Computer Development Milestones

Computers have gone through two major stages of development: mechanical and electronic. Prior to 1945, computers were made with mechanical or electromechanical parts. The earliest mechanical computer can be traced back to 500 BC in the form of the abacus used in China. The abacus is manually operated to perform decimal arithmetic with carry propagation digit by digit.

Blaise Pascal built a mechanical adder/subtractor in France in 1642. Charles Babbage designed a difference engine in England for polynomial evaluation in 1827. Konrad Zuse built the first binary mechanical computer in Germany in 1941. Howard Aiken proposed the very first electromechanical decimal computer, which was built as the Harvard Mark I by IBM in 1944. Both Zuse's and Aiken's machines were designed for general-purpose computations.

Computer Generations Over the past five decades, electronic computers have gone through five generations of development. Table 1.1 provides a summary of the five generations of electronic computer development. Each of the first three generations lasted about 10 years. The fourth generation covered a time span of 15 years. We have just entered the fifth generation with the use of processors and memory devices with more than 1 million transistors on a single silicon chip.

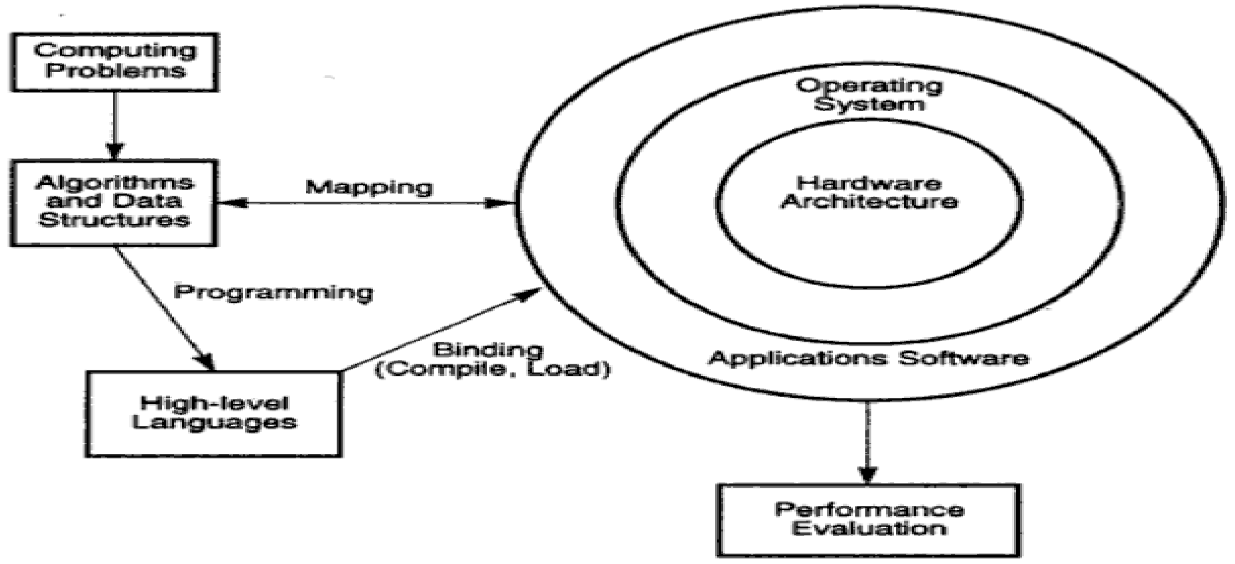
Generation	Technology and Architecture	Software and Applications	Representative Systems
First (1945–54)	Vacuum tubes and relay memories, CPU driven by PC and accumulator, fixed-point arithmetic.	Machine/assembly languages, single user, no subroutine linkage, programmed I/O using CPU.	ENIAC, Princeton IAS, IBM 701.
Second (1955–64)	Discrete transistors and core memories, floating-point arithmetic, I/O processors, multiplexed memory access.	HLL used with compilers, subroutine libraries, batch processing monitor.	IBM 7090, CDC 1604, Univac LARC.
Third (1965–74)	Integrated circuits (SSI/-MSI), microprogramming, pipelining, cache, and lookahead processors.	Multiprogramming and time-sharing OS, multiuser applications.	IBM 360/370, CDC 6600, TI-ASC, PDP-8.
Fourth (1975–90)	LSI/VLSI and semiconductor memory, multiprocessors, vector supercomputers, multicomputers.	Multiprocessor OS, languages, compilers, and environments for parallel processing.	VAX 9000, Cray X-MP, IBM 3090, BBN TC2000.
Fifth (1991–present)	ULSI/VHSIC processors, memory, and switches, high-density packaging, scalable architectures.	Massively parallel processing, grand challenge applications, heterogeneous processing.	Fujitsu VPP500, Cray/MPP, TMC/CM-5, Intel Paragon.

Elements of Modern Computers

Computing Problems It has been long recognized that the concept of computer architecture is no longer restricted to the structure of the bare machine hardware. A modern computer is an integrated system consisting of machine hardware, an instruction set, system software, application programs, and user interfaces. These system elements are depicted in Fig. 1.1. The use of a computer is driven by real-life problems demanding fast and accurate solutions. Depending on the nature of the problems, the solutions may require different computing resources.

For numerical problems in science and technology, the solutions demand complex mathematical formulations and tedious integer or floating-point computations. For alphanumerical problems in business and government, the solutions demand accurate transactions, large database management, and information retrieval operations.

For artificial intelligence (AI) problems, the solutions demand logic inferences and symbolic manipulations. These computing problems have been labeled *numerical computing*, *transaction processing*, and *logical reasoning*. Some complex problems may demand a combination of these processing modes.



Hardware Resources The system architecture of a computer is represented by three nested circles on the right in Fig. 1.1. A modern computer system demonstrates its power through coordinated efforts by hardware resources, an operating system, and application software. Processors, memory, and peripheral devices form the hardware core of a computer system. We will study instruction-set processors, memory organization, multiprocessors, supercomputers, multicomputers, and massively parallel computers.

Special hardware interfaces are often built into I/O devices, such as terminals, workstations, optical page scanners, magnetic ink character recognizers, modems, file servers, voice data entry, printers, and plotters. These peripherals are connected to mainframe computers directly or through local or wide-area networks.

Operating System An effective operating system manages the allocation and deallocation of resources during the execution of user programs. We will study UNIX extensions for multiprocessors and multicomputers in Chapter 12. Mach/OS kernel and OSF/1 will be specially studied for multithreaded kernel functions, virtual memory management, file subsystems, and network communication services. Beyond the OS, application software must be developed to benefit the users. Standard benchmark programs are needed for performance evaluation.

Mapping is a bidirectional process matching algorithmic structure with hardware architecture, and vice versa. Efficient mapping will benefit the programmer and produce better source codes. The mapping of algorithmic and data structures onto the machine architecture includes processor scheduling, memory maps, interprocessor communications, etc. These activities are usually architecture-dependent.

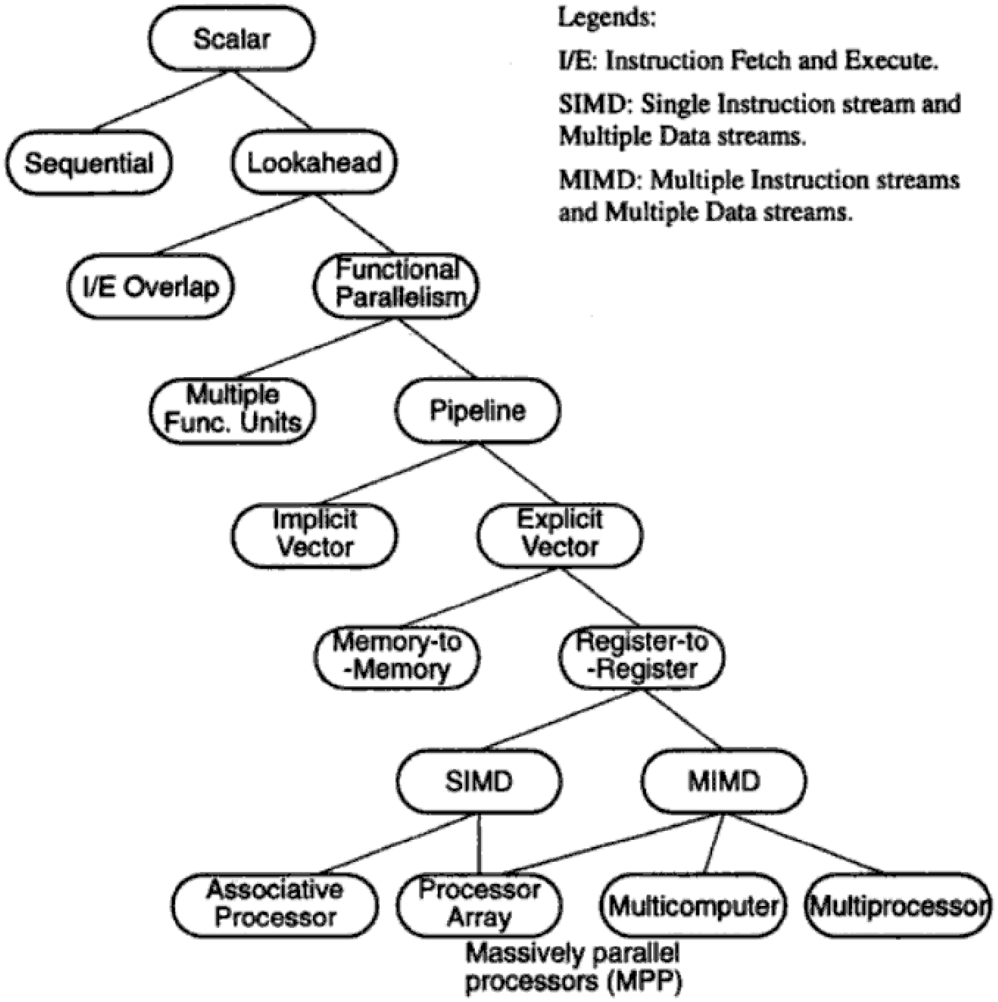
System Software Support Software support is needed for the development of efficient programs in high-level languages. The source code written in a HLL must be first translated into object code by an optimizing compiler. The *compiler* assigns variables to registers or to memory words and reserves functional units for operators. An *assembler* is used to translate the compiled object code into machine code which can be recognized by the machine hardware. A *loader* is used to initiate the program execution through the OS kernel.

Compiler Support There are three compiler upgrade approaches: *preprocessor*, *precompiler*, and *parallelizing compiler*. A preprocessor uses a sequential compiler and a low-level library of the target computer to implement high-level parallel constructs. The precompiler approach requires some program flow analysis, dependence checking, and limited optimizations toward parallelism detection. The third approach demands a fully developed parallelizing or vectorizing compiler which can automatically detect parallelism in source code and transform sequential codes into parallel constructs. These

Evolution of Computer Architecture

The study of computer architecture involves both hardware organization and programming/software requirements. As seen by an assembly language programmer, computer architecture is abstracted by its instruction set, which includes opcode (operation codes), addressing modes, registers, virtual memory, etc.

From the hardware implementation point of view, the abstract machine is organized with CPUs, caches, buses, microcode, pipelines, physical memory, etc. Therefore, the study of architecture covers both instruction-set architectures and machine implementation organizations.



Legends:

I/E: Instruction Fetch and Execute.

SIMD: Single Instruction stream and Multiple Data streams.

MIMD: Multiple Instruction streams and Multiple Data streams.

Lookahead, Parallelism, and Pipelining Lookahead techniques were introduced to prefetch instructions in order to overlap I/E (instruction fetch/decode and execution) operations and to enable functional parallelism. Functional parallelism was supported by two approaches: One is to use multiple functional units simultaneously, and the other is to practice pipelining at various processing levels.

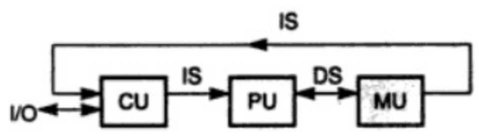
Flynn's Classification Michael Flynn (1972) introduced a classification of various computer architectures based on notions of instruction and data streams. As illustrated in Fig. 1.3a, conventional sequential machines are called SISD (*single instruction stream over a single data stream*) computers. Vector computers are equipped with scalar and vector hardware or appear as SIMD (*single instruction stream over multiple data streams*) machines (Fig. 1.3b). Parallel computers are reserved for MIMD (*multiple instruction streams over multiple data streams*) machines.

An MISD (*multiple instruction streams and a single data stream*) machines are modeled in Fig. 1.3d. The same data stream flows through a linear array of processors executing different instruction streams. This architecture is also known as *systolic arrays* (Kung and Leiserson, 1978) for pipelined execution of specific algorithms.

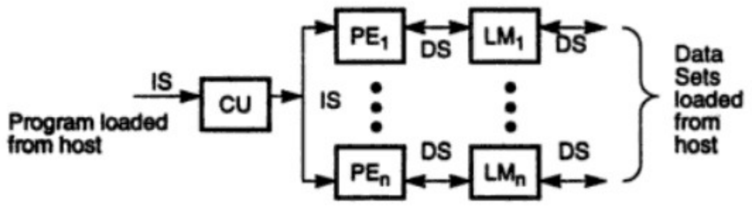
Of the four machine models, most parallel computers built in the past assumed the MIMD model for general-purpose computations. The SIMD and MISD models are more suitable for special-purpose computations. For this reason, MIMD is the most popular model, SIMD next, and MISD the least popular model being applied in commercial machines.

Parallel/Vector Computers Intrinsic parallel computers are those that execute programs in MIMD mode. There are two major classes of parallel computers, namely, *shared-memory multiprocessors* and *message-passing multicomputers*. The major distinction between multiprocessors and multicomputers lies in memory sharing and the mechanisms used for interprocessor communication.

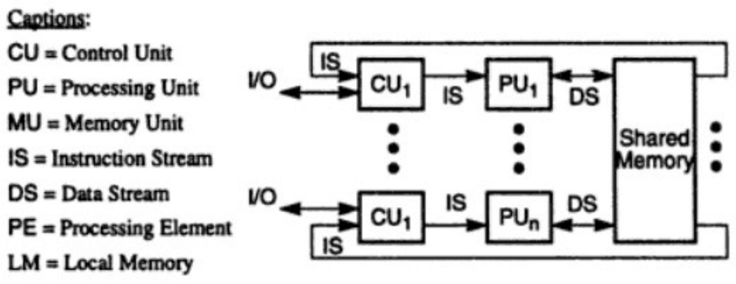
The processors in a multiprocessor system communicate with each other through *shared variables* in a common memory. Each computer node in a multicomputer system has a local memory, unshared with other nodes. Interprocessor communication is done through *message passing* among the nodes.



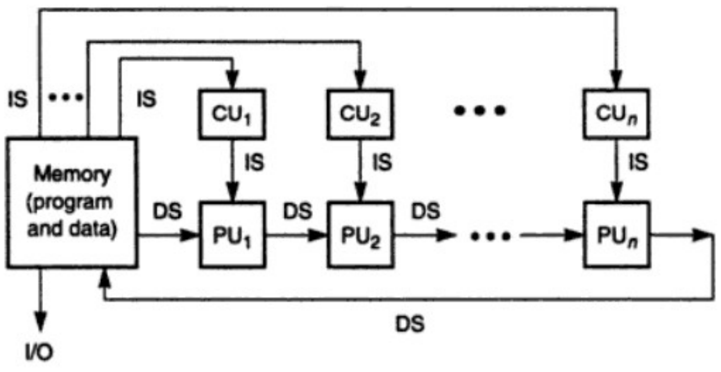
(a) SISD uniprocessor architecture



(b) SIMD architecture (with distributed memory)



(c) MIMD architecture (with shared memory)



(d) MISD architecture (the systolic array)

Clock Rate and CPI The CPU (or simply the *processor*) of today's digital computer is driven by a clock with a constant *cycle time* (τ in nanoseconds). The inverse of the cycle time is the *clock rate* ($f = 1/\tau$ in megahertz). The size of a program is determined by its *instruction count* (I_c), in terms of the number of machine instructions to be executed in the program. Different machine instructions may require different numbers of clock cycles to execute. Therefore, the *cycles per instruction* (CPI) becomes an important parameter for measuring the time needed to execute each instruction.

Performance Factors Let I_c be the number of instructions in a given program, or the instruction count. The CPU time (T in seconds/program) needed to execute the program is estimated by finding the product of three contributing factors:

$$T = I_c \times \text{CPI} \times \tau \tag{1.1}$$

The CPI of an instruction type can be divided into two component terms corresponding to the total processor cycles and memory cycles needed to complete the execution of the instruction. Depending on the instruction type, the complete instruction cycle may involve one to four memory references (one for instruction fetch, two for operand fetch, and one for store results). Therefore we can rewrite Eq. 1.1 as follows:

$$T = I_c \times (p + m \times k) \times \tau \tag{1.2}$$

where p is the number of processor cycles needed for the instruction decode and execution, m is the number of memory references needed, k is the ratio between memory cycle and processor cycle, I_c is the instruction count, and τ is the processor cycle time. Equation 1.2 can be further refined once the CPI components (p, m, k) are weighted over the entire instruction set.

System Attributes The above five performance factors (I_c, p, m, k, τ) are influenced by four system attributes: instruction-set architecture, compiler technology, CPU implementation and control, and cache and memory hierarchy, as specified in Table 1.2.

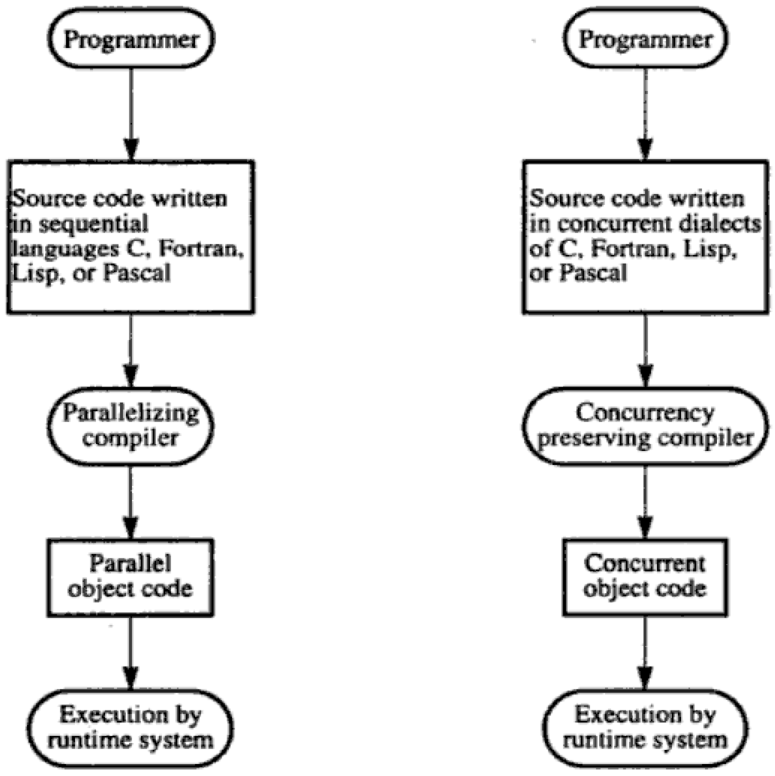
The instruction-set architecture affects the program length (I_c) and processor cycle needed (p). The compiler technology affects the values of I_c, p, and the memory reference count (m). The CPU implementation and control determine the total processor time $(p \cdot \tau)$ needed. Finally, the memory technology and hierarchy design affect the memory access latency $(k \cdot \tau)$. The above CPU time can be used as a basis in estimating the execution rate of a processor.

MIPS Rate Let C be the total number of clock cycles needed to execute a given program. Then the CPU time in Eq. 1.2 can be estimated as $T = C \times \tau = C/f$. Furthermore, $\text{CPI} = C/I_c$ and $T = I_c \times \text{CPI} \times \tau = I_c \times \text{CPI}/f$. The processor speed is often measured in terms of *million instructions per second* (MIPS). We simply call it the MIPS rate of a given processor. It should be emphasized that the MIPS rate varies with respect to a number of factors, including the clock rate (f), the instruction count (I_c), and the CPI of a given machine, as defined below:

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

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

Implicit Parallelism An implicit approach uses a conventional language, such as C, Fortran, Lisp, or Pascal, to write the source program. The sequentially coded source program is translated into parallel object code by a parallelizing compiler. As illustrated in Fig. 1.5a, this compiler must be able to detect parallelism and assign target machine resources. This compiler approach has been applied in programming shared-memory multiprocessors.


Programmer
Source code written in sequential languages C, Fortran, Lisp, or Pascal
Parallelizing compiler
Parallel object code
Execution by runtime system
Programmer
Source code written in concurrent dialects of C, Fortran, Lisp, or Pascal
Concurrency preserving compiler
Concurrent object code
Execution by runtime system

(a) Implicit parallelism (b) Explicit parallelism

Explicit Parallelism The second approach (Fig. 1.5b) requires more effort by the programmer to develop a source program using parallel dialects of C, Fortran, Lisp, or Pascal. Parallelism is explicitly specified in the user programs. This will significantly reduce the burden on the compiler to detect parallelism. Instead, the compiler needs to preserve parallelism and, where possible, assigns target machine resources. Charles Seitz of California Institute of Technology and William Dally of Massachusetts Institute of Technology adopted this explicit approach in multicomputer development.

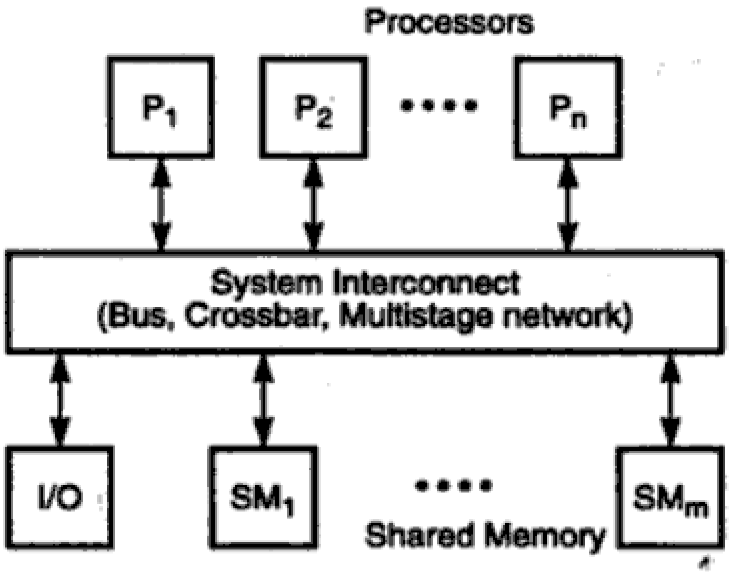
Multiprocessors and Multicomputers

Two categories of parallel computers are architecturally modeled below. These physical models are distinguished by having a shared common memory or unshared distributed memories. Only architectural organization models are described in Sections

Shared-Memory Multiprocessors

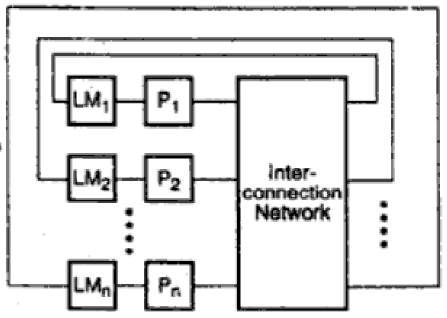
The UMA Model In a UMA multiprocessor model (Fig. 1.6), the physical memory is uniformly shared by all the processors. All processors have equal access time to all memory words, which is why it is called uniform memory access. Each processor may use a private cache. Peripherals are also shared in some fashion.

Multiprocessors are called *tightly coupled systems* due to the high degree of resource sharing. The system interconnect takes the form of a common bus, a crossbar switch,

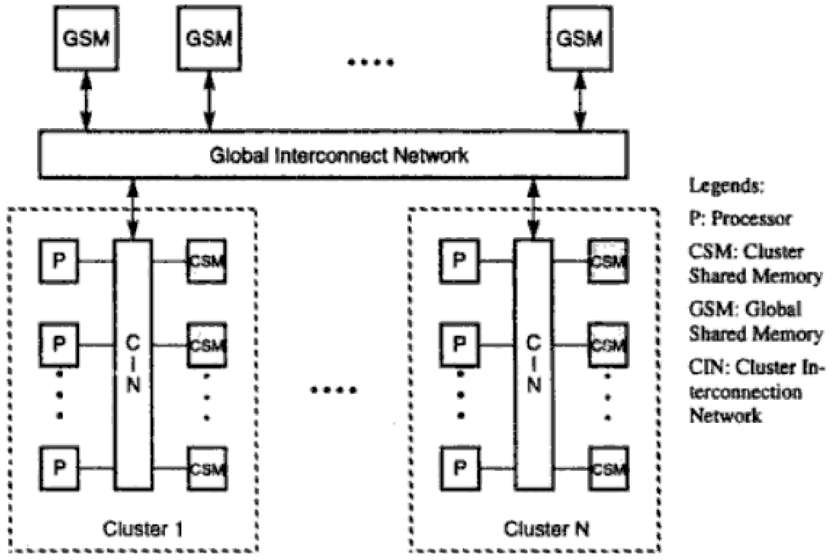


The NUMA Model A NUMA multiprocessor is a shared-memory system in which the access time varies with the location of the memory word. Two NUMA machine models are depicted in Fig. 1.7. The shared memory is physically distributed to all processors, called *local memories*. The collection of all local memories forms a global address space accessible by all processors.

It is faster to access a local memory with a local processor. The access of remote memory attached to other processors takes longer due to the added delay through the interconnection network. The BBN TC-2000 Butterfly multiprocessor assumes the configuration shown in Fig. 1.7a.



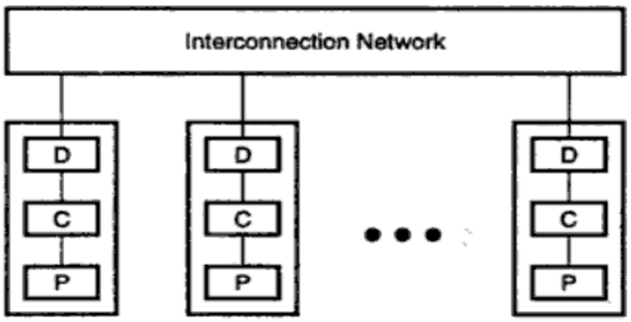
(a) Shared local memories (e.g., the BBN Butterfly)


GSM
GSM
GSM
Global Interconnect Network
P
CSM
CIN
Cluster 1
Cluster N
Legends:
P: Processor
CSM: Cluster Shared Memory
GSM: Global Shared Memory
CIN: Cluster Interconnection Network

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

The COMA Model A multiprocessor using cache-only memory assumes the COMA model. Examples of COMA machines include the Swedish Institute of Computer Science's Data Diffusion Machine (DDM, Hagersten et al., 1990) and Kendall Square Research's KSR-1 machine (Burkhardt et al., 1992). The COMA model is depicted in

The COMA model is a special case of a NUMA machine, in which the distributed main memories are converted to caches. There is no memory hierarchy at each processor node. All the caches form a global address space. Remote cache access is assisted by the distributed cache directories (D in Fig. 1.8). Depending on the interconnection network used, sometimes hierarchical directories may be used to help locate copies of cache blocks. Initial data placement is not critical because data will eventually migrate to where it will be used.

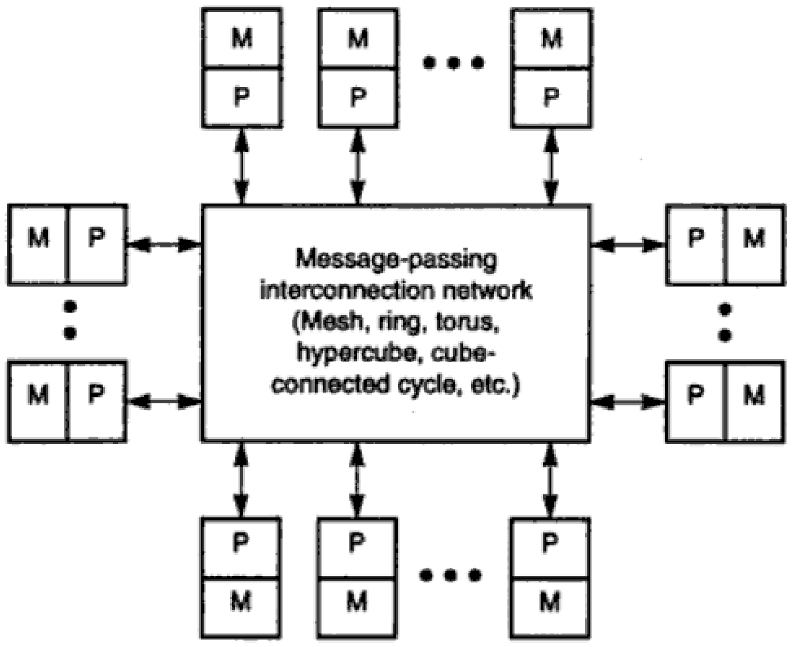


The COMA model of a multiprocessor. (P: Processor, C: Cache, D: Directory; e.g., the KSR-1)

Distributed-Memory Multicomputers

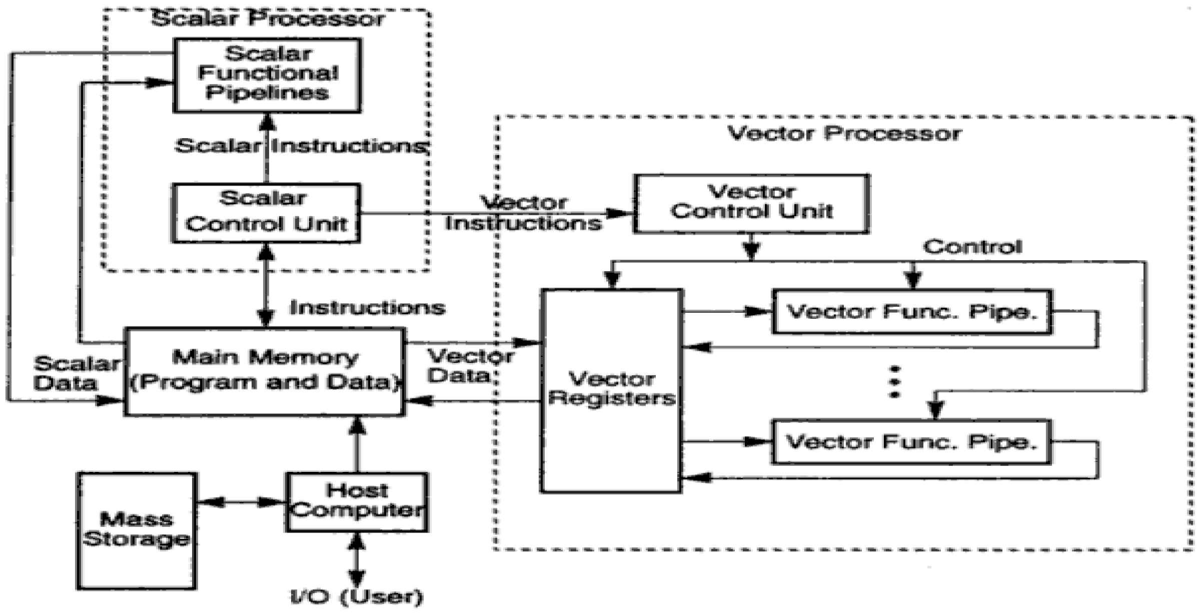
A distributed-memory multicomputer system is modeled in Fig. 1.9. The system consists of multiple computers, often called *nodes*, interconnected by a message-passing network. Each node is an autonomous computer consisting of a processor, local memory, and sometimes attached disks or I/O peripherals.

The message-passing network provides point-to-point static connections among the nodes. All local memories are private and are accessible only by local processors. For this reason, traditional multicomputers have been called *no-remote-memory-access* (NORMA) machines. However, this restriction will gradually be removed in future multicomputers with distributed shared memories. Internode communication is carried out by passing messages through the static connection network.


M
P
Message-passing interconnection network (Mesh, ring, torus, hypercube, cube-connected cycle, etc.)

Vector Processor Models Figure 1.11 shows a *register-to-register* architecture. Vector registers are used to hold the vector operands, intermediate and final vector results. The vector functional pipelines retrieve operands from and put results into the vector registers. All vector registers are programmable in user instructions. Each vector register is equipped with a component counter which keeps track of the component registers used in successive pipeline cycles.

The length of each vector register is usually fixed, say, sixty-four 64-bit component registers in a vector register in a Cray Series supercomputer. Other machines, like the Fujitsu VP2000 Series, use reconfigurable vector registers to dynamically match the register length with that of the vector operands.



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

$$M = \langle N, C, I, M, R \rangle \tag{1.5}$$

where

(1) N is the number of *processing elements* (PEs) in the machine. For example, the Illiac IV has 64 PEs and the Connection Machine CM-2 uses 65,536 PEs.

(2) C is the set of instructions directly executed by the *control unit* (CU), including scalar and program flow control instructions.

(3) I is the set of instructions broadcast by the CU to all PEs for parallel execution. These include arithmetic, logic, data routing, masking, and other local operations executed by each active PE over data within that PE.

(4) M is the set of masking schemes, where each mask partitions the set of PEs into enabled and disabled subsets.

(5) R is the set of data-routing functions, specifying various patterns to be set up in the interconnection network for inter-PE communications.

UNIT II

Program and Network Properties

The exploitation of parallelism has created a new dimension in computer science. In order to move parallel processing into the mainstream of computing, H.T. Kung (1991) has identified the need to make significant progress in three key areas: *computation models* for parallel computing, *interprocessor communication* in parallel architectures, and *system integration* for incorporating parallel systems into general computing environments.

(1) *Flow dependence*: A statement S2 is *flow-dependent* on statement S1 if an execution path exists from S1 to S2 and if at least one output (variables assigned) of S1 feeds in as input (operands to be used) to S2. Flow dependence is denoted as S1 $\rightarrow$ S2.

(2) *Antidependence*: Statement S2 is *antidependent* on statement S1 if S2 follows S1 in program order and if the output of S2 overlaps the input to S1. A direct arrow crossed with a bar as in S1 $\not\rightarrow$ S2 dicates antidependence from S1 to S2.

(3) *Output dependence*: Two statements are *output-dependent* if they produce (write) the same output variable. S1 $\circ\!\!\rightarrow$ S2 indicates output dependence from S1 to S2.

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

(5) *Unknown dependence*: The dependence relation between two statements cannot be determined in the following situations:

- The subscript of a variable is itself subscribed.
- The subscript does not contain the loop index variable.
- A variable appears more than once with subscripts having different coefficients of the loop variable.
- The subscript is nonlinear in the loop index variable.

Control Dependence This refers to the situation where the order of execution of statements cannot be determined before run time. For example, conditional statements will not be resolved until run time. Different paths taken after a conditional branch may introduce or eliminate data dependence among instructions. Dependence may also exist between operations performed in successive iterations of a looping procedure. In the following, we show one loop example with and another without control-dependent iterations. The successive iterations of the following loop are *control-independent*:

Bernstein's Conditions In 1966, Bernstein revealed a set of conditions based on which two processes can execute in parallel. A *process* is a software entity corresponding to the abstraction of a program fragment defined at various processing levels. We define the *input set* I_i of a process P_i as the set of all input variables needed to execute the process.

Similarly, the *output set* O_i consists of all output variables generated after execution of the process P_i. Input variables are essentially operands which can be fetched from memory or registers, and output variables are the results to be stored in working registers or memory locations.

Now, consider two processes P_1 and P_2 with their input sets I_1 and I_2 and output sets O_1 and O_2, respectively. These two processes can execute in parallel and are denoted $P_1 \parallel P_2$ if they are independent and therefore create deterministic results.

Formally, these conditions are stated as follows:

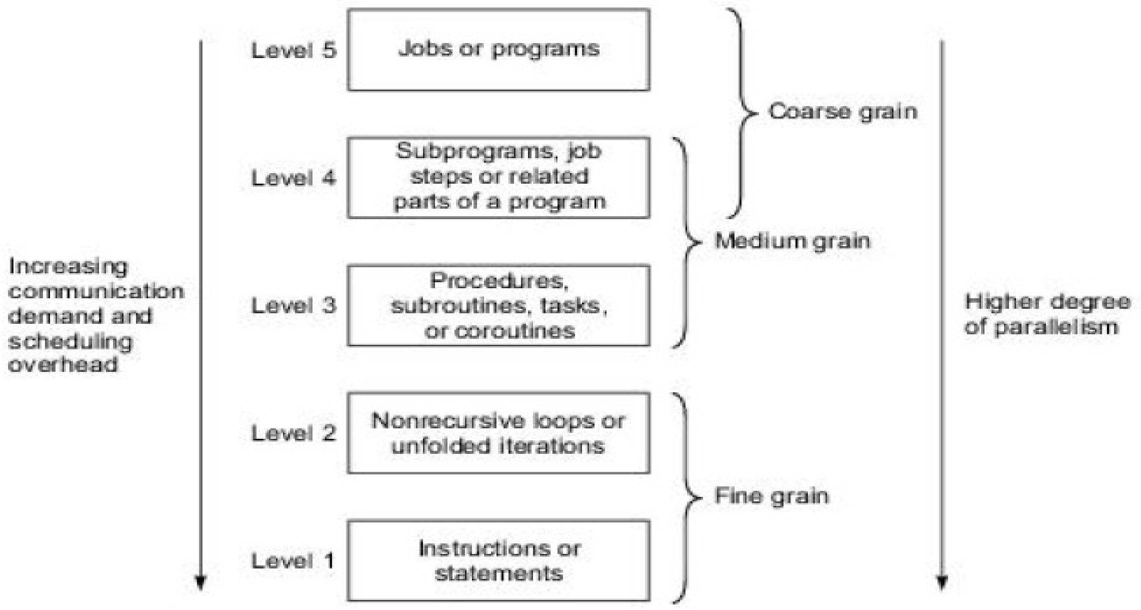
$$\left.\begin{aligned} I_1 \cap O_2 &= \phi \\ I_2 \cap O_1 &= \phi \\ O_1 \cap O_2 &= \phi \end{aligned}\right\} \tag{2.1}$$

Grain size or *granularity* is a measure of the amount of computation involved in a software process. The simplest measure is to count the number of instructions in a grain (program segment). Grain size determines the basic program segment chosen for parallel processing. Grain sizes are commonly described as *fine, medium,* or *coarse*, depending on the processing levels involved.

Latency is a time measure of the communication overhead incurred between machine subsystems. For example, the *memory latency* is the time required by a processor to access the memory. The time required for two processes to synchronize with each other is called the *synchronization latency*. Computational granularity and communication latency are closely related, as we shall see below.

Parallelism has been exploited at various processing levels. As illustrated in Fig. 2.5, five levels of program execution represent different computational grain sizes and changing communication and control requirements. The lower the level, the finer the granularity of the software processes.

In general, the execution of a program may involve a combination of these levels. The actual combination depends on the application, formulation, algorithm, language, program, compilation support, and hardware characteristics. We characterize below the parallelism levels and review their implementation issues from the viewpoints of a programmer and of a compiler writer.


Level 5
Jobs or programs
Level 4
Subprograms, job steps or related parts of a program
Level 3
Procedures, subroutines, tasks, or coroutines
Level 2
Nonrecursive loops or unfolded iterations
Level 1
Instructions or statements
Coarse grain
Medium grain
Fine grain
Increasing communication demand and scheduling overhead
Higher degree of parallelism

Instruction Level At the lowest level, a typical grain contains less than 20 instructions, called *fine grain* in Fig. 2.5. Depending on individual programs, fine-grain parallelism at this level may range from two to thousands. Butler et al. (1991) has shown that single-instruction-stream parallelism is greater than two. Wall (1991) finds that the average parallelism at instruction level is around five, rarely exceeding seven, in an ordinary program. For scientific applications, Kumar (1988) has measured the average parallelism in the range of 500 to 3000 Fortran statements executing concurrently in an idealized environment.

Loop Level This corresponds to the iterative loop operations. A typical loop contains less than 500 instructions. Some loop operations, if independent in successive iterations, can be vectorized for pipelined execution or for lock-step execution on SIMD machines. Some loop operations can be self-scheduled for parallel execution on MIMD machines.

Procedure Level This level corresponds to medium-grain parallelism at the task, procedural, subroutine, and coroutine levels. A typical grain at this level contains less than 2000 instructions. Detection of parallelism at this level is much more difficult than at the finer-grain levels. Interprocedural dependence analysis is much more involved and history-sensitive.

Communication requirement is often less compared with that required in MIMD execution mode. SPMD execution mode is a special case at this level. Multitasking also belongs in this category. Significant efforts by programmers may be needed to restructure a program at this level, and some compiler assistance is also needed.

Subprogram Level This corresponds to the level of job steps and related subprograms. The grain size may typically contain tens or hundreds of thousands of instructions. Job steps can overlap across different jobs. Subprograms can be scheduled for different processors in SPMD or MPMD mode, often on message-passing multicomputers.

Job (Program) Level This corresponds to the parallel execution of essentially independent jobs (programs) on a parallel computer. The grain size can be as high as millions of instructions in a single program. For supercomputers with a small number of very powerful processors, such coarse-grain parallelism is practical. Job-level parallelism is handled by the program loader and by the operating system in general. Time-sharing or space-sharing multiprocessors explore this level of parallelism. In fact, both time and space sharing are extensions of multiprogramming.

2.2.2 Grain Packing and Scheduling

Two fundamental questions to ask in parallel programming are: (i) How can we partition a program into parallel branches, program modules, microtasks, or grains to yield the shortest possible execution time? and (ii) What is the optimal size of concurrent grains in a computation?

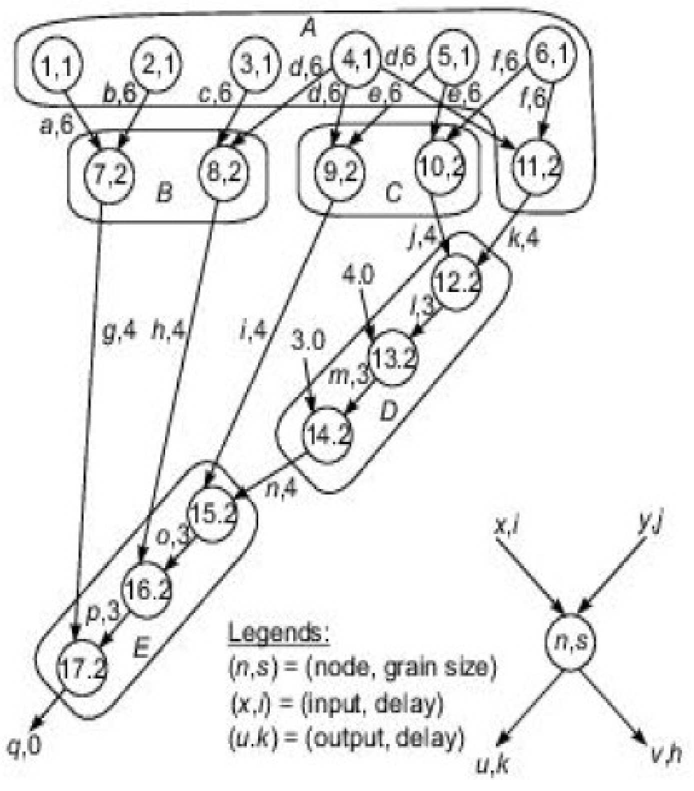
This grain-size problem demands determination of both the number and the size of grains (or microtasks) in a parallel program. Of course, the solution is both problem-dependent and machine-dependent. The goal is to produce a short schedule for fast execution of subdivided program modules.

There exists a tradeoff between parallelism and scheduling/synchronization overhead. The time complexity involves both computation and communication overheads. The program partitioning involves the algorithm designer, programmer, compiler, operating system support, etc. We describe below a *grain packing* approach introduced by Kruatrachue and Lewis (1988) for parallel programming applications.

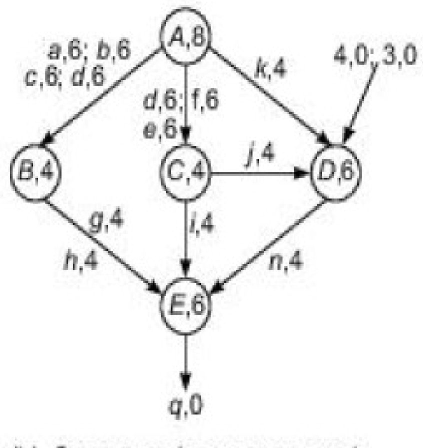
The basic concept of program partitioning is introduced below. In Fig. 2.6, we show an example *program graph* in two different grain sizes. A program graph shows the structure of a program. It is very similar to the dependence graph introduced in Section 2.1.1. Each node in the program graph corresponds to a computational unit in the program. The *grain size* is measured by the number of basic machine cycles (including both processor and memory cycles) needed to execute all the operations within the node.

We denote each node in Fig. 2.6 by a pair (n, s), where n is the *node name* (id) and s is the grain size of the node. Thus grain size reflects the number of computations involved in a program segment. Fine-grain nodes have a smaller grain size, and coarse-grain nodes have a larger grain size.

```
Var a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q
Begin
    1. a := 1               10. j := e × f
    2. b := 2               11. k := d × f
    3. c := 3               12. l := j × k
    4. d := 4               13. m := 4 × l
    5. e := 5               14. n := 3 × m
    6. f := 6               15. o := n × i
    7. g := a × b           16. p := o × h
    8. h := c × d           17. q := p × q
    9. i := d × e
End
```



(a) Fine-grain program graph before packing



(b) Coarse-grain program graph after packing

PROGRAM FLOW MECHANISMS

Conventional computers are based on a control flow mechanism by which the order of program execution is explicitly stated in the user programs. Dataflow computers are based on a datadriven mechanism which allows the execution of any instruction to be driven by data (operand) availability. Dataflow computers emphasize a high degree of parallelism at the fine-grain instructional level. Reduction computers are based on a demand-driven mechanism which initiates an operation based on the demand for its results by other computations.

Control Flow Versus Data Flow

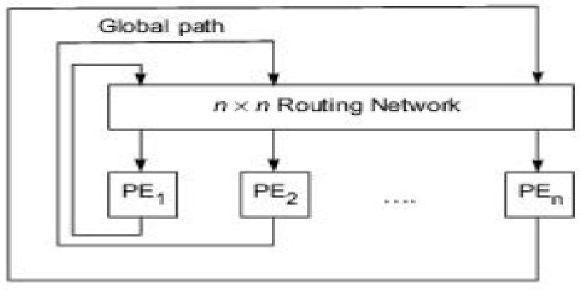
Conventional von Neumann computers use a program counter (PC) to sequence the execution of instructions in a program. The PC is sequenced by instruction flow in a program. This sequential execution style has been called control-driven, as program flow is explicitly controlled by programmers.

A uniprocessor computer is inherently sequential, due to use of the control driven mechanism. However, control flow can be made parallel by using parallel language constructs or parallel compilers. In this book, we study primarily parallel control-flow computers and their programming techniques. Until the data-driven or demand-driven mechanism is proven to be cost-effective, the control-flow approach will continue to dominate the computer industry.

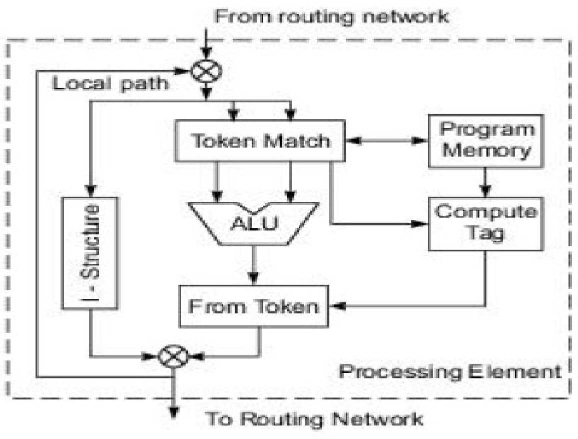
In a dataflow computer, the execution of an instruction is driven by data availability instead of being guided by a program counter. In theory, any instruction should be ready for execution whenever operands become available. The instructions in a data-driven program arc not ordered in any way. Instead of being stored separately in a main memory, data are directly held inside instructions.

Computational results (data tokens) are passed directly between instructions. The data generated by an instruction will be duplicated into many copies and forwarded directly to all needy instructions. Data tokens, once consumed by an instruction, will no longer be available for reuse by other instructions. This data-driven scheme requires no program counter, and no control sequencer. However, it requires special mechanisms to detect data availability, to match data tokens with needy instructions, and to enable the chain reaction of asynchronous instruction executions. No memory sharing between instructions results in no side effects.

The global architecture consists ofn processing elements (PEs) interconnected by an n x n muting network. The entire system supports pipclincd dataflow operations in all n PEs. Inter-PE communications arc done through the pipelincd routing network.



(a) The global architecture



(b) Interior design of a processing element

Within each PE, the machine provides a low-level token matching mechanism which dispatches onlythose instructions whose input data [tokens] are already available. Each datum is tagged with the address of thc instruction to which it belongs and the context in which thc instntction is being executed. Instructions are stored in the program memory. Tagged tokens enter the PE through a local path. The tokens can also be passed to other PEs through the routing network. All internal token circulation operations are pipclincd without blocking. It is the machine’s job to match up data with the same tag to needy instructions. In so doing, new data will be produced with a new tag indicating the successor instructiontsl. Thus, each instruction represents a synchronization operation. New tokens are formed and circulated along the PE pipeline for reuse or to other PEs through the global path, which is also pipelined.

NETWORK PROPERTIES & ROUTING

The topology of an interconnection network can be either static or dynamic. Static networks are formedof point-to-point direct connections which will not change during program execution. Dynamic networks are implemented with switched channels, which are dynamically configured to match the communicationdemand in user programs. Packet switching and routing is playing an important role in modern multiprocessor architecture.

Node Degree and Network Diameter : The number of edges {links or channels) incident on a node is called the node degree d. In the case of unidirectional channels, thc number of channcls into a node is the in degree, and that out of a node is thc out degree. Then thc node degree is thc sum ofthe two. Thc node degree reflects the number of IO ports required per node, and thus the cost of a node. Therefore, the node degree should be kept a (small) constant, in order to reduce oost.

The Diameter D of a network is the maximum shortest path between any two nodes. The path length is measured by the number of links traversed. The network diameter indicates the maximum number of distinct hops between any two nodes, thus providing a figure of communication merit for the network. Therefore, the network diameter should be as small as possible from a communication point of vicw.

Bisection Width: When a given network is cut into two equal halves, the minimum number of edges {channels} along thc cut is called thc bisection width b. In the case of a communication network, each edge may correspond to a channel‘ with w bit wires.

To summarize the above discussions, the performance of an interconnection network is affected by the following factors:

Functionality: refers to how the network supports data routing, interrupt handling, synchronization, request-"message combining, and coherence.

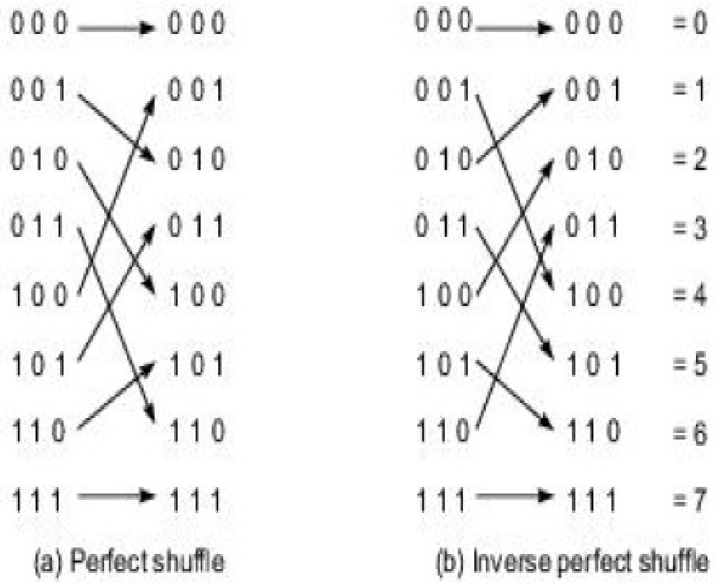
Network Latency:-—This refers to the worst-ease time delay for a unit message to be transferred through the network.

Bandwidth This refers to the maximum data transfer rate, in terms of Mbps or Gbps transmitted through the network

Hardware Complexity'—This refers to implementation costs such as those for wires, switches, connectors, arbitration, and interface logic.

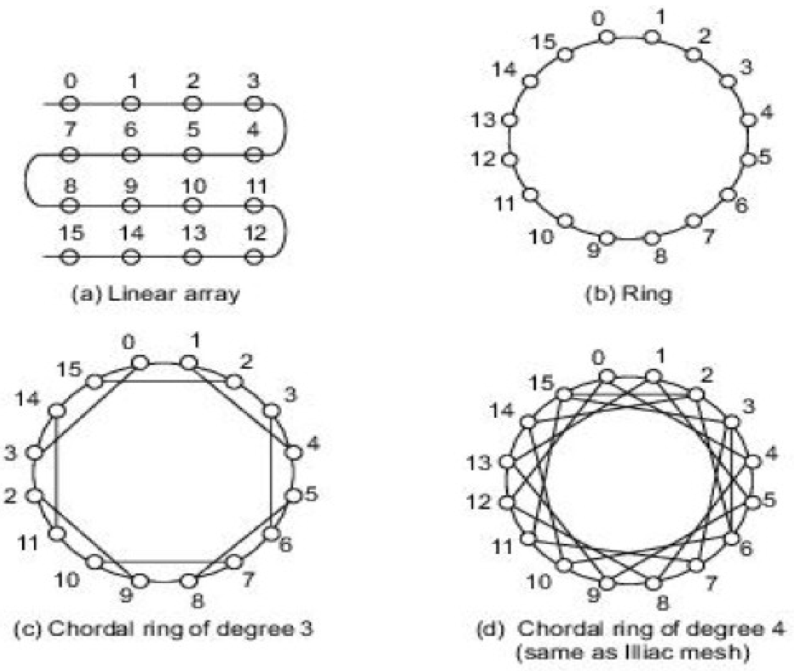
Scalability—This refers to the ability ofa network to be modularly expandable with a scalable performance with increasing machine resources.

Perfect Shuffle and Exchange Perfect shuffle is a special permutation function suggested by Harold Stone (1971) for parallel processing applications. The mapping corresponding to a perfect shuffle is shown in Fig. 2.14a. Its inverse is shown on the right-hand side (Fig. 2.14b).



STATIC CONNECTION NETWORKS:

Static networks use direct links which are fixed once built. This type of network is more suitable for building computers where the communication patterns are predictable or implementable with static connections. We describe their topologies below in terms of network parameters and comment on their relative merits in relation to communication and scalability.



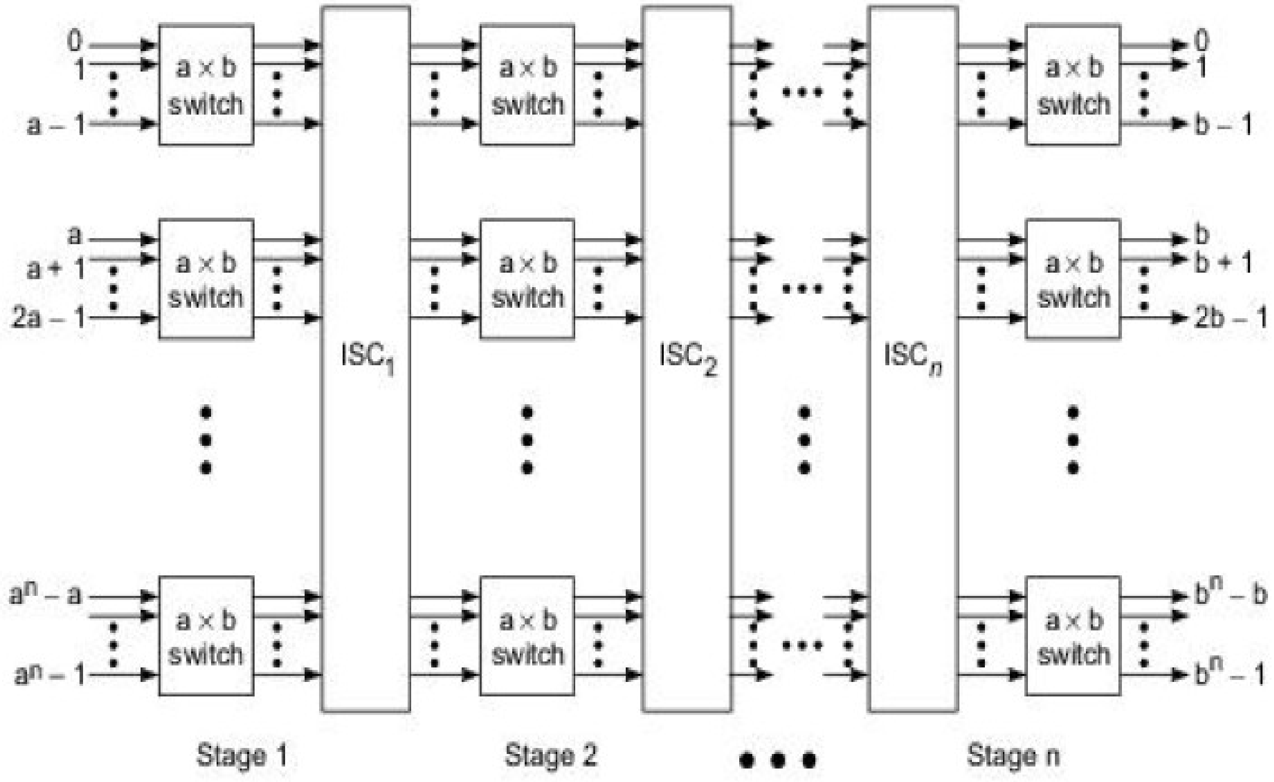
(a) Linear array
(b) Ring
(c) Chordal ring of degree 3
(d) Chordal ring of degree 4 (same as Illiac mesh)

Dynamic Connection Networks

Switch Modules An $a \times b$ *switch module* has a inputs and b outputs. A *binary switch* corresponds to a 2×2 switch module in which $a = b = 2$. In theory, a and b do not have to be equal. However, in practice, a and b are often chosen as integer powers of 2; that is, $a = b = 2^k$ for some $k \geq 1$.

Multistage Interconnection Networks MINs have been used in both MIMD and SIMD computers. A generalized multistage network is illustrated in Fig. 2.23. A number of $a \times b$ switches are used in each stage. Fixed interstage connections are used between the switches in adjacent stages. The switches can be dynamically set to establish the desired connections between the inputs and outputs.

Different classes of MINs differ in the switch modules used and in the kind of *interstage connection* (ISC) patterns used. The simplest switch module would be the 2×2 switches ($a = b = 2$ in Fig. 2.23). The ISC


0
1
a – 1
a
a + 1
2a – 1
a^n – a
a^n – 1
a × b switch
ISC1
ISC2
ISCn
0
1
b – 1
b
b + 1
2b – 1
b^n – b
b^n – 1
Stage 1
Stage 2
Stage n

UNIT III

PIPELINING

Linear PipelineProcessors

A linear pipeline processor is a cascade of processing stages which are linearlyconnected to perform a fixed function over a stream of data flowing from one end tothe other. In modern computers, linear pipelines are applied for instruction execution,arithmetic computation, and memory-access operations

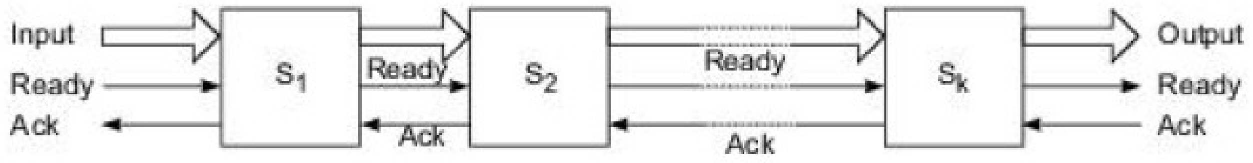
Asynchronous & Synchronous models

A linear pipeline processor is constructed with k processing stages.External inputs(operands) are fed into the pipeline at the first stage S1.The processed results are passed from stage Si ito stage Si+i,The final result emerges from thepipeline at the last stage Sn.Depending on the control of data flow along the pipeline, we model linear pipelinesin two categories: asynchronous and Synchronous.

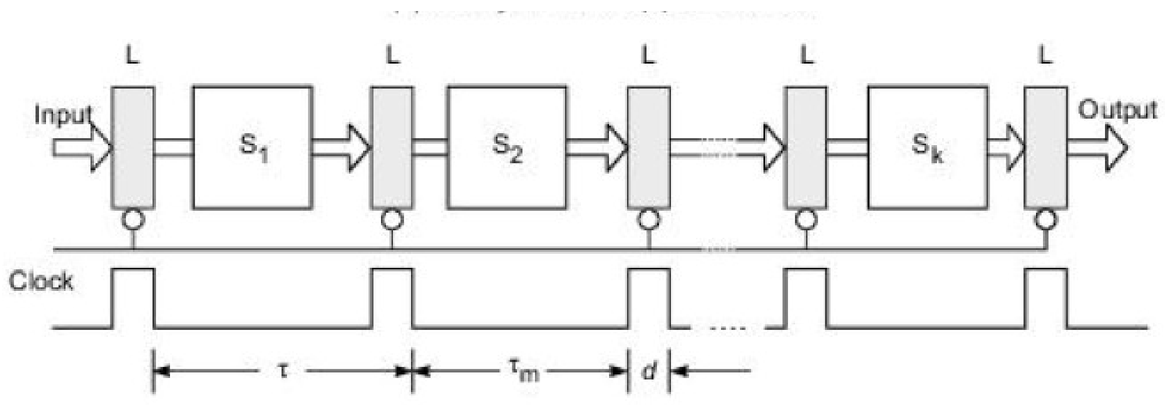
Asynchronous Model

As shown in the figure data flow between adjacent stages in an asynchronous pipeline is controlled by a handshaking protocol.When stage Si is ready to transmit, it sends a ready signal to stage Si+1. After stage receives the incoming data, it returns an acknowledge signal to Si. Asynchronous pipelines are useful in designing communication channels in message- passing multicomputers where pipelined wormhole routing is practiced Asynchronous pipelines may have a variable throughput rate.Different amounts of delay may be experienced in different stages.

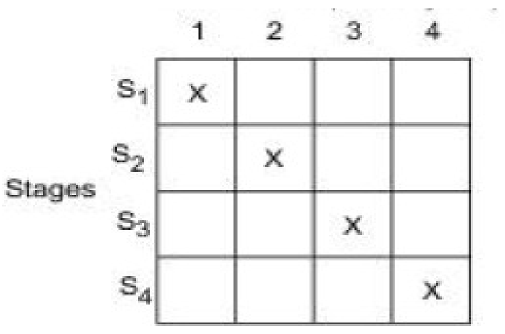
Synchronous Model : Synchronous pipelines are illustrated in Fig. Clocked latches are used to interface



between stages. The latches are made with master-slaveflip-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 approxi mately equal delays in all stages.These delays determine the clock period and thus thespeed of the pipeline. Unless otherwise specified, only synchronous pipelines are studied.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. .This table is essentially a space-time diagram depicting theprecedence relationship in using the pipeline stages. Successive tasks or operations are initiated one per cycle to enter the pipeline.Oncethe 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.



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

Clocking and Timing Control.

The clock cycle of a pipeline is determined below. Let T* be the time delay of thecircuitry in stage Si and d the time delay of a latch, as shown in Fig.

Clock Cycle and Throughput :Denote the maximum stage delay as Tm ,and we canwrite T as T = max{Ti} + d =Tm+d

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, T m »d for one to two orders of magnitude. This implies that the maximum stage delay Tm dominates theclock period.The pipeline frequency is defined as the inverse of the clock period

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 enteringthe pipeline, the actual throughput of the pipeline may be lower than f.This is becausemore 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 tmax be the time delayof the longest logic path within a stage and tm in that of the shortest logic path within a stage.

To avoid a race in two successive stages, we must choose Tm > tmax + s and d < tmn — s. These constraints translate into the following bounds on the clock periodwhen clock skew takes effect: d + tmax +s <= T<= Tm+tmin-s Tn the ideal case s = 0, tmax = Tm , and tmin = d. Thus, we have T=Tm+ d

Speedup, Throughput & Efficiency of Pipeline: Speedup is defined as

$$\text{Speedup} = \frac{\text{Time taken for a given computation by a non-pipelined functional unit}}{\text{Time taken for the same computation by a pipelined version}}$$

Assume a function of k stages of equal complexity which takes the same amount of time T. Non-pipelined function will take kT time for one input. Then Speedup = nkT/(k+n-1)T = nk/ (k+n-1)
Efficiency:It is an indicator of how efficiently the resources of the pipeline are used. If a stage is available during a clock period, then its availability becomes the unit of resource. Efficiency can be defined as

$$\text{Efficiency} = \frac{\text{Number of stage time units actuall used during computation}}{\text{Total number of stage time units available during that computation}}$$

No. of used stage time units = nk there are n inputs and each input uses k stages.

Total no. of stage-time units available = k[k + (n-1)]

It is the product of no. of stages in the pipeline (k) and no. of clock periods taken for computation(k+(n-1)).

No. of used stage time units = nk there are n inputs and each input uses k stages. Total no. of stage-time units available = k[k + (n-1)] It is the product of no. of stages in the pipeline (k) and no. of clock periods taken for computation(k+(n-1)).

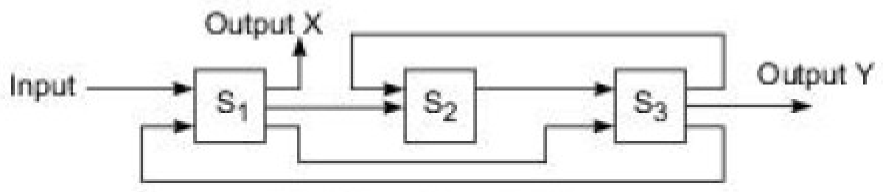
$$\text{Efficiency} = \frac{\text{nk}}{k[k+\text{n-1}]} = \frac{n}{k+n-1}$$

Throughput:It is the average number of results computed per unit time. For n inputs, a k-staged pipeline takes [k+(n-1)]T time units Then, Throughput = n / [k+n-1] T = nf / [k+n-1] where f is the clock frequency

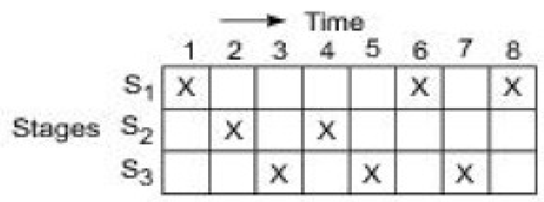
NON LINEAR PIPELINE PROCESSORS:

A dynamic pipeline can be reconfigured to perform variable functions at different times.The traditional linear pipelines are static pipelines because they are used toperform fixed functions.A dynamic pipeline allows feedforward and feedback connections in addition to the streamline connections.

Reservation and Latency analysis:In a static pipeline, it is 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 inaddition to streamline connections.A multifunction dynamic pipeline is shown in Fig. This pipeline has three stages. Besides the streamline connections from S1 to S2 and from S2 to S3, there is a feedforward connection from S1 to S3 and two feedback connections from S3 to S2 and from S3 to S1.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

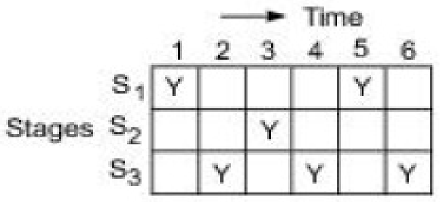


Reservation Tables : The reservation table for a static linear pipeline is trivial in the sense that data flow follows a linear streamline. The reservation table for a dynamicpipeline becomes more interesting because a nonlinear pattern is followed.Given a pipeline configuration, multiple reservation tables can be generated for the evaluation of different functions.Two reservation tables are given in Fig, corresponding to a function X and a function Y, respectively.


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

Each function evaluation is specified by one reservation table.A static pipeline is specified by a single reservation table .A dynamic pipelinemay be specified by more than one reservation table.Each reservation table displays the time-space flow of data through the pipeline forone function evaluation.Different functions may follow different paths on the reservation table.

A number of pipeline configurations may be represented by the same reservation table.There is a many-to-many mapping between various pipeline configurations and different reservation tables.The number of columns in a reservation table is called the evaluation time of a given function.

Latency Analysis The number of time units (clock cycles) between two initiations of a pipeline is the latency betweenthem.Latency values must be non negative 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

COLLISION FREE SCHEDULING: When scheduling events in a nonlinear pipeline, the main objective is to obtain the shortest average latency between initiations witliout causing collisions. In what follows, we present a systematic method for achieving such collision-free scheduling.

Collision Vector: 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 in m<=n-1. The permisable latency p should be as small as possible. The choice is made in the range 1 <= p <= m-1. A permissible latency of p = I 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.

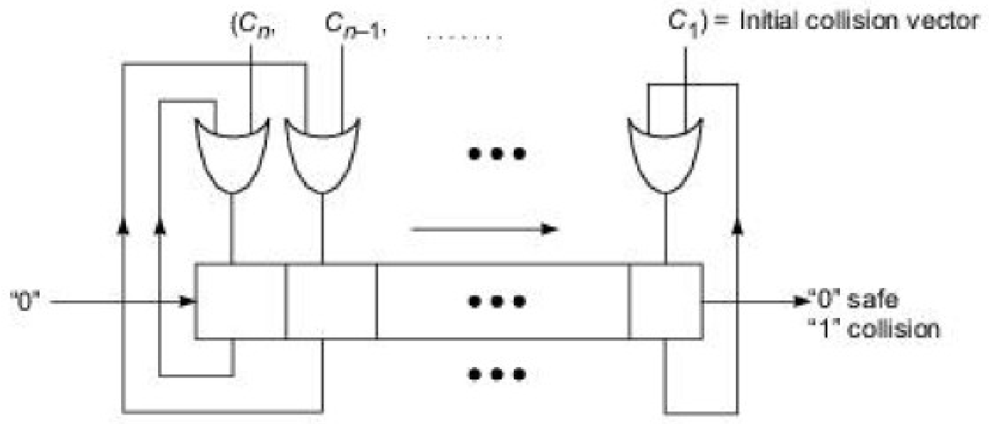
The combined set of permissible and forbidden latencies can be easily displayed by a collision vector, which is an m-bit binary vector $C = (C_m C_{m-1} \ldots C_2 C_1)$. The value of $C_i = 1$ if latency i causes a collision and $C_i = 0$ if latency i is permissible. Note that it is always true that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

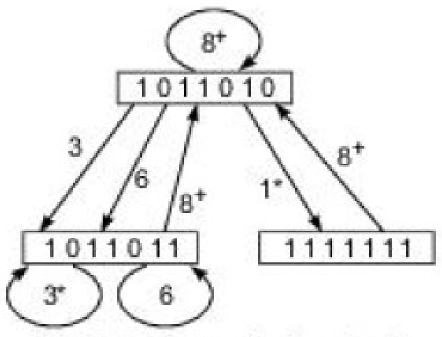
State Diagrams From the above collision vector, one can construct a *state diagram* specifying the permissible state transitions among successive initiations. The collision vector, like C_X above, corresponds to the *initial state* of the pipeline at time 1 and thus is called an *initial collision vector*. 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.

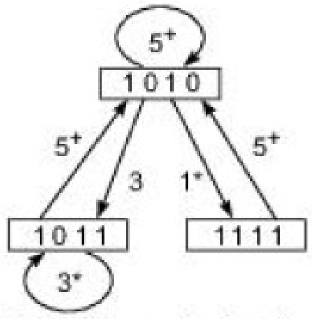
Logical 0 enters from the left end of the shift register. The next state after p shifts is thus obtained by bitwise-ORing the initial collision vector with the shifted register contents. For example, from the initial state $C_X = (1011010)$, the next state (1111111) is reached after one right shift of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



(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

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.

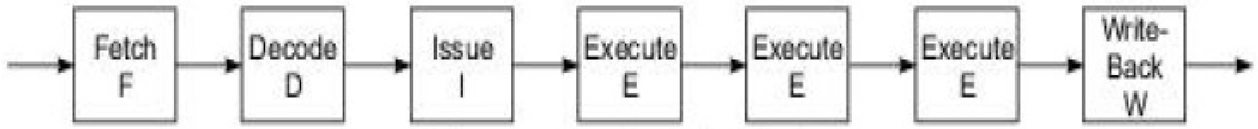
UNIT IV INSTRUCTION PIPELINE DESIGN

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.

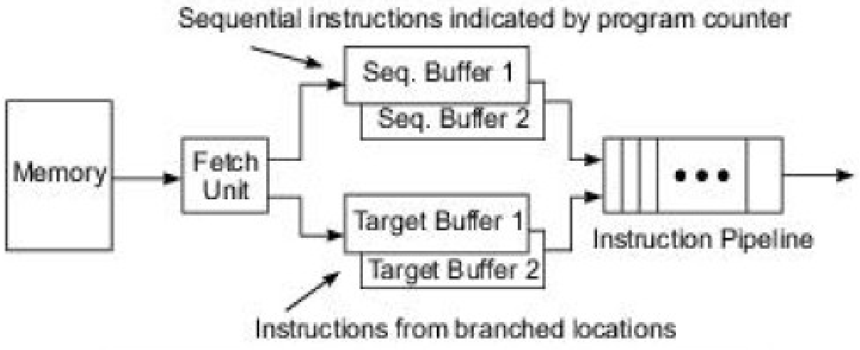


(a) A seven-stage instruction pipeline

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.

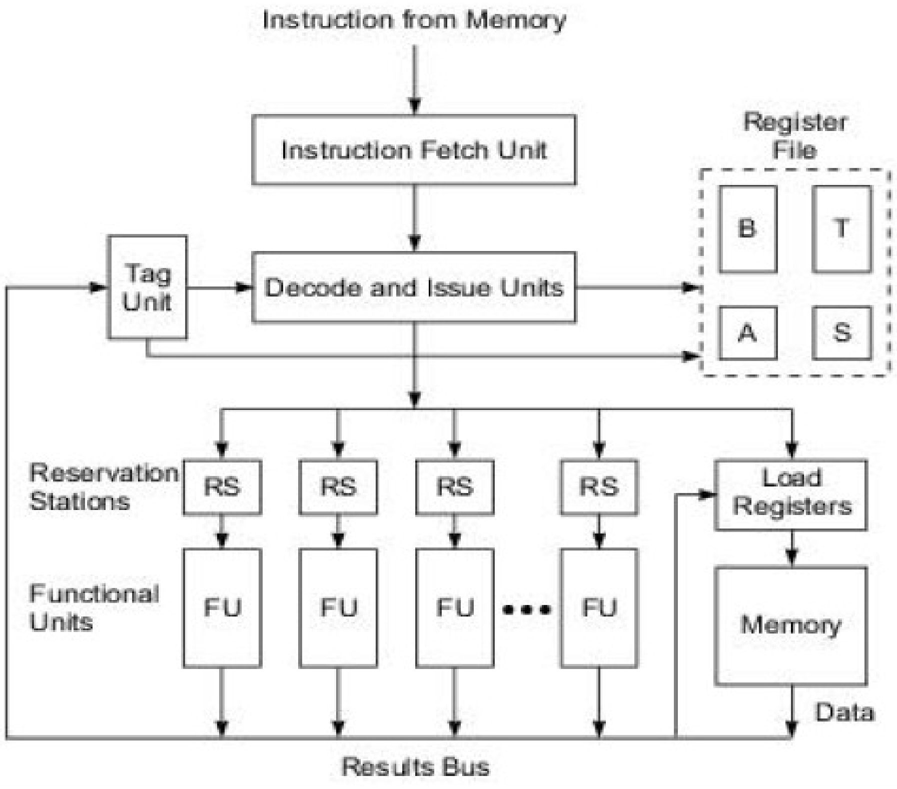


Sequential instructions are loaded into a pair of *sequential buffers* for in-sequence pipelining. Instructions from a branch target are loaded into a pair of *target buffers* for out-of-sequence pipelining. Both buffers operate in a first-in-first-out fashion. These buffers become part of the pipeline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

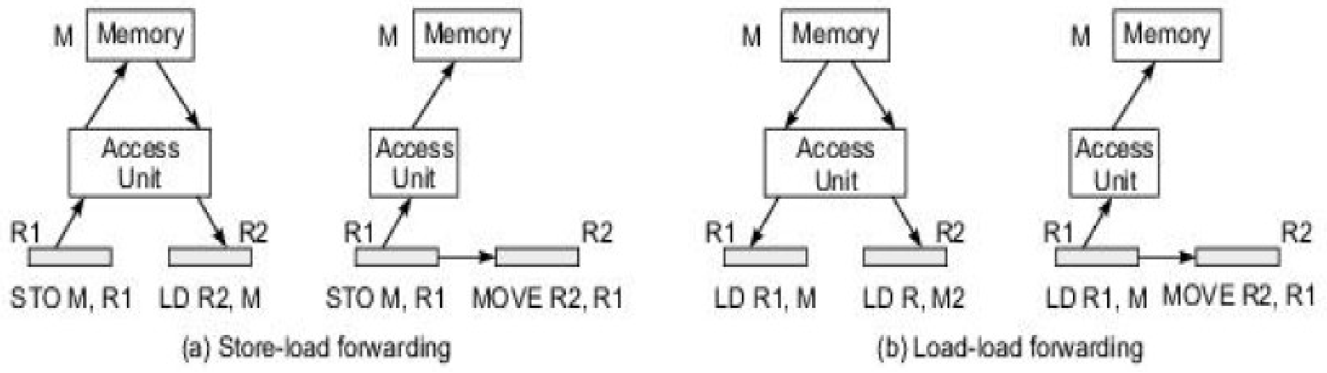
A third type of prefetch buffer is known as a *loop buffer*. This buffer holds sequential instructions contained in a small loop. The loop buffers are maintained by the fetch stage of the pipeline. Prefetched instructions in the loop body will be executed repeatedly until all iterations complete execution. 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

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).



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).



Effect of Branching Three basic terms are introduced below for the analysis of branching effects: The action of fetching a nonsequential or remote instruction after a branch instruction is called *branch taken*. The instruction to be executed after a branch taken is called a *branch target*. The number of pipeline cycles wasted between a branch taken and the fetching of its branch target is called the *delay slot*, denoted by b. In general, $0 \le b \le k-1$, where k is the number of pipeline stages.

When a branch is taken, all the instructions following the branch in the pipeline become useless and will be drained from the pipeline. This implies that a branch taken causes the pipeline to be flushed, losing a number of useful cycles.

These terms are illustrated in Fig. 6.18, where a branch taken causes I_{b+1} through I_{b+k-1} to be drained from the pipeline. Let p be the probability of a conditional branch instruction in a typical instruction stream and q the probability of a successfully executed conditional branch instruction (a branch taken). Typical values of $p = 20\%$ and $q = 60\%$ have been observed in some programs.

The penalty paid by branching is equal to $pqnb\tau$ because each branch taken costs $b\tau$ extra pipeline cycles. Based on Eq. 6.4, we thus obtain the total execution time of n instructions, including the effect of branching, as follows:

$$T_{eff} = k\tau + (n-1)\tau + pqnb\tau$$

we define the following *effective pipeline throughput* with the influence of branching:

$$H_{eff} = \frac{n}{T_{eff}} = \frac{nf}{k+n-1+pqnb} \qquad (6.12)$$

When $n \to \infty$, the tightest upper bound on the effective pipeline throughput is obtained when $b = k - 1$:

$$H^*_{eff} = \frac{f}{pq(k-1)+1} \tag{6.13}$$

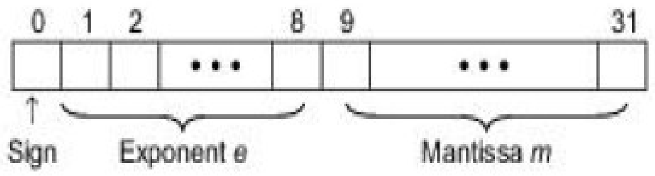
Fixed-Point Operations Fixed-point numbers are represented internally in machines in *sign-magnitude, one's complement*, or *two's complement* notation. Most computers use the two's complement notation because of its unique representation of all numbers (including zero). One's complement notation introduces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



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:

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

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 $e_x \le e_y$ and base $r = 2$.

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

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

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

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

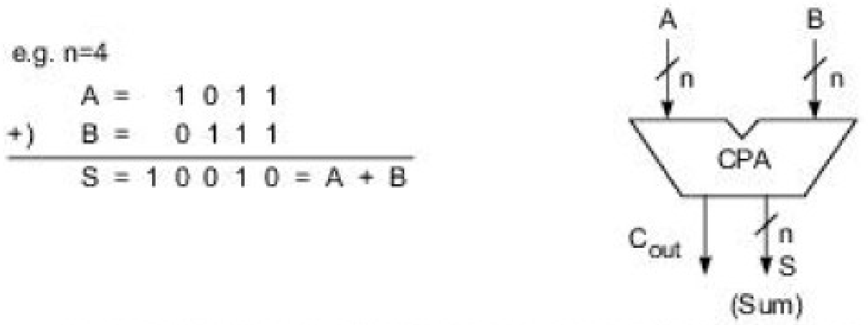
The above equations clearly identify the number of arithmetic operations involved in each floating-point function. These operations can be divided into two halves: One half is for exponent operations such as comparing their relative magnitudes or adding/subtracting them; the other half is for mantissa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

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.

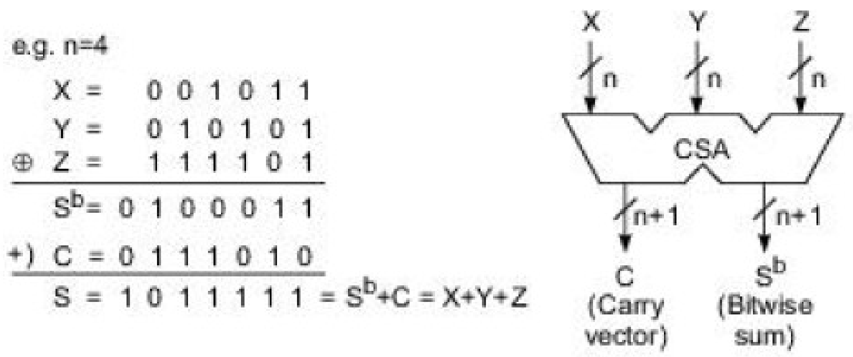
For example, a typical three-stage floating-point adder includes a first stage for exponent comparison and equalization which is implemented with an integer adder and some shifting logic; a second stage for fraction addition using a high-speed carry lookahead adder; and a third stage for fraction normalization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



(a) An *n*-bit carry-propagate adder (CPA) which allows either carry propagation or applies the carry-lookahead technique

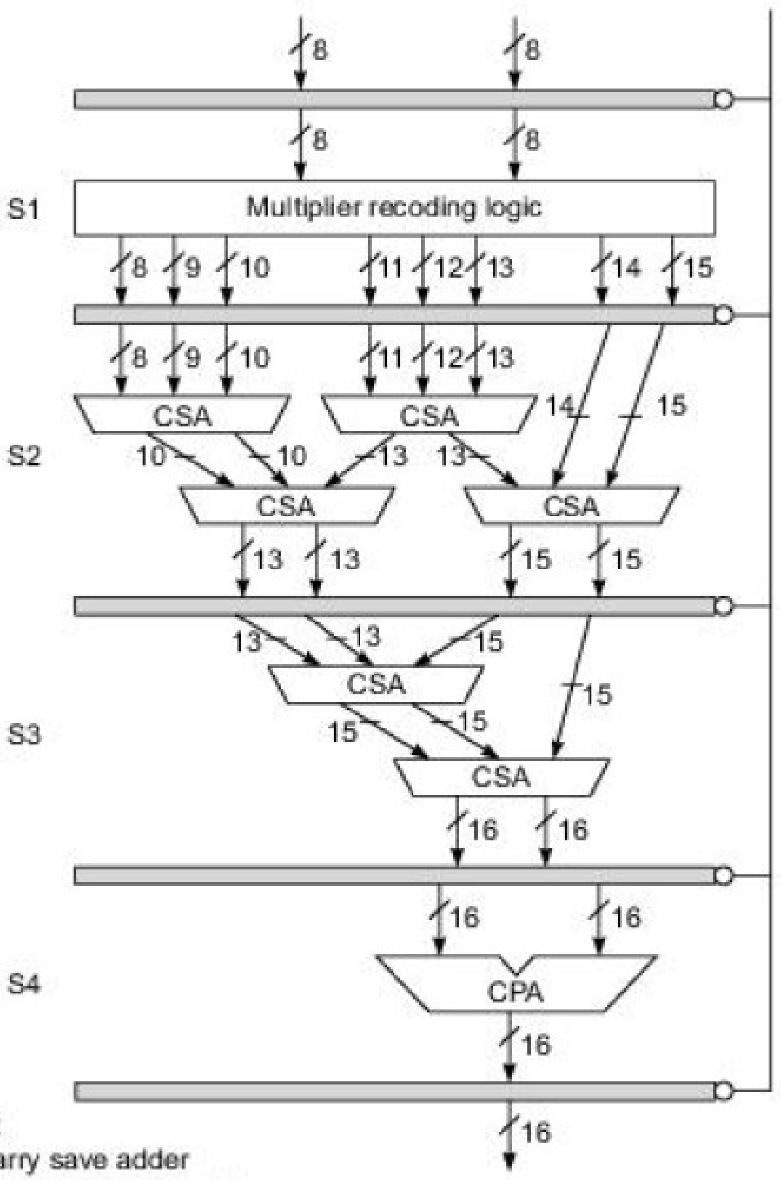


Multiply Pipeline Design Consider as an example the multiplication of two 8-bit integers $A \times B = P$, where P is the 16-bit product. This fixed-point multiplication can be written as the summation of eight partial products as shown below: $P = A \times B = P_0 + P_1 + P_2 + \cdots + P_7$, where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

$$\begin{array}{ccccccccccccccccccc}
 & & & & & & & & & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & = & A \\
 & & & & & & & & \times) & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & = & B \\
\hline
 & & & & & & & & & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & = & P_0 \\
 & & & & & & & & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & = & P_1 \\
 & & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & = & P_2 \\
 & & & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & = & P_3 \\
 & & & & & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & = & P_4 \\
 & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & = & P_5 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & = & P_6 \\
+) & & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & = & P_7 \\
\hline
 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & = & P
\end{array}$$

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.



Captions:
CSA = Carry save adder
CPA = Carry Propagate adder

Cache Coherence Problem

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

Caches in a multiprocessing cnvironrnccnt introduce thc 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: Sharing The cache inconsistency problem occurs only when multiple private caches are used. ln general, three sources of thc problem are identified: sharing of writable data, process migration and I/O activity.

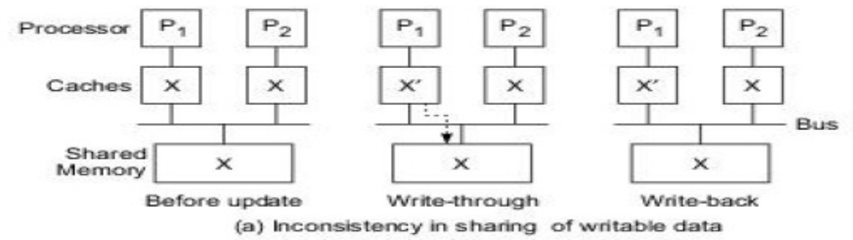
Consider a multiprocessor with two processors, each using a private cache and both sharing the main memory. Let X be a shared data element which has been referenced by both processors. Before update, the three copies of X are consistent.

If processor P. writes new data X’ into the cache, the same copy will be written immediately into the shared memory under a write through policy.

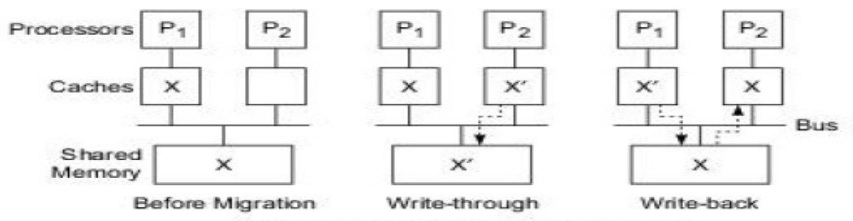
In this case. inconsistency occurs between the two copies {X and X') in the two caches On the other hand, inconsistency may also occur when a write back policy is used, as shown on the right The main memory will be eventually updated when the modified data in the cache are replaced or invalidated

Process Migration and I/O:

The figure shows the shows the occureence 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 processor1 when using write-through caches.



(a) Inconsistency in sharing of writable data



(b) Inconsistency after process migration

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

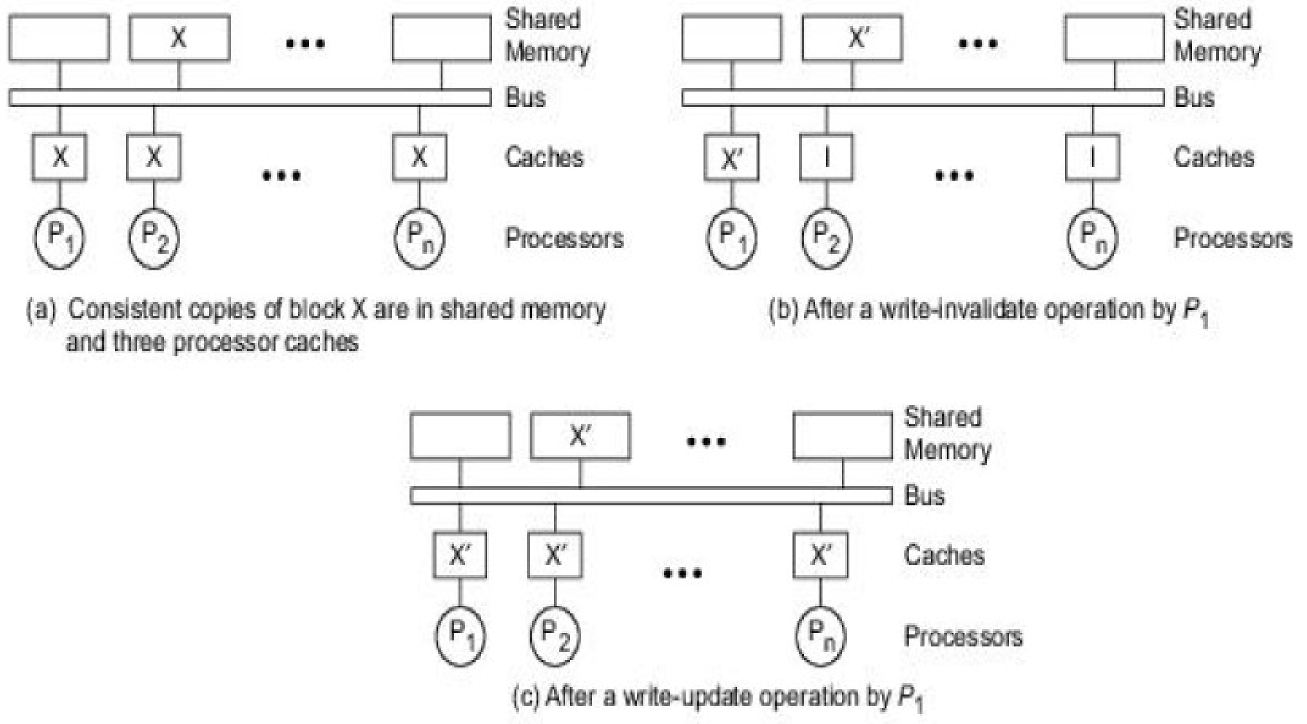
Protocol Approaches

Many of the early commercially available multiproccssors 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 obj ect, the cache controller can take appropriate actions to invalidate thc local copy. Protocols using this mechanism to ensure cohcrcncc arc called snoopy protocols bccausc each cach-c snoops on thc 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 cfficicnt broadcast capability. In such systems, thc cache coherence problem can be solved using some variant of directory schemes.

SNOOPY PROTOCOLS

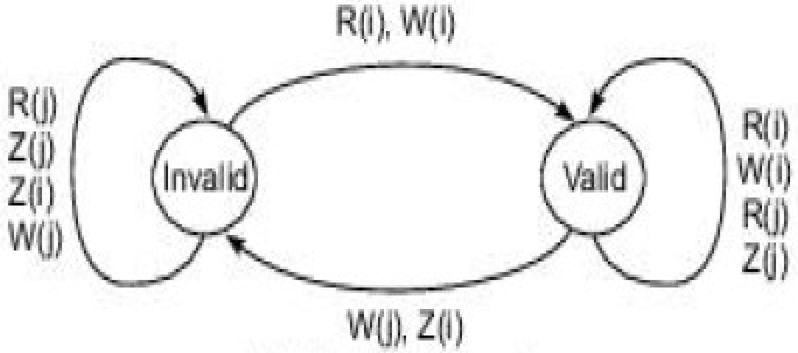
In using private caches associated with processors tied to a common bus, two approaches have been practiced for maintaining cache consist-ecncy:write invalidate and write update policies.Essentially, the write-invalidate policy will invalidate all remote copies when a local cache block is updated. The write update policy will broadcast the new data block to all caches containing a copy of the block.


Shared Memory
Bus
Caches
Processors
X
X'
I
P1
P2
Pn
(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

Using a write-in validate protocol, the processor Pl modifies (writes) its cache from X to X’, and all other copies are invalidated via the bus (denoted I in Fig.). invalidated blocks are sometimes called dirty, meaning they should not be used.

The write update protocol (Fig.c) 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.

Write Through Caches: The states of a cache block copy change with respect to read,write , and replacement operations in thc cache 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 ii can assume one of two possible cache states: valid or invalid.



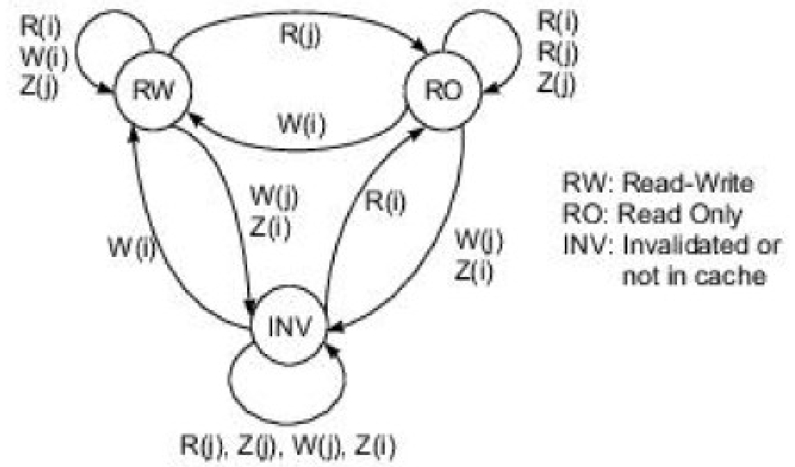
(a) Write-through cache

A remote processor is denoted j, where j # i. For each ofthe two cache states, sis 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 miid valid state , 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 thc block either being invalidated or being replaced (Z(i) or Z(j').

Write Back Caches: The valid‘ state of a write-back cache can be further split into two cache states. Labeled RW and R0 as shown in Fig.. Thc INV (invalidated or not-in-cache} cache state is equivalent to the rm-ora: state mentioned before. This three-state coherence scheme corresponds to an ownership protocol.

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 RD state and every processor having a copy (called a keeper of the copy) can read R(i),R(j) 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 R0 state or the INV state, the cache block becomes uniquely owned when a local write W(i) takes place.



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 $j \neq i$.
R(j) = Read block copy in cache *j* by processor $j \neq i$.
Z(j) = Replace block copy in cache $j \neq i$.

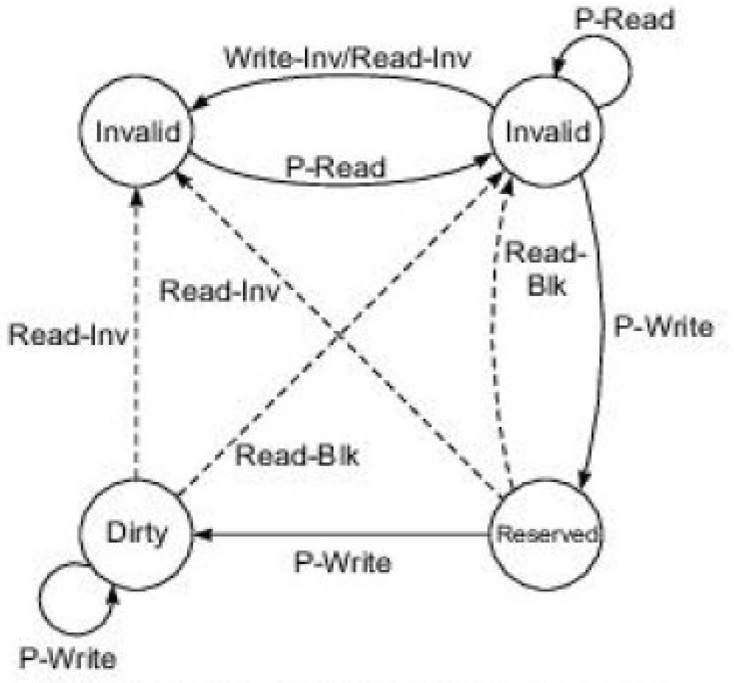
(b) Write-back cache

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:

- *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).


Invalid
Invalid
Dirty
Reserved
Write-Inv/Read-Inv
P-Read
P-Read
Read-Inv
Read-Inv
Read-Blk
Read-Blk
P-Write
P-Write
P-Write

UNIT V : 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.

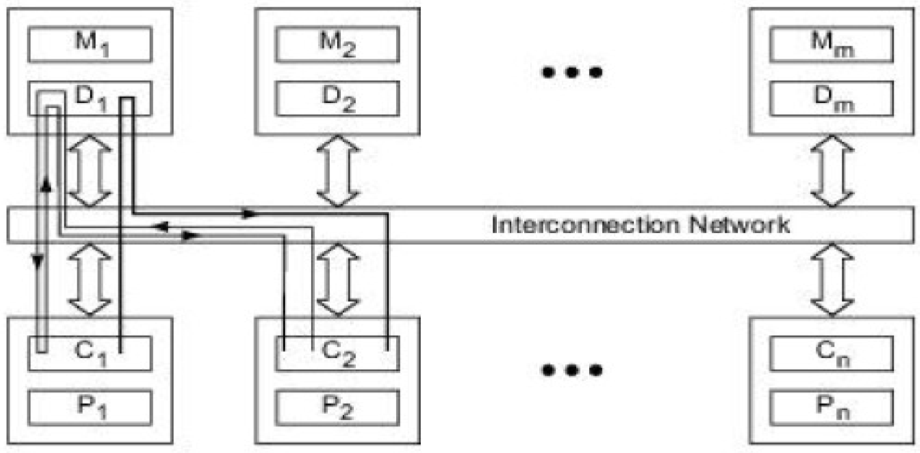
When a multistage or packet switched network is used to build a large multiprocessor with hundreds of processors, the snoopy cache protocols must be modified to suit the network capabilities. Since broadcasting is expensive to perform in such a network, consistency commands will be sent only to those 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.



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.

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.

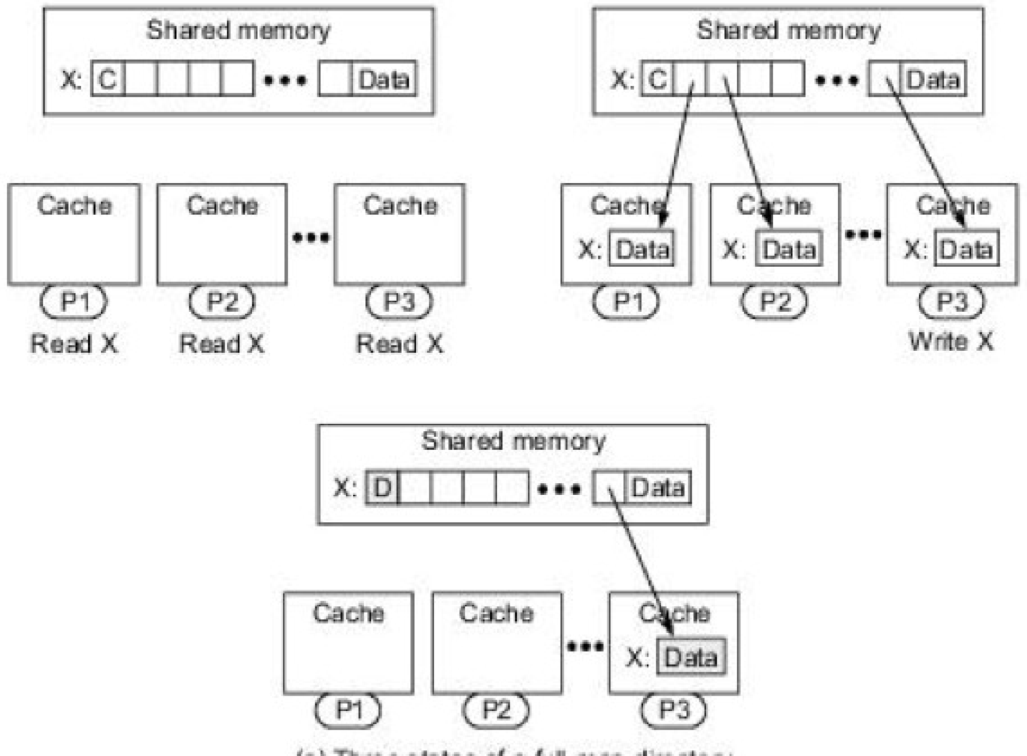
(2) Cache C3 issues a write request to the memory module containing location X and stalls processor P3.

(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.

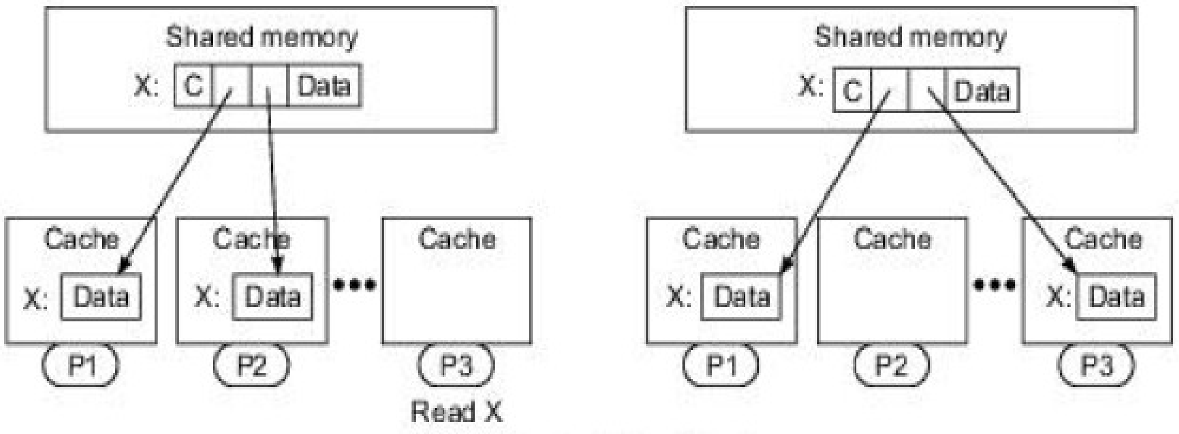
(6) Cache C3 receives the write permission message, updates the state in the cache, and reactivates processor P3.



(a) Three states of a full-map directory

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.



(b) Eviction in a limited directory

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.

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.

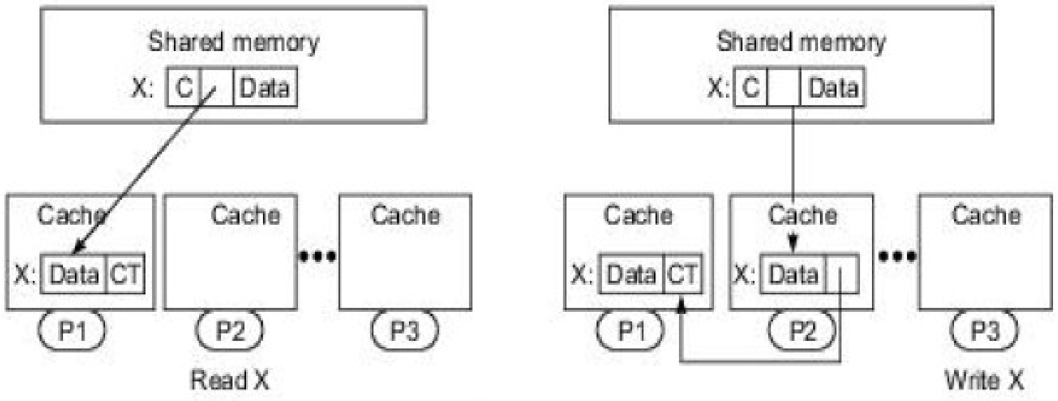
By repeating the above step, all of the caches can cache a copy of the location X. If processor P3 writes to location X, it is necessary to send a data invalidation message down the chain. To ensure sequential consistency, the memory module denies processor P3 write permission until the processor with 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.



(c) The chained directory

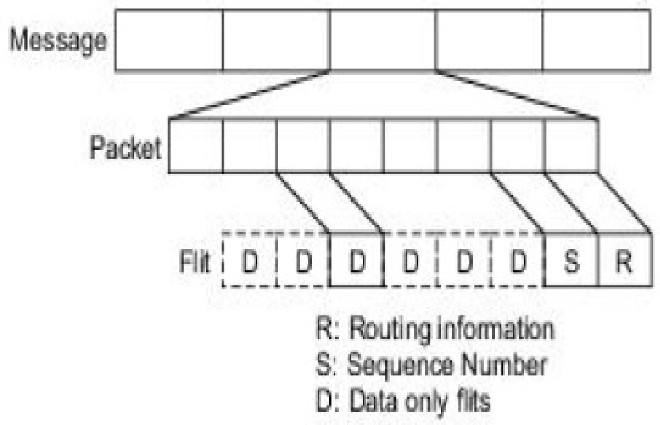
MESSAGE-PASSING MECHANISMS

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.

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.



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.

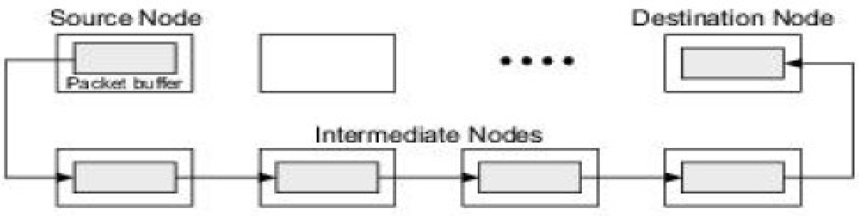
The packet length is determined by the routing scheme and network implementation. Typical packet lengths range from 64 to 512 bits. The sequence number may occupy one to two flits depending on the message length. Other factors affecting the choice of packet and flit sizes include channel bandwidth, router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

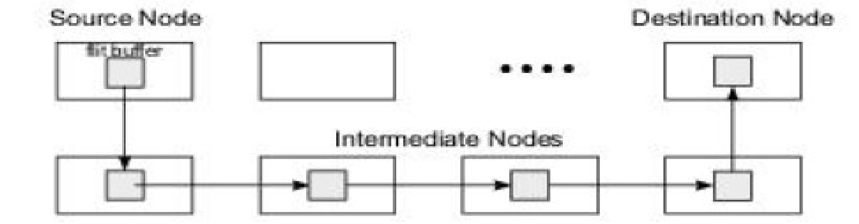
When a packet reaches an intermediate node, it is first stored in the buffer. Then it is forwarded to the next node if the desired output channel and a packet buffer in the receiving node are both available.

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



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.