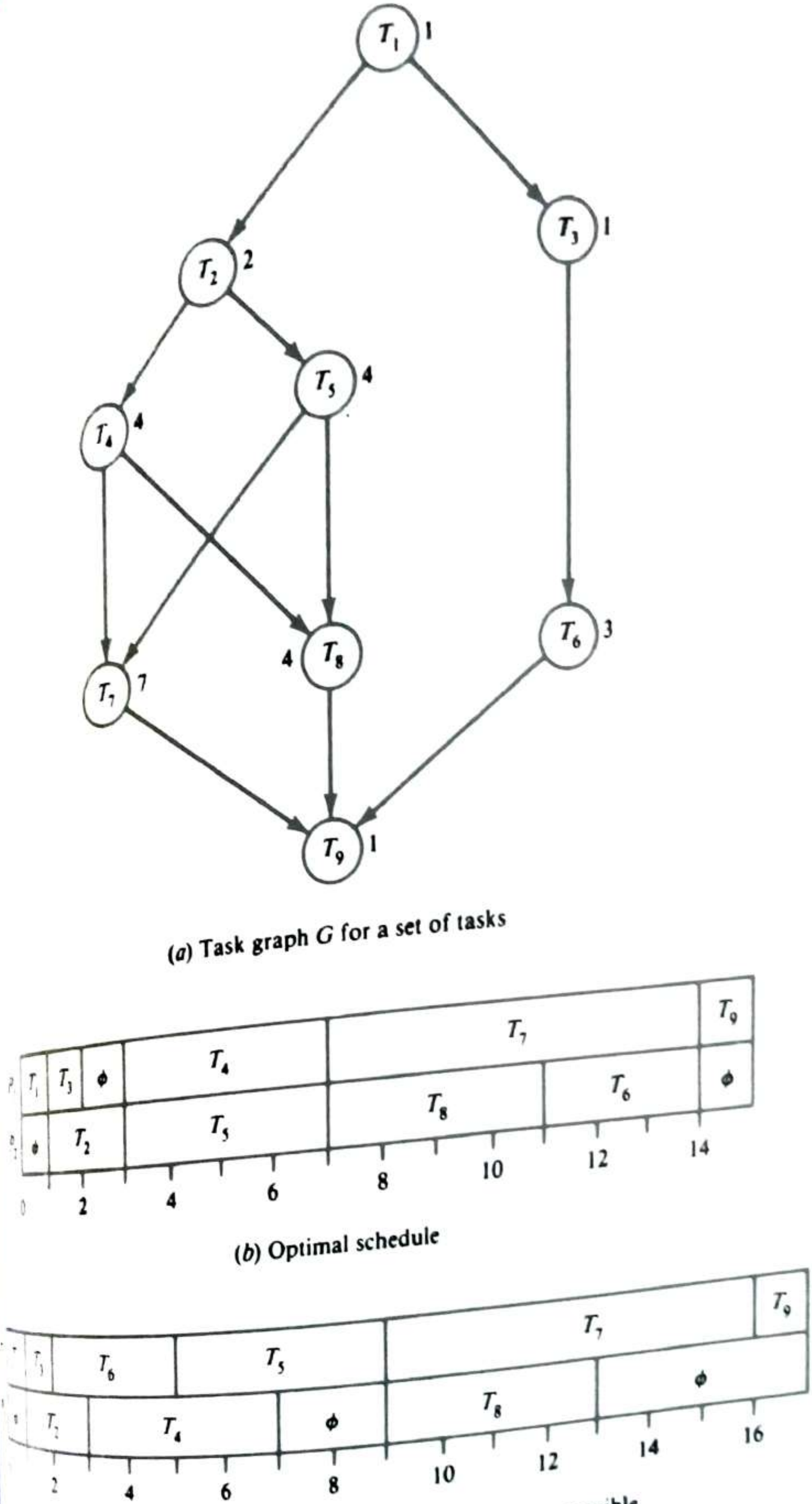


Figure 8.15 Task schedule in Gantt chart form.



(*a*) Task graph *G* for a set of tasks

(*b*) Optimal schedule

(*c*) Schedule when processors are activated as soon as possible

[illegible] Task schedule in chart form, using $p = 2$ processors. (Courtesy of *ACM Computing Surveys*, [illegible] Sept. 1977.)

The mean utilization U_p of the p processor system in the case of Figure 8 is $U_p = (30 - 3)/30 = 0.9$, for $p = 2$. The reader can easily show that by increa the number of processors to 3, the speedup does not increase. In fact, the utiliza reduces to 0.6. Hence, the execution of the process graph in Figure 8.16a is m cost-effective on a two-processor system. The rationale behind the minimizatio finishing or completion time is that system throughput can be maximized if total computation time of each set of processes is minimized. Throughpu defined as the number of process sets processed per unit of time.

There are at least two reasons for minimizing the number of process required to process a process system. The first and most obvious is cost. second reason is the processor utilization. If the number of processors requi to execute a set of processes in a given time is less than the total number of p cessors available, then the remaining processors can be used as backup process for increased reliability and as background processors for noncritical co putations.

A key issue in the study of processor scheduling is the amount of overhead computation time needed to locate a suitable schedule. A scheduling algorithm a procedure that produces a schedule for every given set of processes. An effici scheduling algorithm is one that can locate a suitable schedule in an amount time that is bounded in the length of the input by some polynomial. Constructi of *optimal* schedules is NP-complete in many cases. NP-complete implies that optimal solution may be very difficult to compute in the *worst* possible inp case. However, construction of *suitable* schedules, that is, computing a reasonal answer for the typical input case, is not NP-complete. Therefore suitable schedu can be obtained for concurrent processes.

In this subsection, we examine deterministic schedules which can be used optimize measures of performance. Unless stated explicitly, we assume a scheduli environment which consists of a number of identical processors, a set of process with equal or unequal execution times and a (possibly empty) precedence ord First we consider preemptive schedules using two processors.

In order to understand the preemptive schedule (PS) on p processors, define process graphs with mutually commensurable node weights. A set of nod is said to be *mutually commensurable* if there exists a t such that each node weig is an integer multiple of t. In a preemptive schedule, a processor may be pr empted from an executing process if such an action results in an improved measu of performance. The PS algorithms are due to Muntz and Coffman (1966).

Assume that the process graph consists of n independent processes wi weights (process duration or execution times) of $t_1, t_2, \ldots, t_n$ and p processor The optimal PS has a completion time of:

$$\omega = \max\left\{\max_{1 \le i \le n} \{t_i\}, \frac{1}{p} \sum_{i=1}^{n} t_i\right\} \qquad (8.$$

The optimal PS length cannot be less than the larger of the longest process or th sum of the execution times divided by the number of processors.

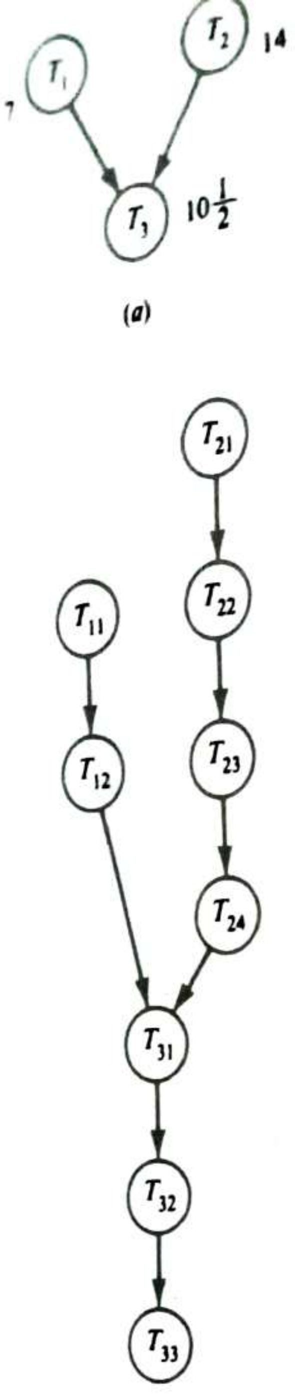
optimal algorithm, the set of nodes of unit weight in a graph are into a sequence of disjoint subsets such that all nodes in a subset are All nodes in the same subset or at the same level are candidates for execution or group scheduling. In a graph of N subsets or levels, node occupies the first level exclusively. Those nodes which may be during the unit time period preceding the execution of the terminal node second level, and so on. The initial or entrance node in the graph the Nth level. Such an assignment of levels generate what is called *partitions*.

In particular, the assignment procedure outlined above corresponds to the precedence partitions. That is, the assignment of nodes to levels is done in a which defers process initiation to the latest possible time without increasing minimum completion time. Such a schedule is called the *latest-scheduling* This strategy assumes that the number of processors available is greater or equal to the maximum number of processes at any level (width of G). This may be contrasted with the *earliest-scheduling strategy*, which schedules as soon as a processor is available and the precedence constraints have satisfied. Note that the earliest strategy produces earliest-precedence parti-

For any arbitrary graph G, a precedence relation will exist between the subsets the latest strategy due to the precedence which exists between the nodes in the graph. A PS can be constructed for graph G by first scheduling the numbered subset, then the subset at the next lower level, and so on. Note when a subset consists of only one node, a node from the next lower subset is up if it does not violate the precedence constraints of the original graph. of the subsets is scheduled optimally, a *subset schedule* results. For two and equally weighted nodes, an optimal subset schedule for G is an PS for G.

This result is extended to the case of graphs having mutually commensurable weights. In order to generate the optimal result, it is necessary to convert G into another graph G_w in which all nodes have equal weights. This is done a node of weight t_i and creating a sequence of n nodes such that $t_i = nt$, illustrated in Figure 8.17. Note that the integrity of the original graph must be ined It can then be shown that an optimal subset schedule for G_w is an PS for G, with $k = 2$.

this approach, one must note whether the number of processes at any level odd. If it is even, then all processes at that level can be executed in the amount of time with no idle time for either of the two processors. If of processes is not a multiple of two, then the last three processes to duled at that level can be executed in no less than $\frac{1}{2}$ unit, since all pro- are of unit duration. By using the form shown in Figure 8.18, three a given level can be executed in minimum time without processor idle scheduling in this manner ensures that no processor is idle, the subset be seen to generate a minimal-length PS. An example of the optimal shown in Figure 8.19. For this example, the optimal subset sequence $T, T_1\}, \{T_5, T_6, T_7\}, \{T_4, T_8\}, \{T_9, T_{10}\}, \{T_{11}\}$.



(*a*)

(*b*)

Figure 8.17 Comparison of a graph with mutually commensurable weights with the corresponding graph having nodes of equal weigh Graph *G* node weights $w_1 = 7$, $w_2 = 14$, $w_3 = 10\ 1/2$; (*b*) graph $w = 3\ 1/2$.

The optimal results derived above can be extended to the case in which number of processors are allowed when the computation graph is a *rooted* and the node weights t_i are mutually commensurable. A *rooted tree* is one in w each node has at most one successor, with the exception of the root or term node, which has no successors. We discuss below some techniques for non emptive schedules.

Recall that, in nonpreemptive or basic schedules, a processor assigned process is dedicated to that process until it is completed. The initial investigati discussed here develop optimal nonpreemptive two-processor schedules arbitrary process orderings in which all processes are of unit duration. A partic

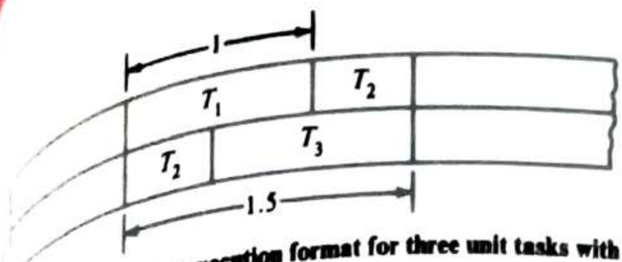


Figure 8.18 Minimum-time execution format for three unit tasks with two processors.

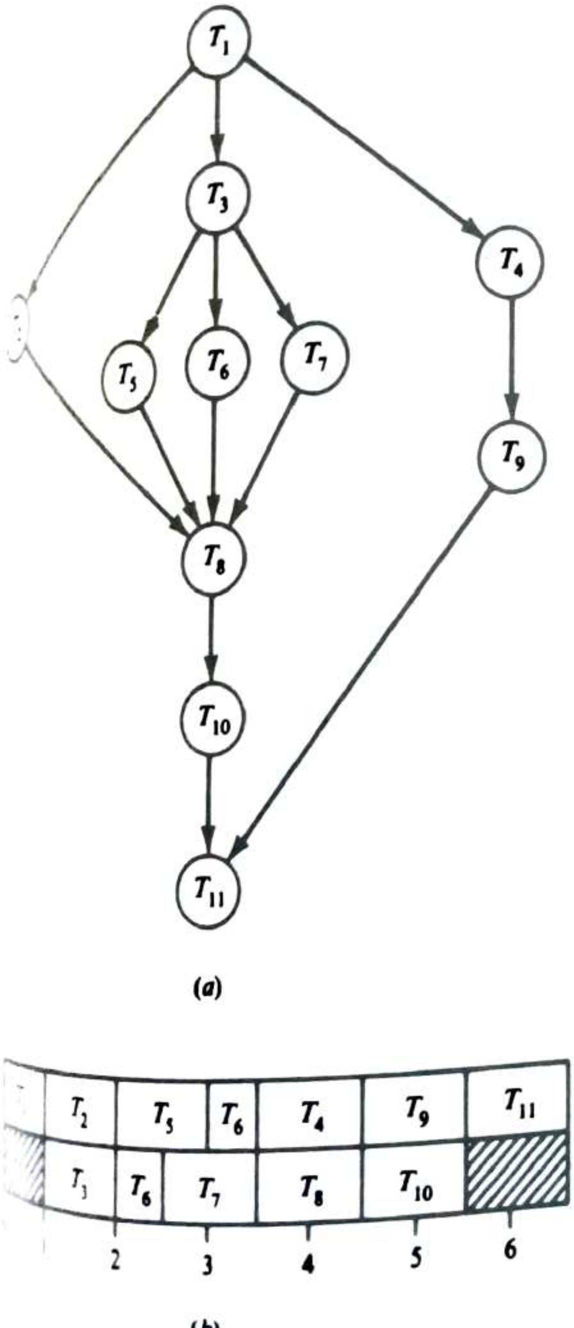


Figure 8.19 Illustration of subset sequence algorithm. (*a*) Graph *G* for a set of tasks, with all nodes having unit weight; (*b*) optimal preemptive schedule. (Courtesy of *ACM Computing Surveys*, Gonzales, Sept. 1977.)

simple class of scheduling algorithms for nonpreemptive schedules is the cl *list-scheduling algorithms*. A list-scheduling algorithm assigns distinct pri processes and allocates resources to the processes with highest priorities those runnable at any time instant when the resource becomes free. A list sch or list L for a graph G of n processes is denoted by $L = (T_1, T_2, \ldots, T_n)$ and rep sents some permutation of the n processes. A process is said to be ready if all of predecessors have been completed. In using a list to generate a schedule, an processor is assigned to the first ready process found in the list. An algorithm generating such an optimal list is described below.

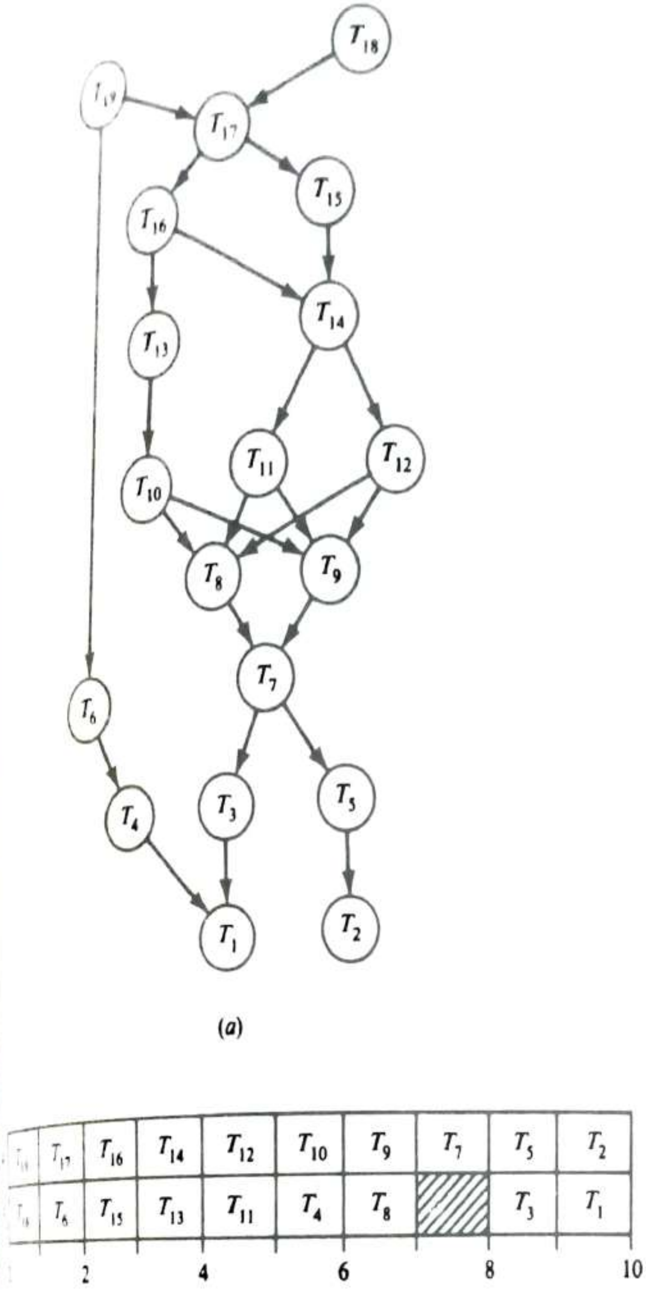
The algorithm is a recursive procedure which begins by assigning subscri in ascending order to the process (processes) which is (are) executed last owing precedence constraints in the process graph. Notice that the set of successors these processes is empty. Assignment proceeds "up the graph" in a manner th considers as candidates for the assignment of the next subscript all processes wh successors have already been assigned a subscript. Consideration of processes this manner amounts to examining processes in a given latest-precedence partitio although the processes are not executed at a time that corresponds to this partitio In effect, the processes in a graph can be initially assigned subscripts in an arbitra manner. This algorithm then reassigns subscripts in the method outlined abo The list is formed by listing the processes in decreasing subscript order, beginni with the last subscript assigned. The optimal schedule is formed by assignin ready processes in the list to idle processors. The algorithm is illustrated in Figu 8.20 by means of a process graph with reassigned subscripts, the resultant list L and the optimal schedule.

The above algorithm does not always yield optimal results when the numb of processors is increased beyond two, or when the number of processors is tw and processes are allowed to have arbitrary durations. We describe a nonpre emptive scheduling method by Hu (1961). Two problems for process of un duration were addressed. In the first case, given a fixed numbers of processors, it required to determine the minimum time required to execute a process graph. Th second case determines the number of processors required to process a graph in given time.

We begin to arrive at a solution to these problems if we develop a labelin scheme for the nodes of the graph. A node T_i is given the label $\alpha_i = X_i + 1$, wher X_i is the length of the longest path from T_i to the final node in the graph. Labelin begins with the final node, which is given the label $\alpha_1 = 1$. Nodes that are one un removed from the final node are given the label 2, and so on. This labeling schem makes it clear that the minimum time $\omega_{\min}$ required to execute the graph is relate to $\alpha_{\max}$, the node(s) with the highest numbered label, by

$$\omega_{\min} \geq \alpha_{\max}$$

The optimal solutions by Hu are limited to rooted trees. Using the labelin procedure described above, one can obtain an optimal schedule for p processor by processing a tree of unit-length processes in the following manner:



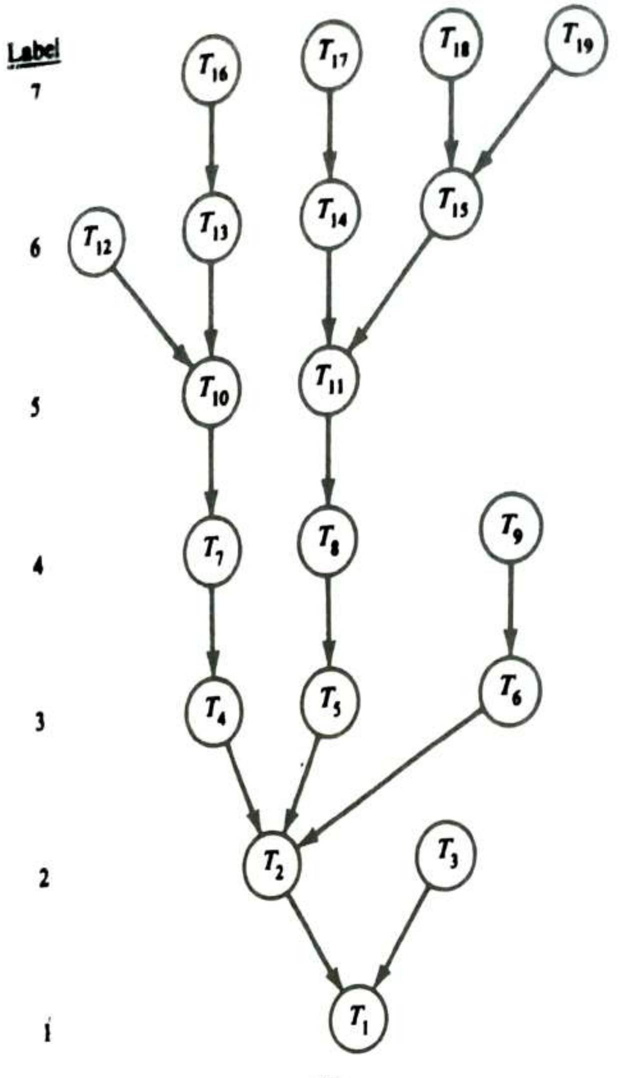
T_{19}	T_{17}	T_{16}	T_{14}	T_{12}	T_{10}	T_9	T_7	T_5	T_2
T_{18}	T_6	T_{15}	T_{13}	T_{11}	T_4	T_8	▨	T_3	T_1

…20 **Illustration of Coffman and Graham algorithm. (*a*) Task graph with reassigned subscripts** … $T_{18}, \ldots, T_1$); (*b*) **optimal schedule. (Courtesy *ACM Computing Surveys*, Gonzales, Sept.**

edule first the p (or fewer) nodes with the highest numbered label, i.e., the ting nodes. If the number of starting nodes is greater than p, choose p nodes se α_i is greater than or equal to the α_i of those not chosen. In case of a tie, choice is arbitrary.

ete the p processed nodes from the graph. Let the term "starting node" refer to a node with no predecessors.

eat steps 1 and 2 for the remainder of the graph.



(a)

P_1	T_{19}	T_{16}	T_{15}	T_{11}	T_8	T_5	T_2	T_1
P_2	T_{18}	T_{14}	T_{13}	T_{10}	T_7	T_4	▨	▨
P_3	T_{17}	T_{12}	T_9	T_6	T_3	▨	▨	▨

(b)

Figure 8.21 Illustration of Hu's optimal algorithm. (a) Rooted tree labeled according to Hu's proce (b) optimal schedule for three processors. (Courtesy *ACM Computing Surveys*, Gonzalez, Sept. 197

The schedules generated in this manner are optimal under the stated constrai The labeling and scheduling procedures are quite simple to implement and illustrated in Figure 8.21.

Recall that the minimum time required to execute a task graph by H procedure is $\alpha_{\max}$. Suppose we wish to process a graph within a prescribed tim where $t = \alpha_{\max} + C$ and C is a nonnegative integer. The minimum numb

...rs p required to process the graph in time t is given by

$$p - 1 < \frac{1}{\gamma^* + C} \sum_{j=1}^{\gamma^*} p(\alpha_{\max} + 1 - j) < p \qquad (8.4)$$

...) denotes the number of nodes in the graph with label α, and γ^* is the ...the constant γ, which maximizes the given expression. To illustrate this ...nsider Figure 8.21. For $C = 0$, for example, value γ^* occurs when $\gamma = 1$ ... This indicates that, in order to process the graph in minimum time, four ...ssors are needed. For $C = 1$, $t = 8$ and γ^* occurs when $\gamma = 2$ or $\gamma = 5$, and ...ocessors are required. Varying C further, we find that three processors are ...d when the processes must be processed within nine units, but only two pro...s are needed for a maximum processing time of 10 units.

Another study by Graham shows that, for a computing system with n identical ...cessors in which processes are assigned arbitrarily to the processors, the ...mpletion time of the set of processes will not be more than twice the time required ...n optimal schedule. This bound was derived in connection with the so-called multiprocessor *anomalies*. These anomalies are derived from the counterintuitive ...servation that the existence of one of the following conditions can lead to an ...crease in execution time:

Replace a given process list L by another list L', leaving the set of process times μ, the precedence order $<$, and the number of processors n unchanged.

Relax some of the restrictions of the partial ordering.

Decrease some of the execution times.

Increase the number of processors.

A general bound has been obtained by executing a set of processes twice. During the first execution, the processes are characterized by the parameters μ, $<$, L, n, and ω (the length of the schedule), and during the second execution by μ', $<'$, L', n', and ω' such that $\mu' \leq \mu$ and every constraint of $<'$ is also in $<$, i.e., $<'$ is contained in $<$. The result of this general bound is that

$$\frac{\omega'}{\omega} \leq 1 + \frac{n - 1}{n'} \qquad (8.5)$$

This bound is the best possible, and, for $n = n'$, the ratio $2 - 1/n$ can be achieved ...the variation of any on of L, μ, or $<$.

The above result was extended to a nonhomogeneous processor system by ... and Liu. Suppose a multiprocessor system consists of n_i processors of speed ...for $i = 1, 2, \ldots, k$, such that $\mu_1 > \mu_2 > \cdots > \mu_k \geq 1$, then

$$\frac{\omega'}{\omega} \leq \frac{\mu_1}{\mu_k} + 1 - \frac{\mu_1}{\sum_{i=1}^{k} n_i \mu_i} \qquad (8.6)$$

Example 8.8 Consider a system with one processor of speed five and processors of speed one. By Eq. 8.6, we have

$$\frac{\omega'}{\omega} \leq \frac{5}{1} + 1 - \frac{5}{10} = \frac{11}{2}$$

Comparing this bound with that in Eq. 8.5 for a multiprocessor system 10 identical processors of speed 1 (by substituting five identical processors speed 1 for the processor of speed 5), the ratio $2 - \frac{1}{10}$ is achieved. The determination of a close to optimal schedule is more important for a heterogeneous system than for a homogeneous system.

Because of the limitations on optimal algorithms, bounds have been de for the behavior of nonoptimal algorithms. The concept of precedence par can be used to generate bounds for processing time and the number of proce for graph structures whose nodes require unit-execution time. As indicated ear precedence partitions group processes into subsets to indicate the earliest latest times during which processes can be initiated and still guarantee minim execution time for the graph. This time is given by the number of partitions an a measure of the longest path in the graph. For a graph of N levels, the minim execution time is $\omega = N$ units. In order to execute a graph in this minimum ti the lower bound on the number of processors p required is given by

$$p \geq \max\left[\max_{1 \leq i \leq N} |L_i \cap E_i|,\ \max_{1 \leq i \leq N}\left[\frac{1}{i}\sum_{i=1}^{N}|L_i|\right]\right]$$

and the upper bound on the number of processors p required is given by

$$p \leq \min_{1 \leq i \leq N}\left[\max_{1 \leq i \leq N}|L_i|,\ \max_{1 \leq i \leq N}|E_i|\right]$$

In both cases, L_i and E_i refer to the ith latest and earliest precedence partitions respectively, and $|x|$ represents the cardiality of the set x. The processes contained in $L_i \cap E_i$ are called *essential processes*. Those processes contained in the subset given by $L_i \cap E_i$ must be initiated $i - 1$ units after the start of the initial process in the graph to guarantee minimum execution time.

8.3.3 Stochastic Scheduling Models

Using nondeterministic techniques, the execution time of a process T_i is given the random variable t_i, with cumulative distribution function F_i. Given a process graph G, let t_G be the random variable representing total execution time (the time from when all processes are started until the last process terminates). Assume has a **cdf** F_G. There is a class of process graphs for which F_G can be expressed simply in terms of F_i. Below, we give a methodology, developed by Robinson (1979), for estimating F_G for this class.

graphs In order to determine the possible execution of job T we define simple process graphs. A subgraph of a process graph G is a *chain* if processes in the subgraph are totally ordered. The length of a chain is processes in the chain. If, in a chain, the initial process is T_i and the processes T_j, we say it is a chain from T_i to T_j. A subgraph of a process graph chain is said to be a chain in G. In the following definition, a class of graphs is defined for which F_G can be expressed simply in terms of the process execution time F_i. Let $C_1, C_2, \ldots, C_m$ be all chains from initial to processes in G. For each chain C_i containing processes $T_{i_1}, T_{i_2}, \ldots$, let X_i be expression $x_{i_1}x_{i_2}\ldots$, formed by concatenating the polynomial variables associated with processes $T_{i_1}, T_{i_2}, \ldots$, respectively. Then G is said to simple if the polynomial $X(G) = X_1 + X_2 + \cdots + X_m$ can be factored so that variable appears exactly once. Examples of simple and nonsimple process graphs are shown in Figure 8.22.

The class of parallel algorithms represented by simple process graphs are those that can be written in block-structured languages with parallel provided no synchronization takes place between any of the components parallel block.

A set of processes is *independent* if, for any two processes T_i and T_j in the set, neither $T_i < T_j$ not $T_j < T_i$. In this example (Figure 8.22a), processes in set $T_4\}$ are independent. So also are processes in set $\{T_3, T_4\}$. Figure 8.22 shows some process graphs to explain the simplicity of graphs.

Let K be the number of processors in the system. If $K \geq \text{width}(G)$, each process begins execution immediately after the last predecessor completes. Let $C_1, \ldots, C_m$ be all the chains from initial to final processes in graph G. Also let execution time of process T_i be t_i with **cdf** F_i. Then the total execution time of system G is the maximum of the execution times of all the chains in G. That is,

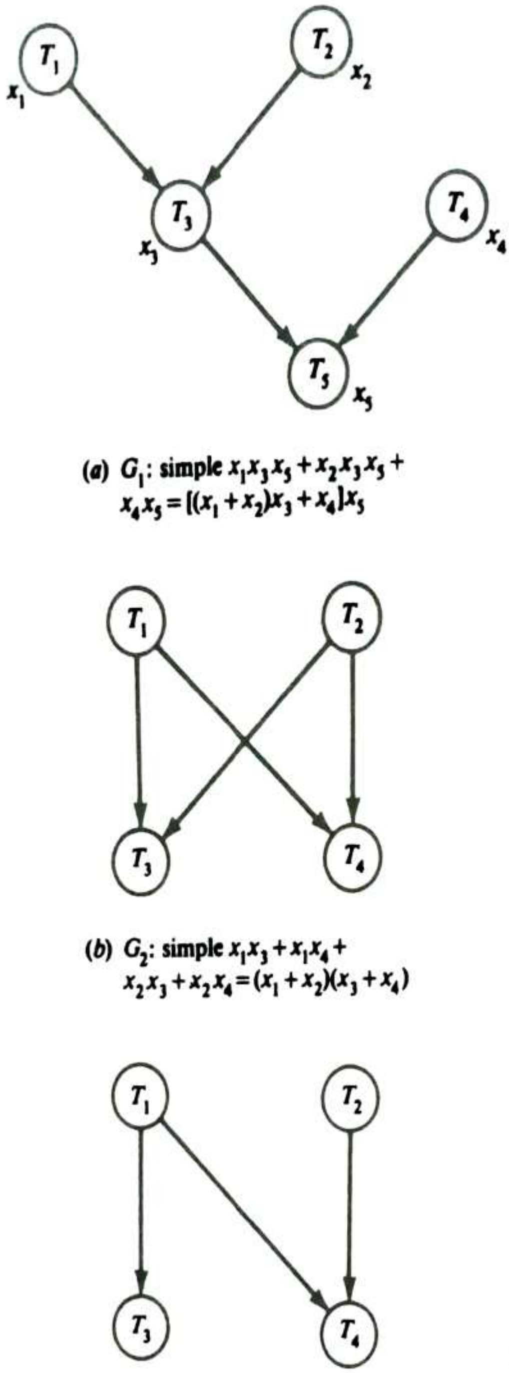
$$t_G = \max_{1 \leq i \leq m} \sum_{T_j \in C_i} t_j \tag{8.9}$$

Note that + and max are commutative and associative operations, respectively. Moreover, + distributes over max. For example, $\max(a, b) + c = \max(a + c, b + c)$. Thus, if G is simple, the expression for t_G above can be factored in terms of max and + so that each random variable appears only once. Then, if t_i's are independent, the expression for F_G may be found by substituting F_i * (convolution) for +, and · (multiplication) for max in the expression for The convolution of **cdfs** F_1 and F_2 is written as follows:

$$F_1 * F_2(t) = \int_{-\infty}^{\infty} F_1(t-u)F_2\, du \tag{8.10}$$

In the example in Figure 8.23, there are three chains, $C_1 = T_1T_3T_5$, $C_2 =$ and $C_3 = T_3T_5$. Therefore

$$t_G = \max\{(t_1 + t_3 + t_5), (t_t + t_3 + t_5), (t_4 + t_5)\}$$



(*a*) G_1: simple $x_1x_3x_5 + x_2x_3x_5 + x_4x_5 = [(x_1 + x_2)x_3 + x_4]x_5$

(*b*) G_2: simple $x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4 = (x_1 + x_2)(x_3 + x_4)$

(*c*) G_3: nonsimple $x_1x_3 + x_1x_4 + x_2x_4$

Figure 8.22 Examples of simple and nonsimp task graphs. (Courtesy *IEEE Trans. Softwa Engg.* Robinson, January 1979.)

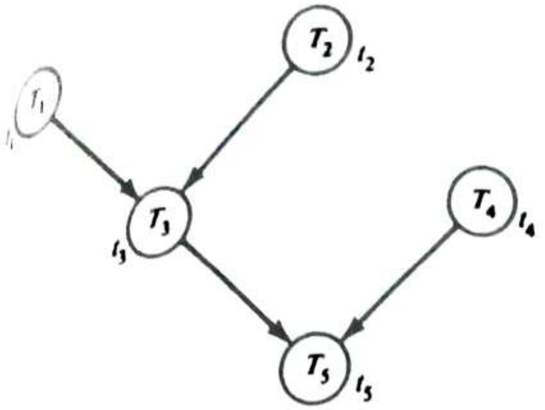
Since $t_3 + t_5$ is common in the first two summations,

$$t_G = \max\{\max(t_1, t_2) + t_3 + t_5, t_4 + t_5\}$$

This expression can be factored further by noting that t_5 is common to both summations:

$$t_G = \max\{\max(t_1, t_2) + t_3, t_4\} + t_5 \qquad (8.11)$$

Chains:
$C_1 = T_1 T_3 T_5$
$C_2 = T_2 T_3 T_5$
$C_3 = T_4 T_5$



$$t_G = \max_{T_j \in C_i} \{\Sigma t_j\} = \max\{(t_1 + t_3 + t_5), (t_2 + t_3 + t_5), (t_9 + t_5)\}$$

$$= \max\{\max(t_1, t_2) + t_3, t_4\} + t_5 \quad \text{and} \quad 1 \le i \le m$$

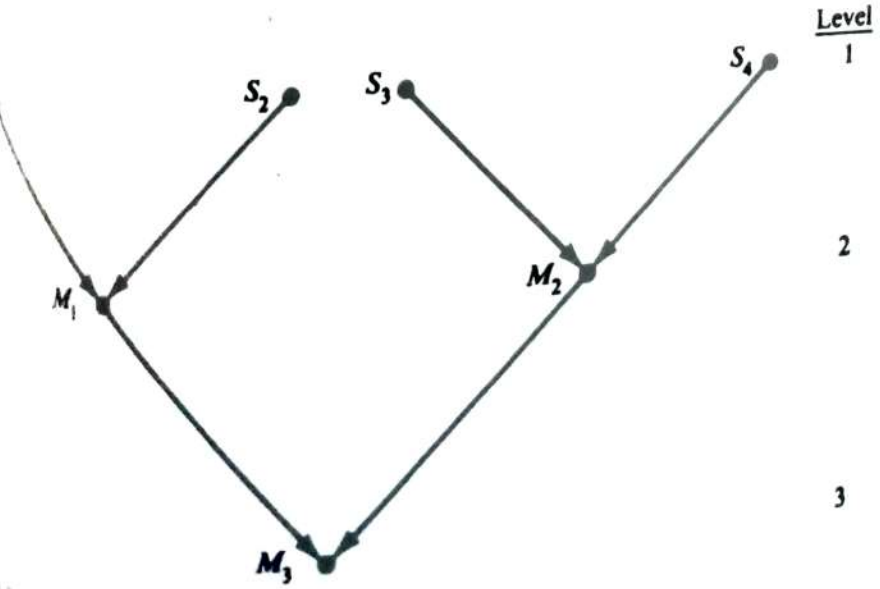
$$F_G = \{(F_1 F_2) \bullet (F_3 F_4)\} \bullet F_5$$

Figure 8.23 Computation of t_G and F_G for task graph G. (Courtesy *IEEE Trans. Software Engg.*, Robinson, Jan. 1979.)

In Eq. 8.11, each random variable appears only once. Hence, F_G may be found by the substitution rule:

$$F_G = (((F_1 \cdot F_2) * F_3) \cdot F_4) * F_5 \tag{8.12}$$

As another example, consider the four-process merge-sort depicted by the rocess graph in Figure 8.24. Process S_i performs the sorting of one distinct subfile which is a fourth of file A. After the pair of either subfiles a_1 and a_2 or a_3 and a_4 sorted, they are merged by the execution of process M_1 or M_2, respectively.



24 Four-process merge sort.

Merging is a method of combining two or more sorted files into a compo file In Figure 8.24, each M_i is a merge of the sorted subfiles produced b mediate predecessors.

If all permutations of keys are equally likely, then the execution tim S_2, S_3, and S_4 have the same **cdf** and the execution times of M_1 and M_2 same **cdf**. Let the **cdf** of the execution times of the S_i's be F_1 and that of M_2 be F_2. Furthermore, let the **cdf** of the execution time of M_3 be F width$(G) = 4$. G is simple and the process-execution times are independ the execution times of the S_i's and M_j be t_{S_i} and t_{M_j}, respectively. Hen execution time of the four-process merge-sort is

$$t_G = \max\{\max(t_{S_1}, t_{S_2}) + t_{M_1}, \max(t_{S_3}, t_{S_4}) + t_{M_2}\} + t_{M_3}$$

Since $t_{S_1}, t_{S_2}, t_{S_3}, t_{S_4}$ have the same **cdf** F_1 and t_{M_1}, t_{M_2} have the same **cdf** **cdf** of t_{M_3} is F_3:

$$F_G = (F_1^2 * F_2)\cdot(F_1^2 * F_2) * F_3 = (F_1^2 * F_2)^2 * F_3$$

This should be compared with the **cdf** of the execution time for a one-p (sequential) merge-sort:

$$F_{\mathbf{seq}} = F_1 * F_1 * F_1 * F_1 * F_2 * F_2 * F_3$$

since the execution time for the sequential merge-sort is

$$t_{\mathbf{seq}} = t_{S_1} + t_{S_2} + t_{S_3} + t_{S_4} + t_{M_1} + t_{M_2} + t_{M_3}$$

Notice that Eq. 8.15 assumes that the processing environment of the sequ merge-sort is the same as the concurrent merge-sort. In practice, this is no since the sequential merge-sort does not encounter interprocess-commun problems or memory conflicts which create overheads in the concurrent me sort. Hence, in practice, $t_{\mathbf{seq}}$ is usually less than that predicted by Eq. 8.15 I next section, we consider the effect of these overheads on the performance of algorithm.

Let μ_G and $\mu_{\mathbf{seq}}$ be the mean execution times of the probability density func $f_G = F'_G$ and $f_{\mathbf{seq}} = F'_{\mathbf{seq}}$, respectively. We can then estimate the theor speedup of the four-process merge-sort as

$$S_p = \frac{\mu_G}{\mu_{\mathbf{seq}}}$$

Equation 8.9 is not very useful when the **cdfs** of the process execution are not known. Bounds can be derived for the mean execution time by using limited knowledge about the execution times of processes. Let us denot expected value of a random variable x by $E(x)$. The level of a process T in a pr graph G is the maximum length of any chain in G from an initial process The *depth* of G, denoted by depth(G), is the maximum level of any process a process graph G with the number of available processors $K \geq$ width(G) with the t_i independent, let $C_1, C_2, \ldots, C_m$ be all chains in G from initial to

Also let H_i be the set of all processes of level i, for $1 \le i \le L$, where ... For any set of n random variables $\{x_i\}$,

$$E\left(\max_{1 \le i \le n} \{x_i\}\right) \ge \max_{1 \le i \le n} \{E(x_i)\} \tag{8.17}$$

which the lower bound follows. For the upper bound, let $t_0 \equiv 0$ and define ... 0 if $C_i \cap H_j$ is empty; otherwise $f(i, j)$ is the index of the single process in H_j. Then from Eq. 8.9,

$$t_G = \max_{1 \le i \le m}\left(\sum_{1 \le j \le L} t_{f(i,j)}\right) \le \sum_{1 \le j \le L} \max_{1 \le i \le m} (t_{f(i,j)}) \tag{8.18}$$

...refore

$$\max_{1 \le i \le m}\left(\sum_{T_j \in C_i} E(t_j)\right) \le E(t_G) \le \sum_{1 \le i \le L} E\left(\max_{T_j \in H_i} t_j\right) \tag{8.19}$$

The upper bound in Eq. 8.19 is useful only if something can be said about $\max(t_j)$). An applicable result from *order statistics* is that, if the random variables ... x_m are independent and identically distributed (i.i.d.) with the mean ... and standard deviation σ, then

$$E\left\{\max_{1 \le i \le m} \{x_i\}\right\} \le \mu + \frac{m-1}{\sqrt{2m-1}}\sigma \tag{8.20}$$

...ce if the number of available processors $K \ge \text{width}(G)$, the t_i's are independent, ...pth$(G) = L$ and the m_j processes on level j have identically distributed execution ...es with the mean μ_j and standard deviation σ_j, then

$$\sum_{1 \le j \le L} \mu_j \le E(t_G) \le \sum_{1 \le j \le L}\left(\mu_j + \frac{m_j - 1}{\sqrt{2m_j - 1}}\sigma^j\right) \tag{8.21}$$

ueueing model Probabilistic models are often formulated to investigate the ...perties of dynamic scheduling methods that take the form of queueing systems. ...se require the specification of certain characteristics and attributes of the ...eing system, such as the probability distribution functions of the interarrival ...s of processes, the service times of the processes, and the specification of the ...ice discipline. The service or queueing discipline is the scheduling rule which ...rmines both the sequence in which processes are executed and the processor-...pancy period each time a process is selected for service. A number of assumptions are usually made regarding queueing systems to make the analytical model ...able. These include the independence of processes and the statistical ...endence of the interarrival and service times.

A very simple model of scheduling in a multiprocessing system consists of p ...cal processing elements and a single infinite queue to which processes arrive. ...odel may be appropriate for a system with a global ready list and no pre... The mean processing time of processes on each processor is $1/\mu$ and the mean interarrival time of processes to the system is $1/\lambda$. Assuming that the ser and interarrival times are exponentially distributed and the service discipline processes is the common first-come-first-serve (FCFS), various perform factors can be obtained. Figure 8.25 illustrates the resulting queueing mode which processes arrive at a rate λ and are serviced at a rate μ. The utilization of processors is

$$\rho = \frac{u}{p}$$

where u is the traffic intensity and is defined by $u = \lambda/\mu$. The mean response of processes is [Kleinrock (1976)]

$$\overline{R}(\rho, u) = \frac{C(\rho, u)}{\mu p(1 - \rho)} + \frac{1}{\mu}$$

where $C(\rho, u)$ is Erlang's C formula and is given by

$$C(\rho, u) = \frac{u^p}{u^p + p!(1 - \rho) \sum_{n=0}^{p-1} \frac{u^n}{n!}}$$

There are a number of other scheduling algorithms, such as *round robin* (R preemptive, and nonpreemptive priority service disciplines, which can be mode by the use of queueing systems. In an RR service discipline, each time a proces selected for execution, it is selected from the head of the ordered queue and alloca a fixed duration of run time called the *time slice* or *quantum*. If a process termina execution before the end of the quantum, it departs from the processor. If at

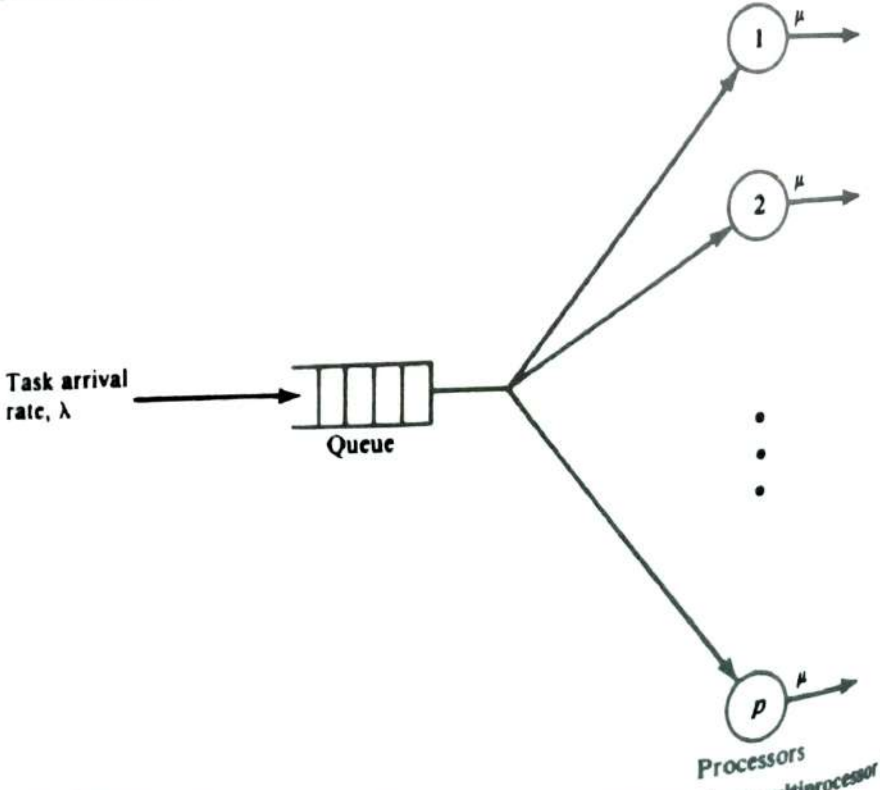


Figure 8.25 Queueing model of first-come-first-serve scheduling discipline in a multiprocessor system

the quantum the process has not completed its execution (requires additional tum), it is recycled to the end of the queue to await its next selection. New arrivals simply join the end of the queue. The RR service discipline can be bined with the preemptive priority discipline to create a multilevel round-robin duling discipline. This discipline is used to give higher priority processes frequent control than lower priority ones. Policies based on priority can be (if the priority of a process remains fixed) or dynamic (if the priority of a ess is allowed to change).

In the RR service discipline, a process in the run state is interrupted at the of its quantum and may enter the ready state. An external event may cause the king of a running process. These transitions may necessitate a context switch. thermore, a running process can cause an explicit process switch by invoking vileged instruction. For example, in the case of a fault, the process can cause a which switches context to the operating system, as in the IBM 370 supervisor (SVC) instruction to be described in Chapter 9.

Long-term scheduling operations are used to control the load on the multi-essor system by making decisions on activating new processes. One method mplement the schedule is to use priority queues for incoming processes. itization of processes in a system may result in indefinite postponement of priority processes if the arrival rate of the high priority processes is high. A of processes which cooperate to solve a problem may be given higher priority n a single independent process.

Since there are many processors as well as memory modules to be scheduled, may be useful to perform *group scheduling*, in which a set of related processes are gned to processors to run simultaneously. Group scheduling can be extended make placement decisions for groups of objects at a time, or to swap groups of ted objects in and out. These different group schedulers have several possible ntages. First, if closely related processes run in parallel, blocking due to hronization and frequency of context switching may be reduced. These will ffect aid in increasing performance. Second, if placement decisions are made group of objects with known reference patterns, the "distance" between the ous processes and their referenced objects might be minimized. Hence, tive memory management for a set of related processes is easier since me period for sharing is restricted to the short presence of the processes system. In general, a group assignment will not be very successful in lessening mber of context switches unless the processes within the group are "in step" so of them will be blocked from lack of input or other synchronization rements.